

# Visión Artificial



Jorge Castell Martínez - 26648116V - Grupo 1.2

Curso 2024/25

# Índice

<u>Calibración</u>	<u>2</u>
<u>Filtros</u>	<u>10</u>
<u>Clasificador</u>	<u>19</u>
<u>SIFT</u>	<u>28</u>
<u>DL</u>	<u>31</u>
<u>Rectificación</u>	<u>34</u>

# Índice de imágenes

- [Imagen 1: Ejemplo de imagen del chessboard.](#)
- [Imagen 2: Imagen tomada de unas fundas de un juego de cartas coleccional.](#)
- [Imagen 3: Diagrama de los apuntes de la asignatura.](#)
- [Imagen 4: Ejecución del script en el eje vertical.](#)
- [Imagen 5: Ejecución del script en el eje horizontal.](#)
- [Imagen 6: Ejecución del programa de filtros de ejemplo usando un filtro Gaussiano en modo ROI.](#)
- [Imagen 7: Ejecución del programa de filtros de ejemplo usando un box filter.](#)
- [Imagen 8: Comparación side by side de las figuras con y sin filtro.](#)
- [Imagen 9: Filtro gaussiano por defecto.](#)
- [Imagen 10: Filtro gaussiano con 3 cascades \(el filtro aplicado 3 veces\)](#)
- [Imagen 11: Comparativa de los algoritmos de convolución manual y de OpenCV.](#)
- [Imagen 12: Clasificación de un objeto con el método de histogramas.](#)
- [Imágenes 13 y 14: Clasificación de objetos con el método Hu Moments.](#)
- [Imágenes 15 y 16: Clasificación de objetos con el método ORB.](#)
- [Imagen 17: Clasificación de un objeto con el método ORB.](#)
- [Imágenes 18 y 19: Clasificación de objetos con el método mediapipe.](#)
- [Imágenes 20 y 21: Clasificación de posturas de una mano con el método hands.](#)
- [Imagen 22: Clasificación de posturas de una mano con el método hands.](#)
- [Imágenes 23 y 24: Clasificación de dos cartas del mismo personaje con el método SIFT.](#)
- [Imágenes 25 y 26: Clasificación de diferentes cartas con el método SIFT.](#)
- [Imágenes 27 y 28: Imagen de entrenamiento 1 junto con su máscara.](#)
- [Imágenes 29 y 30: Imagen de entrenamiento 2 junto con su máscara.](#)
- [Imágenes 31 y 32: Imagen de entrenamiento 3 junto con su máscara.](#)
- [Imágenes 33 y 34: Ejemplos de ejecución de prepare.py.](#)
- [Imagen 35: Resultado de la ejecución del entrenamiento.](#)
- [Imagen 36: Stream de vídeo y la máscara aplicada con el entrenamiento.](#)
- [Imagen 37: Ejemplo de ejecución de\\_rectify\\_plane.py.](#)
- [Imagen 38: Ejemplo de ejecución de rectify\\_plane.py con la imagen rectify\\_3.jpg](#)
- [Imagen 39: Medida real keychron.](#)
- [Imagen 40: Ejemplo de ejecución de rectify\\_plane.py con la imagen rectify\\_4.jpg](#)
- [Imagen 41: Medida real navaja.](#)
- [Imagen 42: Ejemplo de ejecución de rectify\\_plane.py con la imagen rectify\\_5.jpg](#)
- [Imagen 43: Ejemplo de ejecución de rectify\\_plane.py con la imagen rectify\\_6.jpg](#)
- [Imagen 44: Medida real dados.](#)

# Calibración

- 1) Calibra tu cámara mediante múltiples imágenes de un *chessboard* siguiendo las instrucciones del README en `code/calibrate`. Usa el modo standard o por defecto de la cámara y toma nota de su resolución (W x H).

He tomado imágenes del chessboard `pattern.png` con una webcam virtual a partir de mi teléfono usando OBS, y he fijado el tamaño en 800x600. Luego, he ejecutado `./calibrate.py -dev 'dir:virtualwebcam/*.png' > resultado`, lo cual me ha proporcionado un fichero con los datos necesarios sobre el error de ajuste, la matriz  $K$  y los coeficientes de distorsión radial:

```
800x600 25.0fps
ok
1
RMS: 0.14417152328765143
camera matrix:
[[732.  0.  408.]
 [ 0.  731.  311.]
 [ 0.  0.  1.]]
distortion coefficients: [ 1.590e-01 -9.960e-01 -1.000e-03
-0.000e+00  1.456e+00]
```



Imagen 1: Ejemplo de imagen del chessboard.

Con estos datos, podemos proceder con los siguientes ejercicios.

- 2) Con el parámetro  $f$  obtenido en la calibración, determina el campo visual (FOV, *field of view*) horizontal y vertical.

Usando la fórmula  $\tan\left(\frac{FOV}{2}\right) = \frac{\frac{w}{2}}{f}$  podemos averiguar el campo visual tanto vertical como horizontal de la cámara utilizada sabiendo que  $f = 732$ :

$$\text{FOV horizontal} = 2 * \arctan\left(\frac{W}{2f}\right) \approx 57.31^\circ$$

$$\text{FOV vertical} = 2 * \arctan\left(\frac{H}{2f}\right) \approx 44.63^\circ$$

- 3) Comprueba que estos datos son consistentes con el tamaño de la imagen que obtienes con un objeto de tamaño conocido y situado a una distancia conocida.

Podemos usar la herramienta `medidor.py` situada en `code/util` y medimos algo de lo que sepamos su medida:



Imagen 2: Imagen tomada de unas fundas de un juego de cartas colecciónable.

Ahora, sabiendo que la medida exacta de las fundas de la imagen anterior es de 91mm ( $X=9.1\text{cm}$ ), su medida aproximada en pixels es de  $u=327$  pixels y la altura a la que se ha tomado la imagen es a  $Z=20\text{cm}$ .

Ahora, despejando  $u = f \frac{X}{Z}$ , tenemos  $f = \frac{uZ}{X} = \frac{327*20}{9.1} = 718.68$  pixels. Sabiendo que la  $f$  que hemos extraído de `calibrate.py` es de 732, calculamos que ha habido un porcentaje de diferencia de un **1.85%**.

**4) Calcula aproximadamente:**

- a) el tamaño en pixels que tendrá una persona situada a 10m de la cámara.

Para resolver este problema, hacemos uso de la fórmula de calibración aproximada:

$u = f \frac{X}{Z}$  donde  $f$  es la distancia focal extraída de la matriz  $K$ ,  $X=170\text{cm}$

(aproximadamente la altura de una persona) y  $Z=1000\text{cm}$ , de modo que:

$$u = 732 \frac{170}{1000} = 124.44 \text{ pixels.}$$

- b) A qué distancia se encuentra un coche que en la imagen tiene 20px de ancho.

Para resolver este otro problema utilizamos la misma fórmula, donde ahora tenemos  $u=20$  pixels,  $f=732$  pixels y tamaño de un coche ( $X\approx180\text{cm}$ ).

Despejando  $u = f \frac{X}{Z}$ , tenemos  $z = f \frac{X}{u}$ , por lo tanto:

$$z = 732 \frac{180}{20} = 6588 \text{ cm} = 65.9 \text{ metros.}$$

- 5) Determina a qué altura hay que poner la cámara para obtener una vista cenital completa de un campo de baloncesto.**

Las medidas de un campo de baloncesto son 28x15m, por lo que seleccionamos la medida mayor como nuestro valor  $X (=2800\text{cm})$ , teniendo ya  $f=732$  pixels y tomaremos el ancho de nuestra cámara como el espacio en pixel que ocupará el campo en la cámara ( $u=800$  pixels).

Por lo tanto, despejando la fórmula  $u = f \frac{X}{Z} \Rightarrow Z = f \frac{X}{u}$  tenemos que  $Z = 732 \frac{2800}{800} = 2562 \text{ cm} = 25.62$  metros de altura.

- 6) Calcula el tamaño en pixels de la pelota cuando está en el suelo.**

Ahora nos piden calcular  $u$ , por lo que usaremos de las medidas anteriores el resto de variables,  $f=732$  pixels,  $Z = 2562 \text{ cm}$  y  $X=61\text{cm}$  (tamaño de una pelota estándar).

Ahora, usando  $u = f \frac{X}{Z} = 732 \frac{61}{2562} = 17.43$  pixels

- 7) Escribe una función que devuelva la altura de la pelota sobre el suelo a partir de su diámetro aparente en pixels.**

Este problema se puede resolver relativamente rápido a partir de los datos anteriores, así que primero explicaré cómo resolverlo razonadamente y luego dejaré el fragmento de código.

Para resolver este problema tenemos una nueva  $u$  (argumento variable), por lo que, sabiendo que  $X$  se mantiene constante (61cm), podemos calcular  $Z$  para saber la nueva altura a la que estará la pelota usando la fórmula  $Z = f \frac{X}{u}$ . Una vez calculada, simplemente debemos calcular la diferencia entre la  $Z$  original (la altura de la cámara para ver toda la pista, 2562 cm) y la nueva  $Z$ , que es la distancia de la cámara a la pelota, lo cual nos devolverá la distancia de la pelota al suelo. El código en Python es el siguiente:

```

camera_height = 2562 #centimeters
ball_width = 61 #centimeters
focal_length = 732 #pixels

def calculate_ball_height(pixel_diameter):
    if pixel_diameter <= 0:
        raise ValueError("El diámetro en píxeles debe ser mayor que 0.")

    new_height = focal_length * (ball_width / pixel_diameter)

    # La distancia al suelo es la altura de la cámara menos Z
    distance_to_floor = camera_height - new_height

    return distance_to_floor

```

**8) Haz una aplicación para medir el ángulo que definen dos puntos marcados con el ratón en la imagen.**

Para comenzar, voy a basar mi script en [medidor.py](#), añadiendo los cálculos disponibles en la última página del notebook [imagen.ipynb](#), donde tenemos:

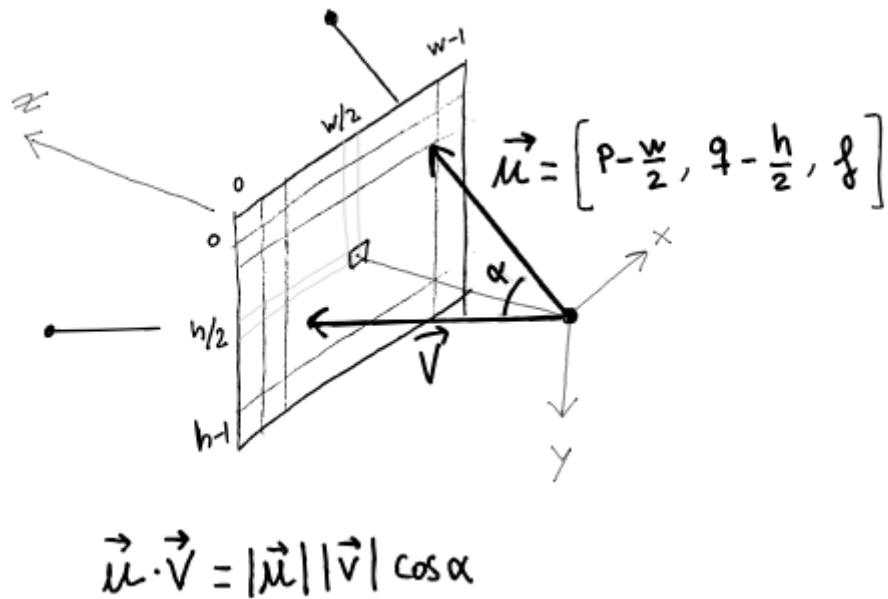


Imagen 3: Diagrama de los apuntes de la asignatura

En el siguiente diagrama se nos explica cómo poder extraer el ángulo que forman dos puntos en una cámara de modelo pinhole. En concreto, este diagrama asume que el eje óptico de la cámara pasa por el centro exacto de la imagen. De esta manera, el cálculo, aunque sea aproximado, es más sencillo para nosotros, por lo que para realizar el script, nosotros hemos hecho lo mismo.

Una vez extraídos los vectores  $u$  y  $v$ , podemos utilizar la fórmula del producto escalar de dos vectores (que aparece en la imagen) para extraer el ángulo que forman.

A continuación se deja el script creado:

```
#!/usr/bin/env python

#This script is based on the one found in /code/util/medidor.py

import cv2 as cv
from umucv.stream import autoStream
from collections import deque
import numpy as np
from umucv.util import putText

points = deque(maxlen=2)

def fun(event, x, y, flags, param):
    if event == cv.EVENT_LBUTTONDOWN:
        points.append((x,y))

cv.namedWindow("webcam")
cv.setMouseCallback("webcam", fun)

focal_length = 732 #pixels
camera_width = 800
camera_height = 600

for key, frame in autoStream():
    for p in points:
        cv.circle(frame, p, 3, (0,0,255), -1)
    if len(points) == 2:
        cv.line(frame, points[0],points[1],(0,0,255))
        c = np.mean(points, axis=0).astype(int)
        #Lo calculamos de manera aproximada, asumiendo que el eje óptico pasa
        por el centro de la matriz de pixels (en este caso
        # asumimos que el centro estará en el punto (400,300))
        u = [(points[0])[0] - camera_width/2, (points[0])[1] -
        camera_height/2, focal_length]
        v = [(points[1])[0] - camera_width/2, (points[1])[1] -
        camera_height/2, focal_length]
        a = np.arccos(np.dot(u,v)/(np.linalg.norm(u)*np.linalg.norm(v)))
        a = np.rad2deg(a)
        putText(frame,f'{a:.5f}°',c)

    cv.imshow('webcam',frame)
```

Si ahora utilizamos el script y hacemos una linea de extremo a extremo en el eje vertical y el horizontal, en teoría deberíamos de obtener nuestro FOV:

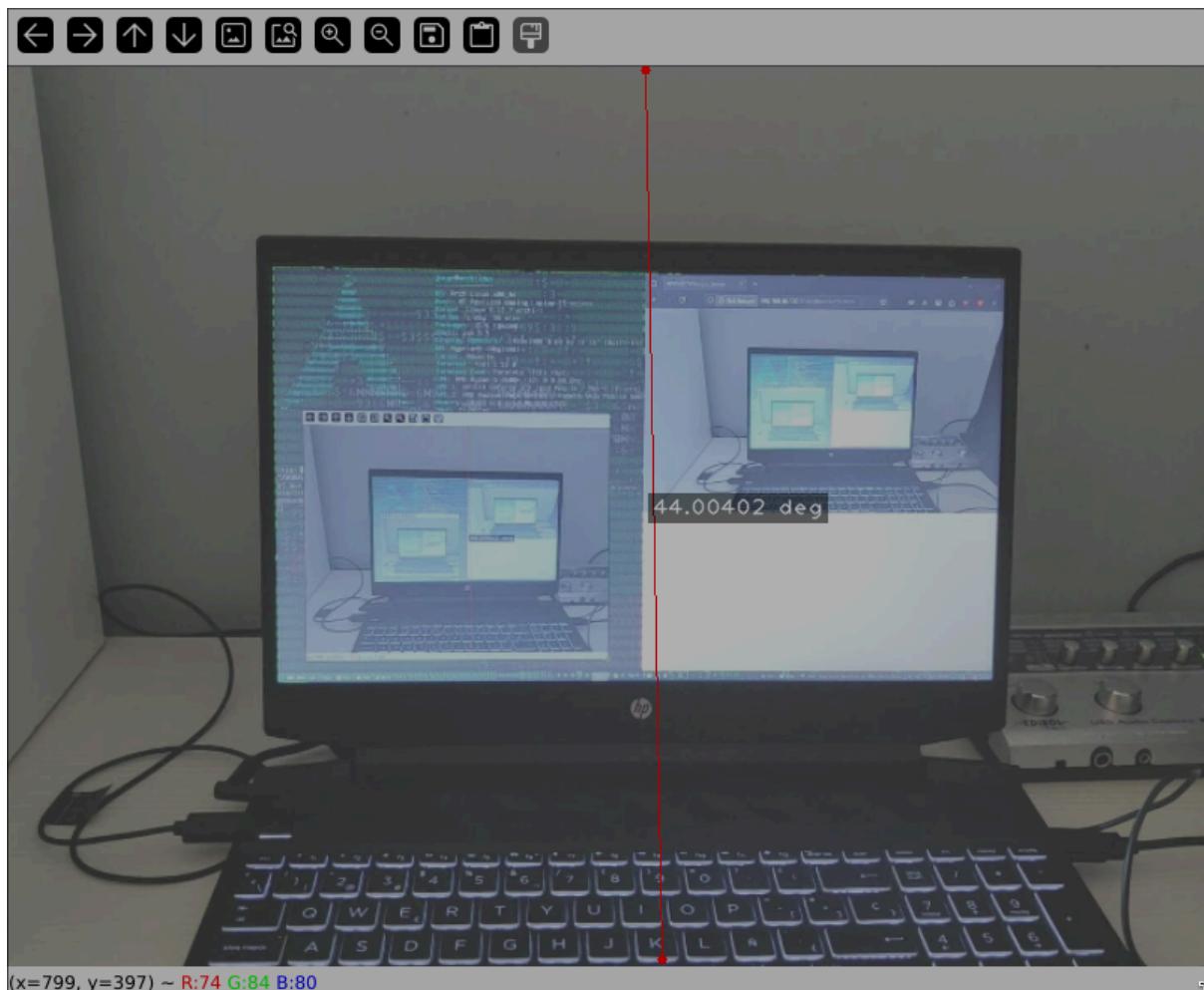


Imagen 4: Ejecución del script en el eje vertical

Viendo los resultados de los apartados anteriores, sabemos que nuestro FOV vertical era de aproximadamente  $44.63^\circ$ , lo que se corresponde casi al 100% con la imagen anterior.

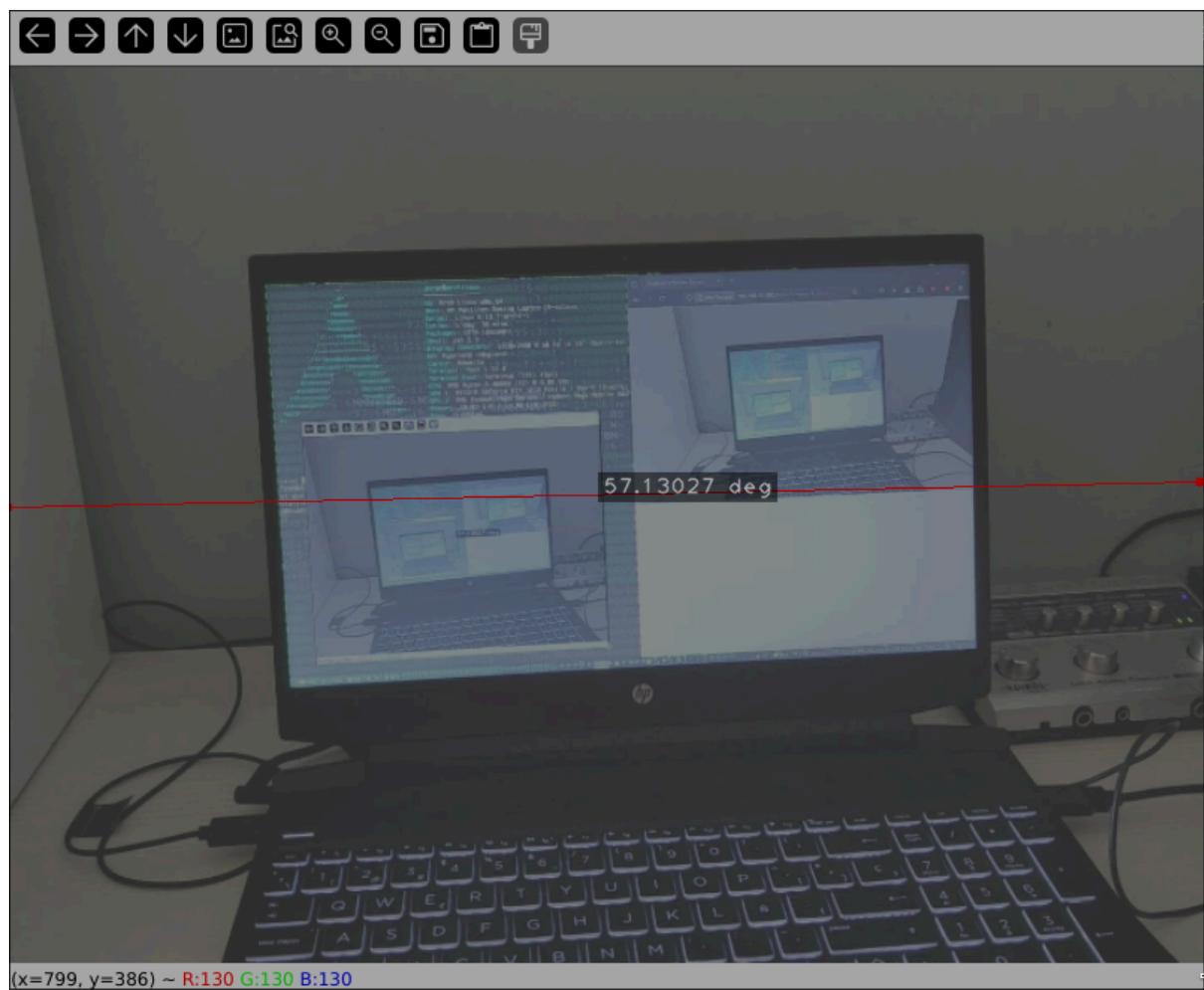


Imagen 5: Ejecución del script en el eje horizontal.

Con esta otra ejecución y consultando apartados anteriores, vemos que el FOV horizontal es de  $57.31^\circ$ , lo cual nuevamente casi se corresponde al 100% con la imagen anterior.

# Filtros

- 1) Muestra en vivo el efecto de diferentes filtros, seleccionando con el teclado el filtro deseado y modificando sus parámetros (p.ej. el nivel de suavizado) con trackbars. Aplica el filtro en un ROI para comparar el resultado con el resto de la imagen ([ejemplo](#)). Diseña el ejercicio de forma que sea fácilmente ampliable.

Para realizar este ejercicio me he basado en los scripts `help_window.py`, `roi.py`, `box_vs_gaussian.py`. He creado una variable para saber qué filtro está seleccionado, así como las variables ‘toggle’ para el color, inversión de imagen y modo ROI. Luego, tras implementar la lógica del teclado y el mensaje de ayuda, he creado una subrutina `apply_filter(img)`, que simplemente aplica los varios filtros disponibles que hay en el paquete cv añadiendo como parámetros los valores de los trackbar (si aplica).

```
#!/usr/bin/env python

import cv2 as cv
import numpy as np
from umucv.stream import autoStream
from umucv.util import Help, ROI

help = Help(
    """
    BLUR FILTERS

    0: do nothing
    1: box
    2: Gaussian
    3: median
    4: bilateral
    5: min
    6: max

    c: color/monochrome
    r: only roi
    i: invert on/off

    SPC: pause

    h: show/hide help
    """
)

color = False
invert = False
current_filter = 0
paused = False
roi_mode = False

# Parámetros para los filtros
```

```

cv.namedWindow('result')
cv.createTrackbar('Box Size', 'result', 15, 100, lambda v: None)
cv.createTrackbar('Gaussian Sigma', 'result', 3, 100, lambda v: None)
cv.createTrackbar('Min/Max Size', 'result', 15, 100, lambda v: None)

# Inicializar ROI
region = ROI("result")

for key, frame in autoStream():
    help.show_if(key, ord('h'))

    if key == ord('0'):
        current_filter = 0
    if key in [ord(str(i)) for i in range(1, 7)]:
        current_filter = int(chr(key))
    if key == ord('c'):
        color = not color
    if key == ord('i'):
        invert = not invert
    if key == ord('r'):
        roi_mode = not roi_mode
    if key == ord(' '):
        paused = not paused

    if paused:
        continue

    if not color:
        result = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
    else:
        result = frame

    if invert:
        result = 255 - result

#Valores de los trackbar
box_ksize = cv.getTrackbarPos('Box Size', 'result') or 1
gaussian_sigma = cv.getTrackbarPos('Gaussian Sigma', 'result')
min_max_ksize = cv.getTrackbarPos('Min/Max Size', 'result') or 1

def apply_filter(img):
    if current_filter == 1:
        return cv.boxFilter(img, -1, (box_ksize, box_ksize))
    elif current_filter == 2:
        return cv.GaussianBlur(img, (0, 0), gaussian_sigma)
    elif current_filter == 3:
        return cv.medianBlur(img, 15)
    elif current_filter == 4:
        return cv.bilateralFilter(img, 9, 75, 75)
    elif current_filter == 5:
        kernel = np.ones((min_max_ksize, min_max_ksize), np.uint8)
        return cv.erode(img, kernel)

```

```

        elif current_filter == 6:
            kernel = np.ones((min_max_ksize, min_max_ksize), np.uint8)
            return cv.dilate(img, kernel)

    return img

#Cambia entre aplicar a toda la imagen y el ROI en amarillo
if roi_mode and region.roi:
    x1, y1, x2, y2 = region.roi
    roi = result[y1:y2 + 1, x1:x2 + 1]
    filtered_roi = apply_filter(roi)
    result[y1:y2 + 1, x1:x2 + 1] = filtered_roi
    cv.rectangle(result, (x1, y1), (x2, y2), color=(0, 255, 255),
thickness=2)
else:
    result = apply_filter(result)

cv.imshow('result', result)

```

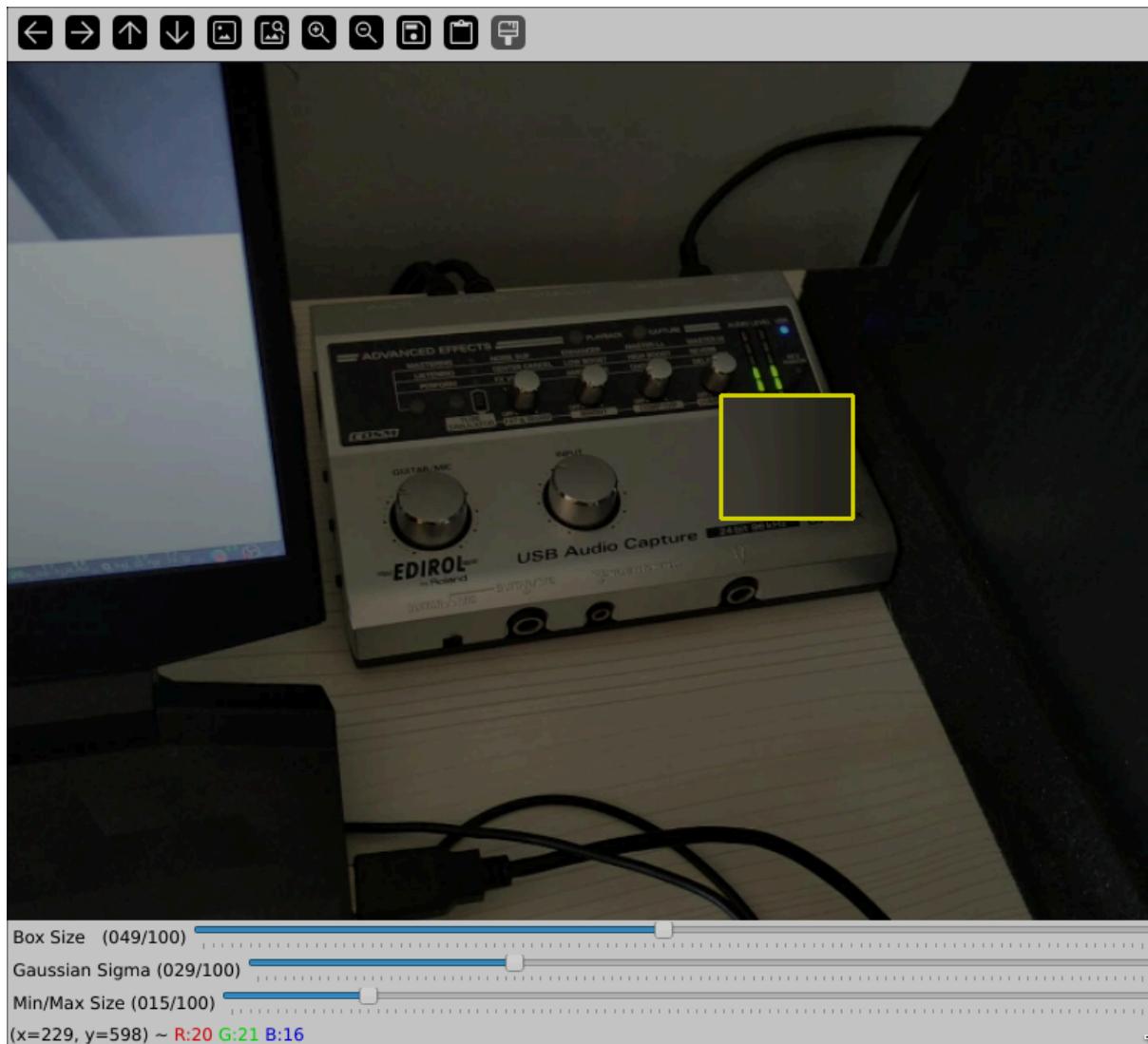


Imagen 6: Ejecución del programa de filtros de ejemplo usando un filtro Gaussiano en modo ROI.

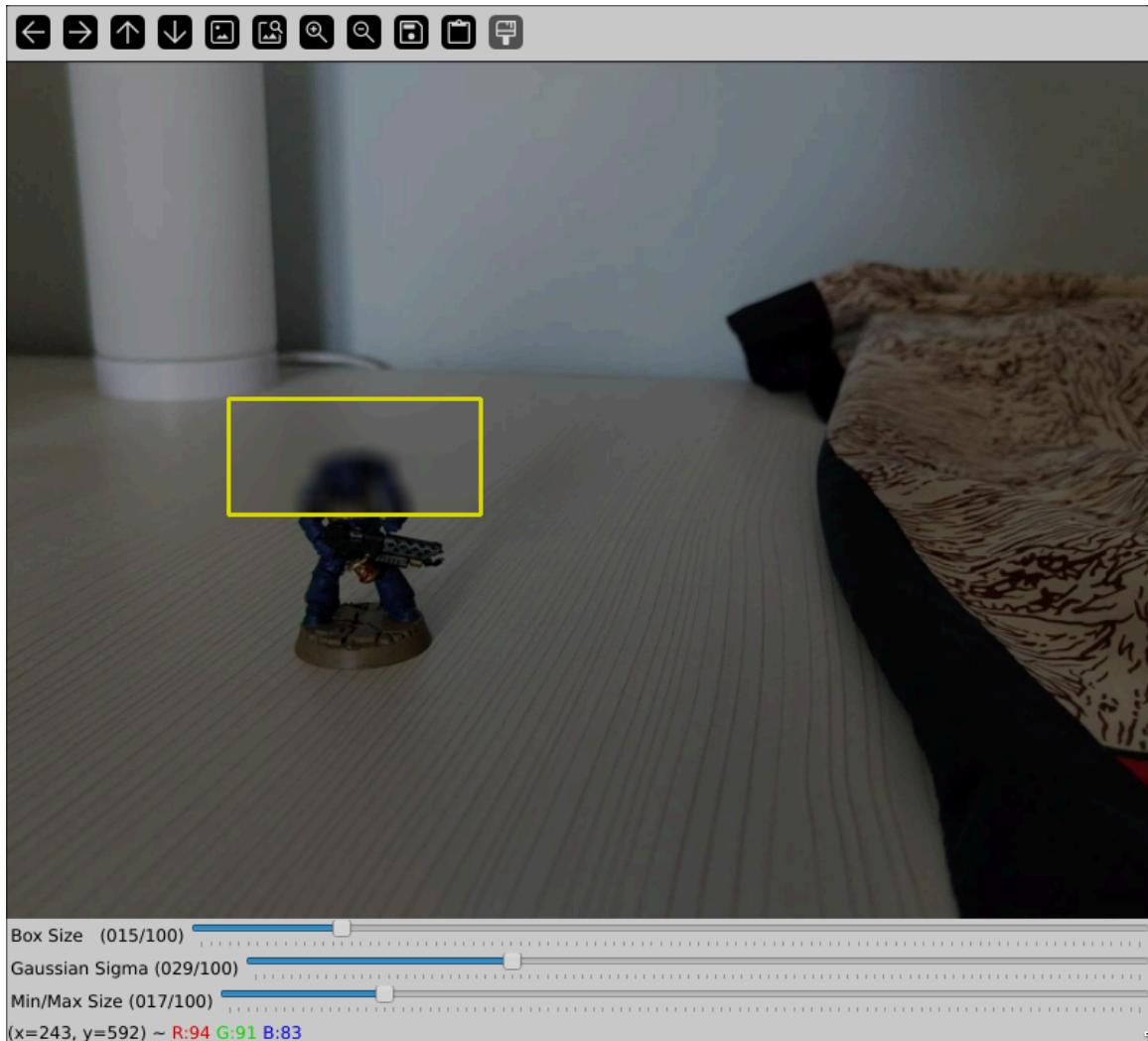


Imagen 7: Ejecución del programa de filtros de ejemplo usando un box filter.

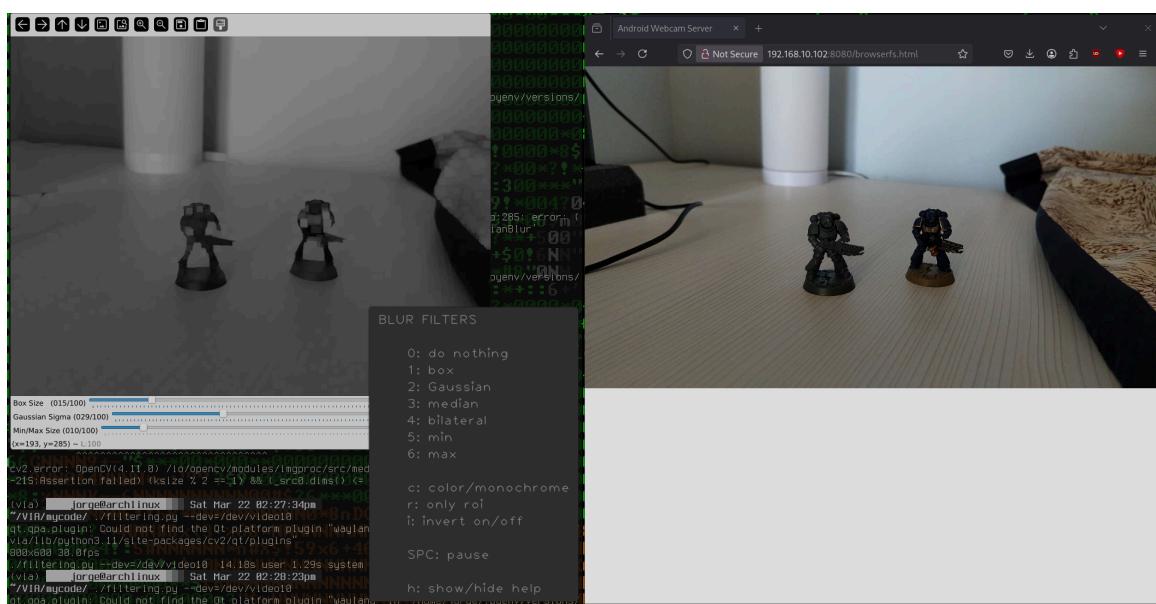


Imagen 8: Comparación side by side de las figuras con y sin filtro (filtro max).

## 2) Comprueba la propiedad de "cascading" del filtro gaussiano.

Para realizar esto, introducimos este pequeño cambio en el código del ejercicio anterior:

```
for _ in range(gaussian_cascading):
    ksize = max(1, 2 * int(3 * gaussian_sigma) + 1)
    filtered = cv.GaussianBlur(filtered, (ksize, ksize), gaussian_sigma)
```

Si realizamos esto en el caso del filtro 2 y añadimos un trackbar para el cascading, tendremos ahora esencialmente un “multiplicador de filtro”, de forma que podemos suavizar la imagen sin tener que alterar el sigma, que es mucho más sensible.

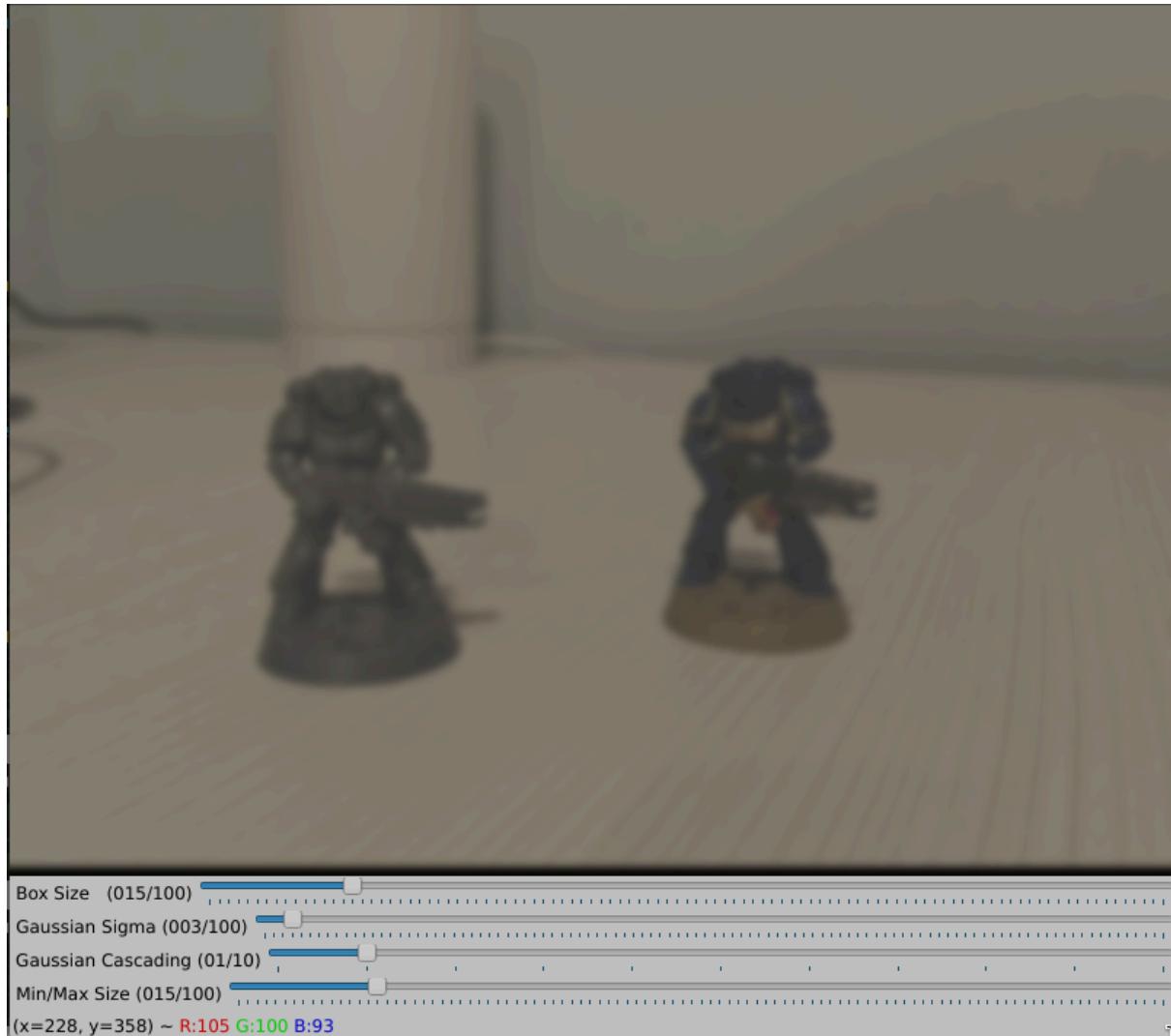


Imagen 9: Filtro gaussiano por defecto.

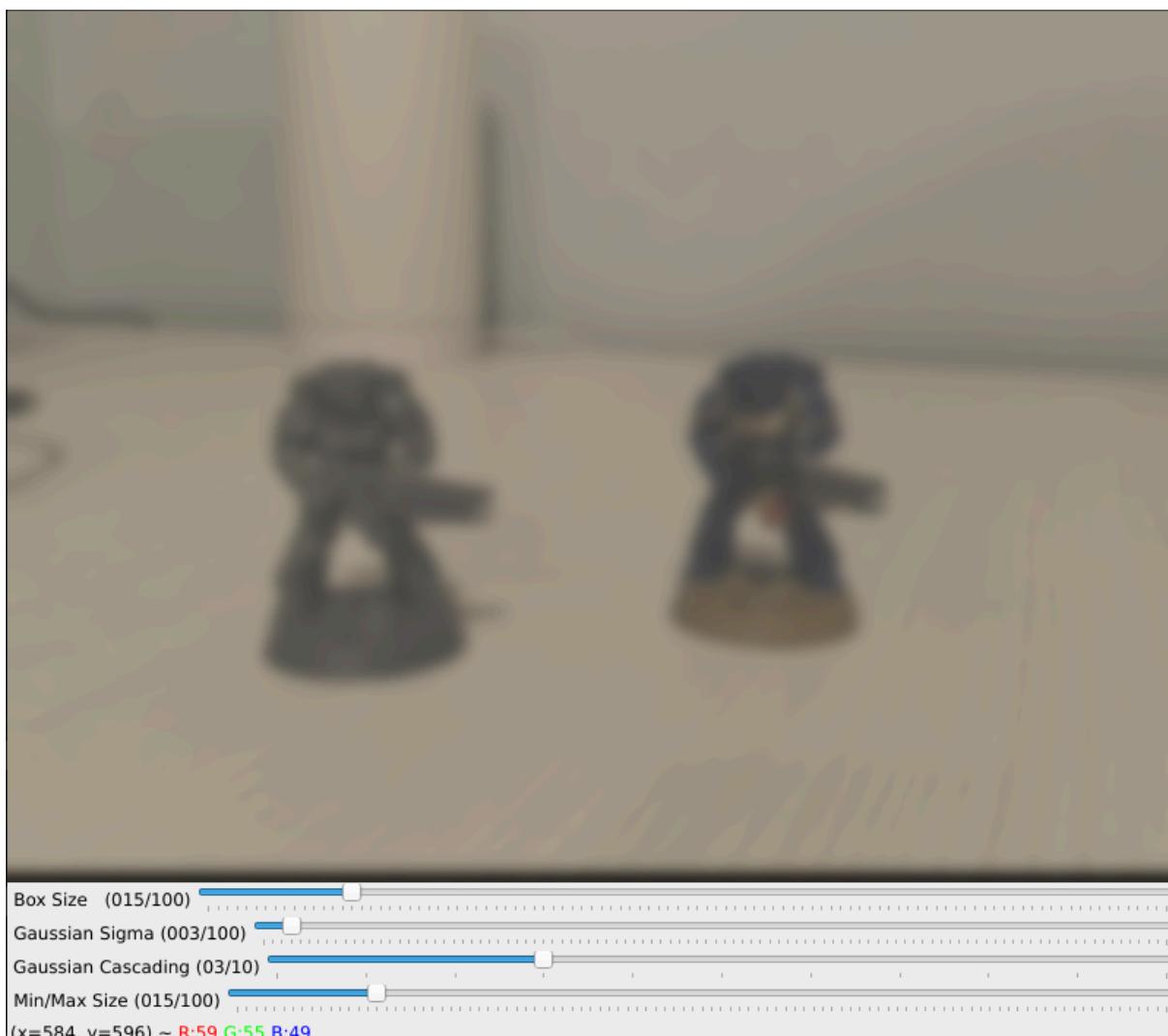


Imagen 10: Filtro gaussiano con 3 cascades (el filtro aplicado 3 veces).

### 3) Comprueba la propiedad de "separabilidad" del filtro gaussiano.

Para poder comprobar la separabilidad, la añadimos también un nuevo fragmento al código anterior:

```
if gaussian_separability:
    kernel = cv.getGaussianKernel(ksize, gaussian_sigma)
    filtered = cv.sepFilter2D(filtered, -1, kernel, kernel)
else:
    filtered = cv.GaussianBlur(filtered, (ksize, ksize), gaussian_sigma)
```

\*se ha asociado la variable `gaussian_separability` la tecla `g`, con sus respectivas modificaciones en el bucle de `autoStream()`.

Para comprobar el funcionamiento, simplemente aplicamos un sigma absurdamente alto, para que en nuestro `video_stream` se genere mucho lag y apretamos la tecla 's'. Si todo va bien, debería de funcionar de manera mucho más fluida. Esto se debe a que la separabilidad se puede traducir en términos generales como una aplicación más eficiente de los filtros, siendo en este caso, en vez de una aplicación a toda la matriz de pixels, una aplicación fila a fila, por lo que [según internet](#) se pasa de un orden general de  $O(n^2)$  a un

orden de  $O(2N)$ . Debido a la naturaleza de la demostración, no puedo adjuntar pruebas de esto.

#### 4) Implementa el box filter con la imagen integral.

Para implementar el box filter, he buscado [cómo podría implementarse](#), por lo que he empezado construyendo la imagen integral del stream, convirtiéndola a float32 para que no dé error. Tras esto, he lanzado un bucle doble para recorrer cada pixel de la imagen y aplicarle el filtro integral. La diferencia no es muy notoria con el de OpenCV, así que he buscado diferencias, y aparentemente el integral es más eficiente cuando el tamaño de caja es muy grande.

```
def box_filter_integral(img, ksize):
    if ksize % 2 == 0:
        ksize += 1
    r = ksize // 2

    # Si es color, aplicar por canal
    if len(img.shape) == 3:
        channels = cv.split(img)
        filtered = [box_filter_integral(c, ksize) for c in channels]
        return cv.merge(filtered)

    # Convertir a float32 para evitar overflow
    img = img.astype(np.float32)
    h, w = img.shape
    integral = cv.integral(img)[1:, 1:]

    result = np.zeros_like(img)

    for y in range(h):
        y1 = max(y - r, 0)
        y2 = min(y + r, h - 1)
        for x in range(w):
            x1 = max(x - r, 0)
            x2 = min(x + r, w - 1)

            A = integral[y1 - 1, x1 - 1] if y1 > 0 and x1 > 0 else 0
            B = integral[y1 - 1, x2] if y1 > 0 else 0
            C = integral[y2, x1 - 1] if x1 > 0 else 0
            D = integral[y2, x2]

            area = (y2 - y1 + 1) * (x2 - x1 + 1)
            result[y, x] = (D - B - C + A) / area

    return result.astype(img.dtype)
```

**5) Implementa en Python desde cero (usando bucles) el algoritmo de convolución con una máscara general y compara su eficiencia con la versión de OpenCV.**

Para realizar este apartado, he buscado en internet cómo funciona un filtro de convolución, viendo que funciona como una multiplicación con un ‘kernel’, o máscara como lo conocemos en la asignatura. ChatGPT también me ha dicho que aplique el padding que se ve en el código para que el filtro se pueda aplicar a toda la imagen, para que las zonas cercanas al borde también tengan el filtro. He encontrado la máscara usada en StackOverflow, y como no modifica drásticamente la imagen la he dejado. Tras esto, he comprobado los tiempos de ambos y los he mostrado tanto en pantalla como en la propia imagen. A continuación está el código:

```
#!/usr/bin/env python

import cv2 as cv
import numpy as np
import time
from umucv.util import putText


def manual_convolution(image, kernel):
    h, w = image.shape
    kh, kw = kernel.shape
    pad_h, pad_w = kh // 2, kw // 2
    result = np.zeros_like(image, dtype=np.float32)

    padded_image = np.pad(image, ((pad_h, pad_h), (pad_w, pad_w)), mode='constant', constant_values=0)

    # Convolución manual
    for i in range(h):
        for j in range(w):
            region = padded_image[i:i + kh, j:j + kw]
            result[i, j] = np.sum(region * kernel)

    return result


image = cv.imread('image.png', cv.IMREAD_GRAYSCALE)

# Kernel de internet
kernel = np.array([[2, 4, 2], [4, 8, 4], [2, 4, 2]], dtype=np.float32) / 16

# Medidas de tiempo

start_manual = time.time()
manual_result = manual_convolution(image, kernel)
end_manual = time.time()

start_cv = time.time()
opencv_result = cv.filter2D(image, -1, kernel)
end_cv = time.time()
```

```

manual_time = (end_manual - start_manual) * 1000 # Milisegundos
opencv_time = (end_cv - start_cv) * 1000

print(f'Tiempo manual: {manual_time:.2f} ms')
print(f'Tiempo OpenCV: {opencv_time:.2f} ms')

cv.imshow('Original', image)
putText(manual_result, f'Manual: {manual_time:.2f} ms')
cv.imshow('Manual Convolution', np.uint8(manual_result))
putText(opencv_result, f'OpenCV: {opencv_time:.2f} ms')
cv.imshow('OpenCV Convolution', opencv_result)
cv.waitKey(0)

cv.destroyAllWindows()

```

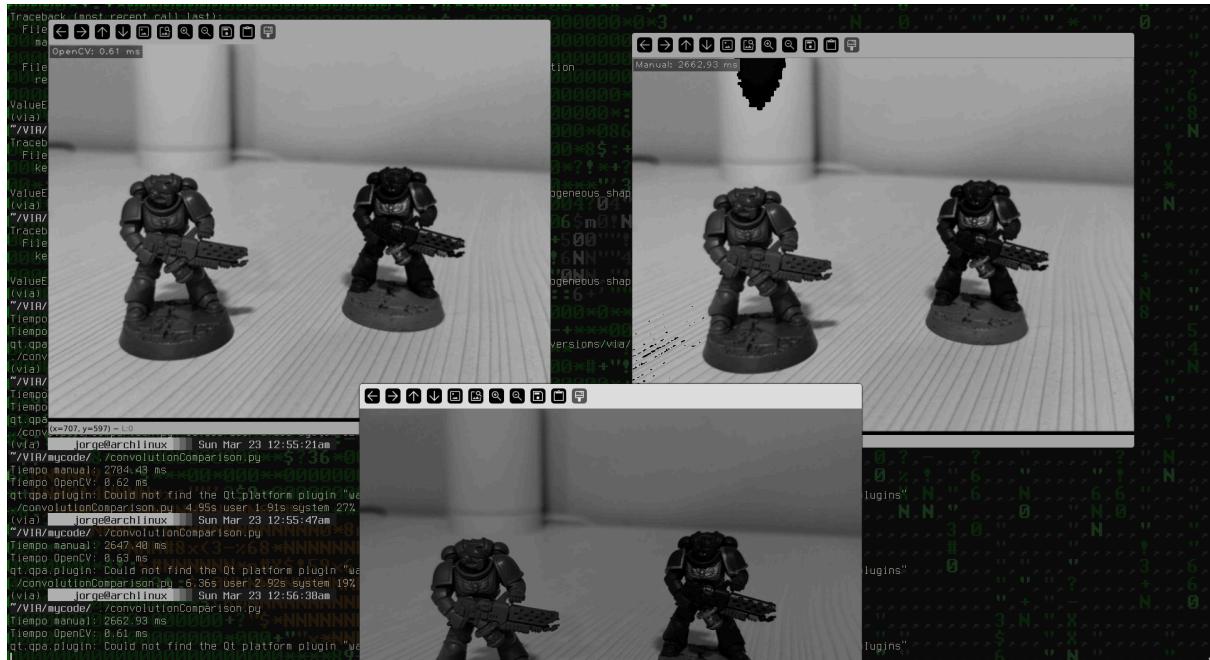


Imagen 11: Comparativa de los algoritmos de convolución manual y de OpenCV.

# Clasificador

- 1) Diseña la aplicación para que sea cómodo añadir nuevos métodos. Cada método debe implementarse en un módulo diferente con un interfaz común.

Para mí, lo más sensato es usar una aproximación modular, donde cada clasificador tiene su propia clase (fichero en python), ya que es la solución más escalable. Dicho esto, se ha generado una clase principal `modular_classifier.py` donde se carga el clasificador específico para todos los modelos disponibles en la carpeta `models/`. Los modelos implementados han sido `hist`, `orb` y `hu` (éstos dos últimos a sugerencia de ChatGPT).

- 2) Puedes permitir que se añadan modelos sobre la marcha y dar la opción de salvarlos.

Voy a aprovechar para dejar el código en este apartado, ya que está la función de añadir modelos:

```
#!/usr/bin/env python

import cv2 as cv
import argparse
import os
import importlib
from umucv.stream import autoStream

# Cargar clasificadores desde classifiers/
def load_classifier(name):
    try:
        return importlib.import_module(f'classifiers.{name}')
    except ModuleNotFoundError:
        print(f"[ERROR] Clasificador '{name}' no encontrado.")
        exit(1)

def load_model_features(models_dir, prepare_func):
    model_features = {}
    for file in os.listdir(models_dir):
        path = os.path.join(models_dir, file)
        img = cv.imread(path)
        if img is not None:
            model_features[file] = prepare_func(img)
    return model_features

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--models', required=True)
    parser.add_argument('--method', required=True)
    parser.add_argument('--dev', required=False)
    args = parser.parse_args()

    # Cargar clasificador
    clf = load_classifier(args.method)
    prepare, compare = clf.prepare, clf.compare
```

```

model_feats = load_model_features(args.models, prepare)
print(f"[INFO] Modelos cargados: {list(model_feats.keys())}")

cv.namedWindow("result")

for key, frame in autoStream():
    query_feat = prepare(frame)
    scores = {name: compare(query_feat, feat) for name, feat in
model_feats.items()}
    best = min(scores, key=scores.get)
    conf = scores[best]

    cv.putText(frame, f"Best match: {best} ({conf:.4f})", (10, 30),
               cv.FONT_HERSHEY_SIMPLEX, 0.8, (0, 255, 0), 2)

    for i, (name, score) in enumerate(sorted(scores.items(), key=lambda x:
x[1])[:3]):
        cv.putText(frame, f"{name}: {score:.4f}", (10, 60 + 25*i),
                   cv.FONT_HERSHEY_PLAIN, 1.2, (255, 255, 255), 1)

    if cv.waitKey(1) & 0xFF == ord('m'):
        import tkinter as tk
        from tkinter.simpledialog import askstring
        from tkinter import messagebox

        snapshot = frame.copy()

        root = tk.Tk()
        root.withdraw() # Ocultar ventana principal

        name = askstring("Nuevo modelo", "Nombre del nuevo modelo:")
        if name:
            filename = f"{name}.jpg"
            path = os.path.join(args.models, filename)
            if os.path.exists(path):
                overwrite = messagebox.askyesno("Sobrescribir",
f"Sobrescribir {filename} ?")
                if not overwrite:
                    continue

                cv.imwrite(path, snapshot)
                model_feats[filename] = prepare(snapshot)
                print(f"[INFO] Modelo '{filename}' añadido.")
            root.destroy()
            cv.imshow("result", frame)
            if cv.waitKey(1) & 0xFF in [27, ord('q')]:
                break

        cv.destroyAllWindows()

if __name__ == "__main__":
    main()

```

Como podemos ver, he usado el módulo tkinter para poder generar una ventana emergente para darle un nombre al modelo, ya que si lo hago con las herramientas de umucv (Ctrl+S), se guarda el scrot de nombre igual al momento de captura.

**3) Precomputa todo lo necesario para cada modelo, para que la comparación sea lo más rápida posible.**

La solución más sencilla para hacer esto es como se ve en el apartado anterior, en base a lo escrito en los argumentos, buscar la clase con el mismo nombre en la carpeta `classifiers/` y lanzar sus dos métodos, prepare y compare. En concreto, el que nos interesa para la precomputación es prepare. Aquí tenemos el ejemplo de la clase `orb.py`:

```
import cv2 as cv
import numpy as np

orb = cv.ORB_create()

def prepare(img):
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
    kps, des = orb.detectAndCompute(gray, None)
    return des if des is not None else np.array([])

def compare(desc1, desc2):
    if len(desc1) == 0 or len(desc2) == 0:
        return float("inf")
    matcher = cv.BFMatcher(cv.NORM_HAMMING, crossCheck=True)
    matches = matcher.match(desc1, desc2)
    return np.mean([m.distance for m in matches]) if matches else float("inf")
```

Donde vemos que realmente como preparación nuestra hay poca, simplemente pasar la imagen a gris y pasársela en este caso a la función de computación de ORB.

**4) Implementa un método basado en el "embedding" obtenido por [mediapipe \(code/DL/embedder\)](#).**

Usando como ejemplo este [cuaderno](#) de Google, aplicamos el mismo formato que hay para las clases de los otros métodos:

```
import numpy as np
import cv2 as cv
from mediapipe.tasks.python import vision
from mediapipe.tasks import python as mp_python
from mediapipe.tasks.python.vision import ImageEmbedder
import mediapipe as mp
import os

# Cargar el modelo .tflite solo una vez
MODEL_PATH = "/home/jorge/VIA/umucv/code/DL/mp_embedder/embedder.tflite"

if not os.path.exists(MODEL_PATH):
    raise FileNotFoundError(f"[ERROR] Modelo TFLite no encontrado: {MODEL_PATH}")
```

```

base_options = mp_python.BaseOptions(model_asset_path=MODEL_PATH)
options = vision.ImageEmbedderOptions(base_options=base_options,
12_normalize=True, quantize=False)
embedder = ImageEmbedder.create_from_options(options)

def prepare(img):
    img_rgb = cv.cvtColor(img, cv.COLOR_BGR2RGB)
    mp_img = mp.Image(image_format=mp.ImageFormat.SRGB, data=img_rgb)
    result = embedder.embed(mp_img)
    return np.array(result.embeddings[0].embedding)

def compare(e1, e2):
    return np.linalg.norm(e1 - e2) # distancia euclídea

```

He usado el embedder que hay disponible en el repositorio del profesor que he clonado en mi máquina. **la ruta MODEL\_PATH se debe cambiar para poder probar el programa, ya que estoy usando una ruta absoluta.**

A continuación voy a dejar algunos ejemplos de ejecución del programa `modular_classifier.py`:

Comenzamos con el método de histograma, que calcula el histograma de la imagen modelo en escala de grises, lo cual no es nada eficiente ya que perdemos información útil, como el color y la forma de la imagen, además de que es sensible a la luz, por lo que no es muy eficaz a la hora de dar una respuesta con gran nivel de confianza.

**Best match: persona.jpg (1.0032)**

persona.jpg: 1.0032  
dados\_ace.jpg: 1.2053  
meshtastic.jpg: 1.2065



Imagen 12: Clasificación de un objeto con el método de histogramas.

Ahora, pasamos al método [Hu Moments](#), que analiza formas. Aquí, podemos ver las fallas del modelo comparando, por ejemplo, una caja de dados con forma de prisma cuadrangular, con una pequeña navaja:



Imágenes 13 y 14: Clasificación de objetos con el método Hu Moments.

Como vemos, piensa que la caja de dados es una navaja, cuando el modelo está presente.

Ahora, vamos a pasar a [ORB](#). ORB funciona de manera parecida a SIFT, por lo que funciona muy bien. Está basado en dos métodos adicionales, FAST, que es un detector de esquinas rápido, y BRIEF, un descriptor binario eficiente. Si a esto le añadimos que está libre de patentes y es gratuito de usar, tenemos una herramienta muy potente. Vamos a ver su eficiencia:



Imágenes 15 y 16: Clasificación de objetos con el método ORB.



Imagen 17: Clasificación de un objeto con el método ORB.

Podemos apreciar que ORB no tiene ningún tipo de duda con los objetos que ha clasificado, ya que sus valores son significativamente menores a la siguiente mejor coincidencia.

Por último, tenemos el método mediapipe, que, como ya sabemos, es una librería que pertenece a Google. En concreto, vamos a usar su función de `image_embedder` para poder clasificar las imágenes pasadas haciendo cross-referencing con modelos ya precomputados. Al tratarse de un generador de vectores (algo así como hashing de una imagen), la comparativa es extremadamente rápida, ya que compara dos vectores entre sí y no tiene que ir comparando pixel a pixel, puntos clave como esquinas o directamente histogramas de color. Por otro lado, al tratarse de un modelo, es muy costoso generar uno por primera vez. Por suerte, el repositorio de la asignatura ya nos ofrece uno.

A continuación dejo unas imágenes para demostrar su capacidad de clasificación:



Imágenes 18 y 19: Clasificación de objetos con el método mediapipe.

Podemos ver que duda un poco más con la imagen del dispositivo [meshtastic](#), pero sigue clasificando las imágenes correctamente según los modelos dados.

**5) Implementa un reconocedor de gestos de manos basado, por ejemplo, en la distancia procrustes.**

Usando el script encontrado en [umucv/code/DL/mp\\_hands/hands.py](#) encontramos la lógica en la que basarnos para realizar este ejercicio. Añadiendo unos nuevos modelos en la subcarpeta `models/hands` podemos ver qué tal funciona.

```
#Script based on /code/DL/mp_hands/hands.py
import cv2 as cv
import numpy as np
import mediapipe as mp
from scipy.spatial import procrustes

mp_drawing = mp.solutions.drawing_utils
mp_drawing_styles = mp.solutions.drawing_styles
mp_hands = mp.solutions.hands

hands_detector = mp_hands.Hands(
    model_complexity=0,
    min_detection_confidence=0.5,
    min_tracking_confidence=0.5)

visualize = False

def set_visualization(enabled=True):
    global visualize
    visualize = enabled

def extract_landmarks(img):
    img_rgb = cv.cvtColor(img, cv.COLOR_BGR2RGB)
    results = hands_detector.process(img_rgb)
    if not results.multi_hand_landmarks:
        return None
    if visualize:
        for hand_landmarks in results.multi_hand_landmarks:
            mp_drawing.draw_landmarks(
                img,
                hand_landmarks,
                mp_hands.HAND_CONNECTIONS,
                mp_drawing_styles.get_default_hand_landmarks_style(),
                mp_drawing_styles.get_default_hand_connections_style())
    landmarks = results.multi_hand_landmarks[0].landmark
    return np.array([[lm.x, lm.y] for lm in landmarks])

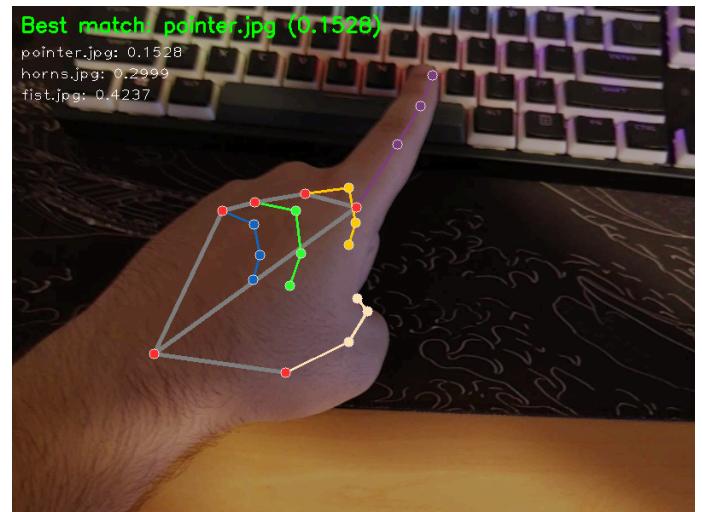
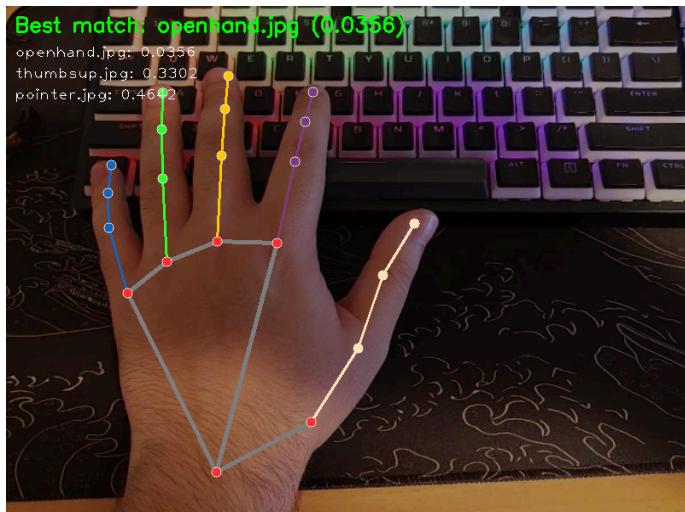
def normalize_landmarks(landmarks):
    if landmarks is None:
        return None
    return (landmarks - landmarks.mean(axis=0)) / landmarks.std(axis=0)

def prepare(img):
    landmarks = extract_landmarks(img)
    return normalize_landmarks(landmarks)
```

```

def compare(l1, l2):
    if l1 is None or l2 is None:
        return float('inf')
    try:
        mtx1, mtx2, disparity = procrustes(l1, l2)
        return disparity
    except Exception:
        return float('inf')

```



Imagenes 20 y 21: Clasificación de posturas de una mano con el método hands.

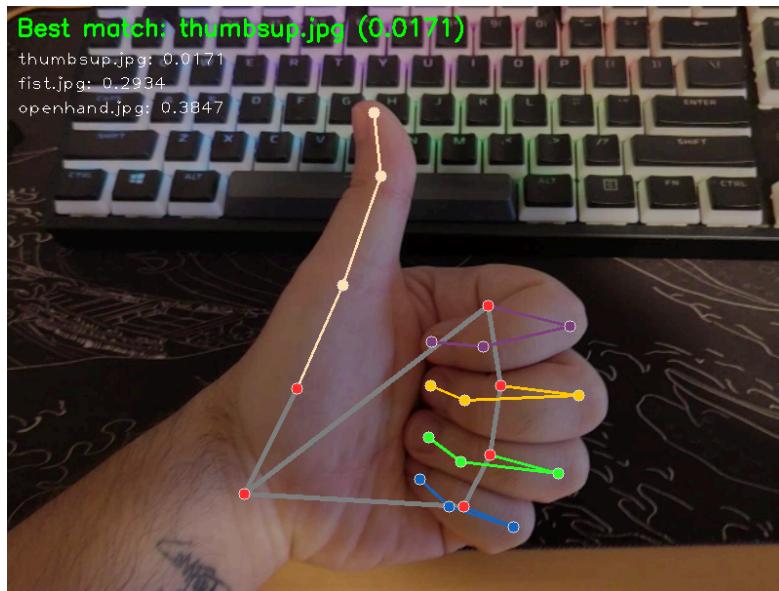


Imagen 22: Clasificación de posturas de una mano con el método hands.

Como podemos comprobar, el clasificador funciona correctamente con los modelos implementados (thumbsup, fist, openhand, pointer y horns). Cabe destacar la mención de procrustes, que se ha usado para devolver la similitud entre el modelo y el stream que se le pasa como feed.

Esto realmente se habría conseguido comparando uno con el otro, pero ya que nos ofrecen un índice de disparidad en la función `procrustes`, podemos pasársela directamente al stream para poner el ranking. Cuanto más se asemeja la imagen a un modelo, más cercano a cero será el valor mostrado por pantalla. Los resultados se ordenan de mayor a menor similitud (o menor a mayor diferencia).

## SIFT

SIFT es el método de visión artificial más interesante para mí personalmente, ya que creo que es el más factible para hacer un proyecto personal que tengo en mente sobre la digitalización de un juego de mesa para poder jugar de manera remota con un juez automático. El juego de mesa es concretamente [One Piece Card Game](#) (OPTCG), y la herramienta que planteo es muy parecida a una ya disponible para el famoso juego [Magic The Gathering](#), [Spelltable](#). La idea, por supuesto, es hacer una versión para OPTCG que sea gratuito y open source, integrando en él las reglas que tiene el juego de cartas.

Es por esto, que para probar el algoritmo de SIFT, he generado una nueva carpeta de modelos `models/OPTCG`, donde he dejado imágenes de cartas que tengo disponibles.

El script se ha realizado basándonos en el que hay disponible en [umucv/code/SIFT/sift.py](#).

```
#Script based on /code/SIFT/sift.py

import cv2 as cv

sift = cv.SIFT_create(nfeatures=1000)
matcher = cv.BFMatcher()

model_keypoints = {}
model_descriptors = {}
model_images = {}
show_matches = False

def prepare(img):
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
    kps, des = sift.detectAndCompute(gray, None)
    return (kps, des)

def compare(query_feat, model_feat):
    qkps, qdes = query_feat
    mkps, mdes = model_feat

    if qdes is None or mdes is None:
        return float("inf")
```

```

matches = matcher.knnMatch(qdes, mdes, k=2)
good = []
for m in matches:
    if len(m) >= 2:
        best, second = m
        if best.distance < 0.55 * second.distance:
            good.append(best)

return 1 / (len(good) + 1e-5)

#Draw thumbnail to better distinguish model
def draw_best_match(frame, best_name, model_path):
    thumb = cv.imread(model_path)
    if thumb is not None:
        thumb = cv.resize(thumb, (100, 100))
        h, w = frame.shape[:2]
        frame[0:100, w - 100:w] = thumb

def match_and_draw(frame, query_feat, model_feat):
    global show_matches

    if not show_matches:
        return frame

    qkps, qdes = query_feat
    mkps, mdes = model_feat

    if qdes is None or mdes is None:
        return frame

    matches = matcher.knnMatch(qdes, mdes, k=2)
    good = []
    for m in matches:
        if len(m) >= 2:
            best, second = m
            if best.distance < 0.55 * second.distance:
                good.append(best)

    imgm = cv.drawMatches(frame, qkps, model_images.get('current', frame),
    mkps, good,
        flags=0,
        matchColor=(128,255,128),
        singlePointColor=(128,128,128),
        outImg=None)

    return imgm

def set_visualization(flag):
    global show_matches
    show_matches = flag

```

Si ignoramos la parte de dibujar thumbnails y `match_and_draw`, el propio método compare es bastante sencillo, ya que crea una lista de matches “buenos”, y lo transforma en una medida de distancia inversa, es decir, cuantas más coincidencias, menor será ese número.

Aunque genera muchísimo lag, es bastante efectivo detectando las cartas mencionadas anteriormente, incluso usando cartas con arte alternativo, que son diseños muy parecidos a los originales con pequeñas diferencias de estilo y color (que no importa en nuestro caso):



Imágenes 23 y 24: Clasificación de dos cartas del mismo personaje con el método SIFT.



Imágenes 25 y 26: Clasificación de diferentes cartas con el método SIFT.

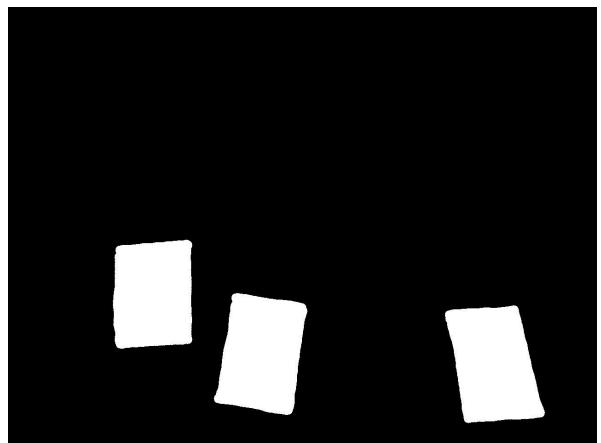
# DL

- 1) Entrena un modelo de deep learning con tus propias imágenes. Puedes utilizar la herramienta [ultralytics](#) y apoyarte en el ejemplo [code/DL/yolotrain](#). Otra posibilidad es reproducir el ejemplo [code/DL/UNET](#).

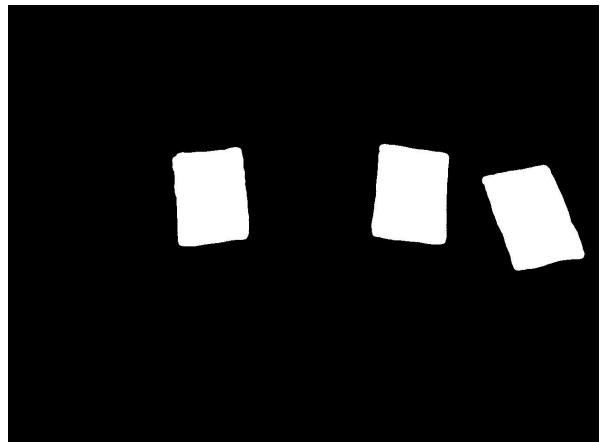
En mi caso, la opción más sencilla de seguir y la que menos recursos va a consumir en mi ordenador, que ya tiene unos años, es el ejemplo de UNET, por lo que vamos a comenzar seleccionando imágenes propias de lo que queramos.

Copiamos los ficheros necesarios de [code/DL/UNET](#), `prepare.py`, `train.py`, `myUNET.py`, `check.py` y `rununet.py`.

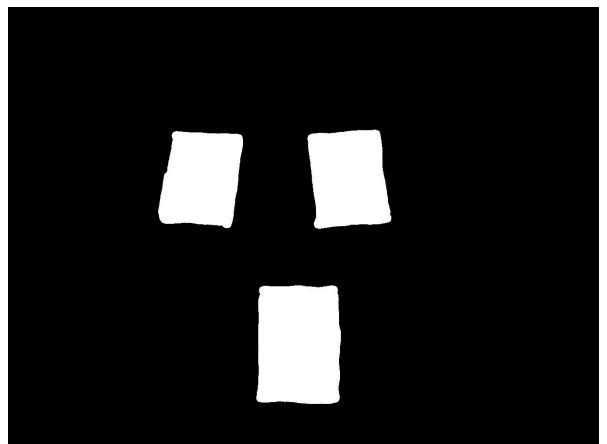
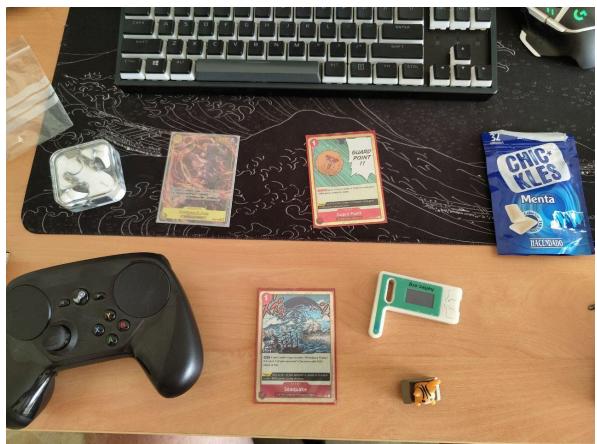
Lo primero que tenemos que hacer es tomar imágenes modelo. Vamos a seguir la misma dinámica que con SIFT, por lo que voy a tomar imágenes de cartas de OPTCG con otros objetos y generar máscaras válidas para las mismas.



Imágenes 27 y 28: Imagen de entrenamiento 1 junto con su máscara.

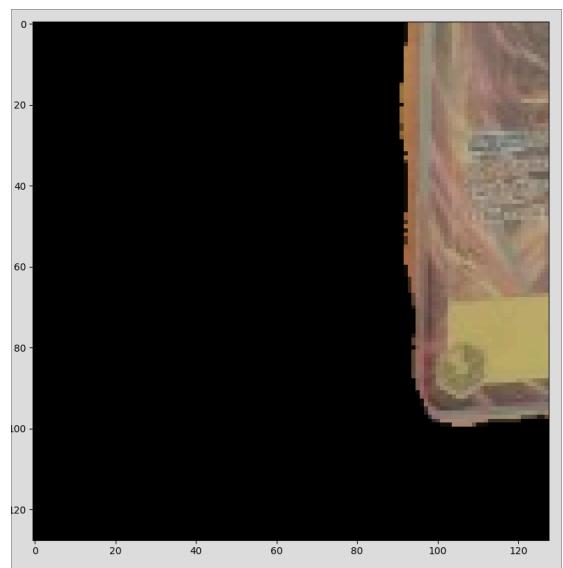


Imágenes 29 y 30: Imagen de entrenamiento 2 junto con su máscara.



Imágenes 31 y 32: Imagen de entrenamiento 3 junto con su máscara.

Lanzando el script `prepare.py` obtenemos algunos recortes y la máscara puesta sobre la imagen:



Imágenes 33 y 34: Ejemplos de ejecución de `prepare.py`.

Después, pasamos al propio entrenamiento del modelo de Deep Learning, con el script `train.py`, que tarda bastante:

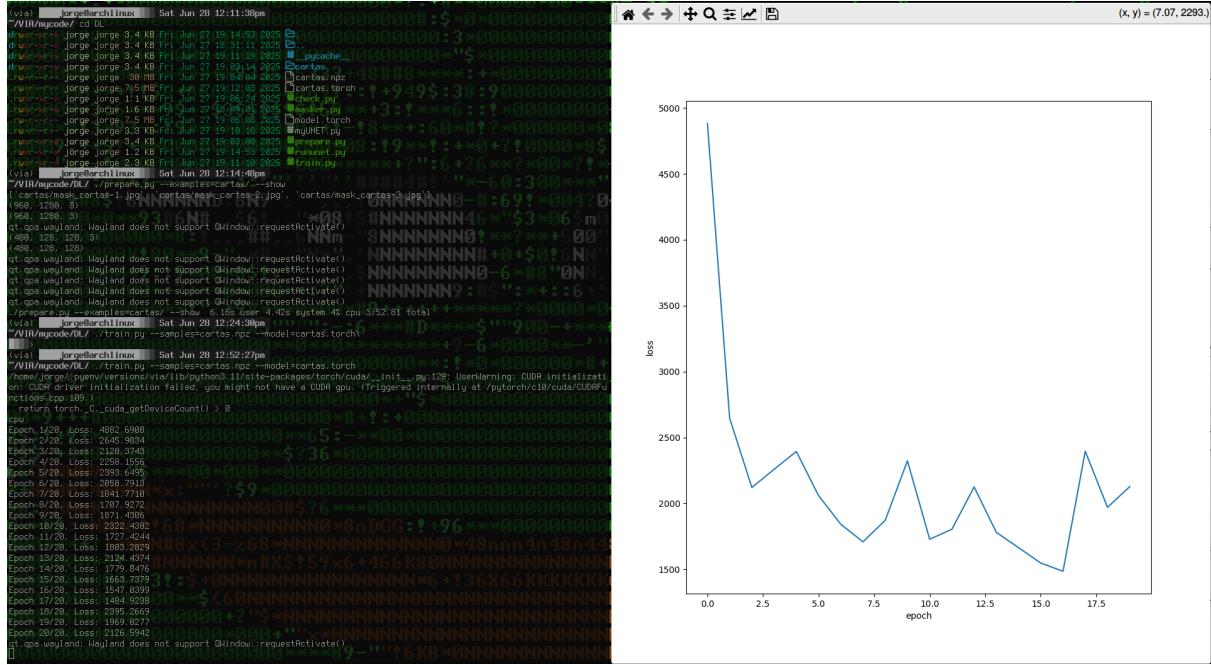


Imagen 35: Resultado de la ejecución del entrenamiento.

Ahora, tras haberle dado unas tres o cuatro pasadas, he ejecutado el fichero `check.py` (voy a omitir la captura de pantalla) y he comprobado que funciona suficientemente bien.

Ahora, ejecutamos `rununet.py` pasándole como stream nuestra fuente de vídeo, y generamos un escenario con un montón de objetos, entre los cuales hay cartas como las que hemos usado para el entrenamiento:

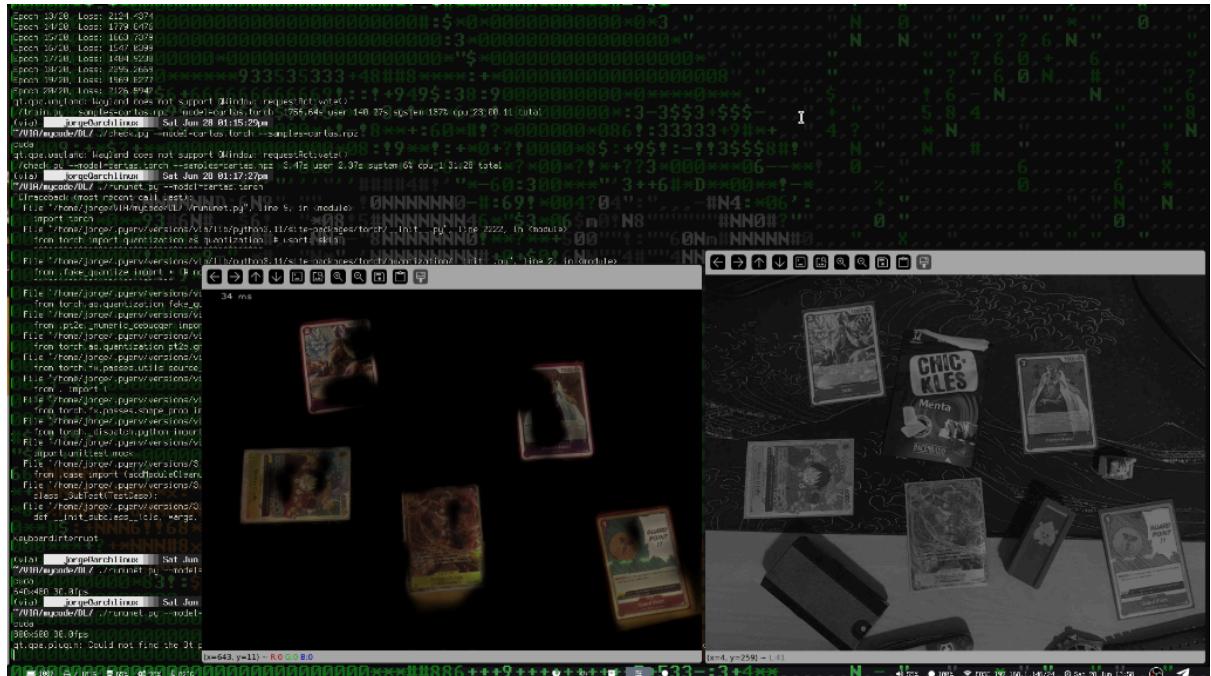


Imagen 36: Stream de vídeo y la máscara aplicada con el entrenamiento.

Como podemos observar, no funciona excesivamente bien, pero es capaz de detectar cartas que no se han usado para entrenar el modelo relativamente bien (aunque viendo las cartas de la parte de abajo, indudablemente funciona mejor con estas).

## Rectificación

- 1) Deshaz la deformación de perspectiva de la imagen de un plano para medir distancias (tomando manualmente referencias conocidas). Por ejemplo, mide la distancia entre las monedas en `coins.png` o la distancia a la que se realiza el disparo en `gol-eder.png`. Las coordenadas reales de los puntos de referencia y sus posiciones en la imagen deben pasarse como parámetro en un archivo de texto. Aunque puedes mostrar la imagen rectificada para comprobar las operaciones, debes marcar los puntos y mostrar el resultado sobre la imagen original. Verifica los resultados con imágenes originales tomadas por ti.

A la hora de hacer este ejercicio he hecho dos ficheros .py. El primero lo he utilizado casi exclusivamente para medir la distancia usando la regla de `coins.py` para aclararme. Después he copiado el código y lo he modificado para que acepte los puntos escritos en un fichero de texto.

Buscando en los [notebooks](#) de la asignatura y por [internet](#) (y con ayuda de ChatGPT para boilerplates) he acabado con el script `_rectify_plane.py`.



Imagen 37: Ejemplo de ejecución de `_rectify_plane.py`.

Como podemos observar, es bastante bueno midiendo una distancia en la regla de la imagen, pero este script no nos sirve ya que hemos hecho trampas; tenemos las medidas del DNI hardcodeadas y simplemente le indicamos dónde está.

Para la versión real de `rectify_plane.py` lo hemos hecho como tiene que ser: un fichero de texto con las medidas en la imagen junto con su medida real. Al principio no he comprendido muy bien el enunciado, pero gracias a ChatGPT he entendido que simplemente se ponen las medidas conocidas reales de un objeto y su reflejo en la imagen en pixels. Para ello, he cogido el script `/code/util/medidor.py` y lo he modificado para que me de las coordenadas de cada punto cuando pincho en la imagen. Con esta herramienta, he apuntado las medidas de 3 ficheros, `medidas_rectify_3`, `medidas_rectify_4`, `medidas_rectify_5` y `medidas_rectify_6`. Estos ficheros van emparejados con sus contrapartes en imagen `rectify_3.jpg`, `rectify_4.jpg`, `rectify_5.jpg` y `rectify_6.jpg`. En cuanto a las modificaciones del fichero, he creado las variables `points_img` y `points_real`, que son una lista de tuplas que contienen los pares de coordenadas. El resto funciona prácticamente igual, solo que ahora se generan dos ventanas, la imagen original pasada por parámetro y la imagen rectificada. Ahora las distancias se marcan sobre la imagen real. El fichero es demasiado grande como para copiar el código en el fichero, además de que todo el bulto ha sido para mostrar la imagen rectificada, así que solo voy a explicar algunos snippets interesantes:

```
# Cargar imagen
img = cv.imread(sys.argv[1])
if img is None:
    print(f"Error: no se pudo cargar la imagen {sys.argv[1]}")
    sys.exit(1)

# Cargar puntos de referencia desde el archivo
ref_file = sys.argv[2]
with open(ref_file) as f:
    lines = [line.strip().split() for line in f if line.strip() and not
line.startswith("#")]
    if len(lines) != 4:
        print("Error: el archivo de referencia debe contener exactamente 4
puntos")
        sys.exit(1)
    points_img = [tuple(map(float, l[:2])) for l in lines]
    points_real = [tuple(map(float, l[2:])) for l in lines]

# Convertir puntos a np.array
src = np.array(points_img, dtype=np.float32)
dst = np.array(points_real, dtype=np.float32)

# Escala de 100px por cm
SCALE = 100
dst_scaled = dst * SCALE

# Calcular homografia
H_mat, _ = cv.findHomography(src, dst_scaled)
```

Este primer snippet muestra cómo se cargan los puntos de interés desde un fichero de texto, que tiene un formato de tipo `{coord_x, coord_y, dist_real_x, dist_real_y}`. También cómo cargamos la imagen a partir del parámetro, pero eso no nos importa. Lo interesante es convertir los puntos a un array compatible con numpy (para poder aplicar después una traslación para asegurar que la imagen rectificada se vea completa) y le aplicamos una escala de 100 pixels por centímetro. Esto luego nos servirá a la hora de hacer los cálculos sobre la imagen, ya que tendremos que dividirlo entre el mismo valor para deshacer la escala.

```
cv.line(orig_display, clicks[0], clicks[1], (255, 0, 255), 2)
p1 = np.array([[clicks[0]]], dtype=np.float32)
p2 = np.array([[clicks[1]]], dtype=np.float32)
p1_real = cv.perspectiveTransform(p1, H_mat)[0][0]
p2_real = cv.perspectiveTransform(p2, H_mat)[0][0]
dist = np.linalg.norm(p2_real - p1_real) / SCALE # distancia original
# dividido entre la escala, 100px por cm
c = tuple(np.mean(clicks, axis=0).astype(int))
cv.putText(orig_display, f"{dist:.2f} cm", c, cv.FONT_HERSHEY_SIMPLEX, 1.3,
(255, 0, 255), 3)
```

Este otro snippet muestra cómo se calcula la distancia real entre dos puntos, usando `cv.perspectiveTransform` pasándole la matriz que he comentado de la traslación. Después calculamos la diferencia y lo dividimos entre nuestra escala y deberíamos tener nuestra medida real calculada con cierto grado de precisión.

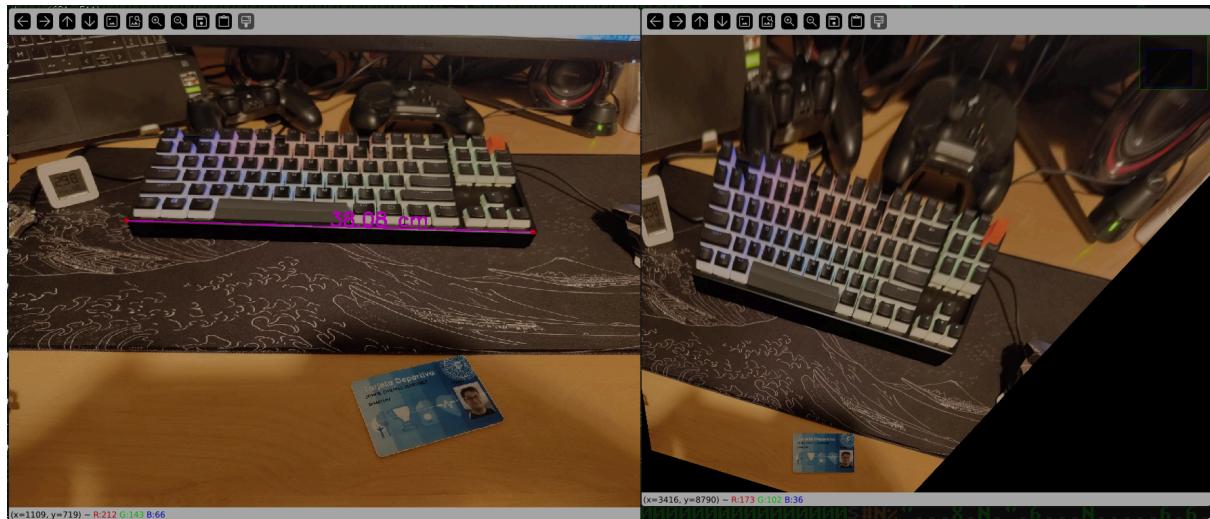


Imagen 38: Ejemplo de ejecución de `rectify_plane.py` con la imagen `rectify_3.jpg`.

En este caso, la distancia que he marcado es el largo de mi teclado, un Keychron K8, que nos da 38.08cm. La referencia ha sido mi viejo carnet de la Universidad de Almería con una medida estándar de carnet.



Imagen 39: Medida real keychron.

En este caso, el teclado mide más o menos 36.3cm, por lo que la medida se ha ido un poco, pero sigue siendo un porcentaje de error de un  $\approx 5\%$ .

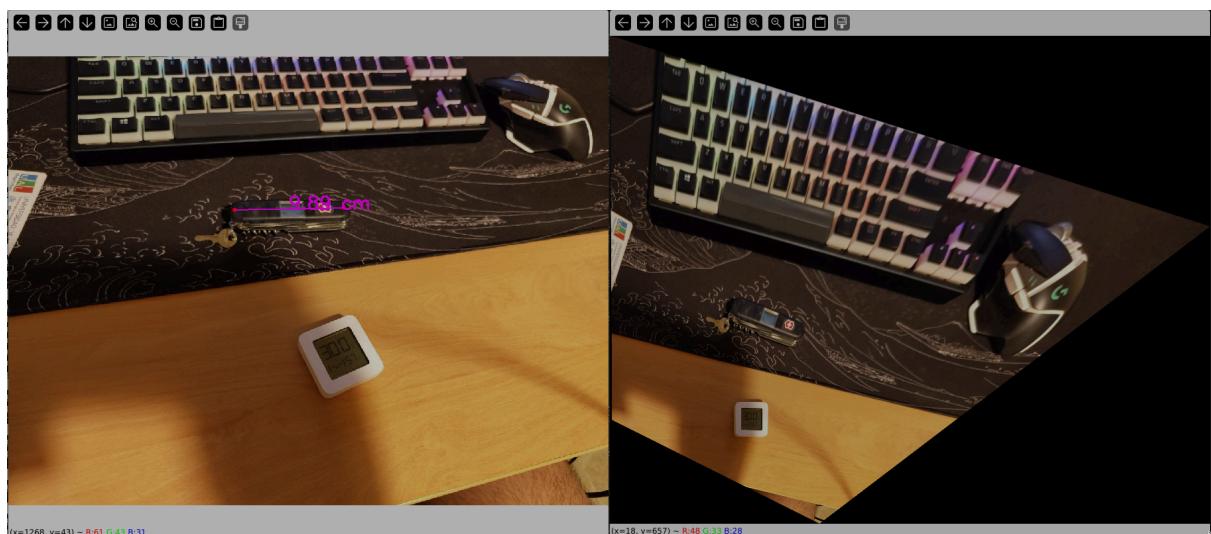


Imagen 40: Ejemplo de ejecución de `rectify_plane.py` con la imagen `rectify_4.jpg`.

Para esta imagen, la referencia ha sido un termómetro Zigbee con una medida de 4x4cm. La medida marcada en este caso ha sido una navaja multiusos Victorinox, que ha dado 9.89cm.



Imagen 41: Medida real navaja.

Como se puede observar, la navaja mide poco más de 9cm, por lo que tenemos un porcentaje de error de  $\approx 9.9\%$ .

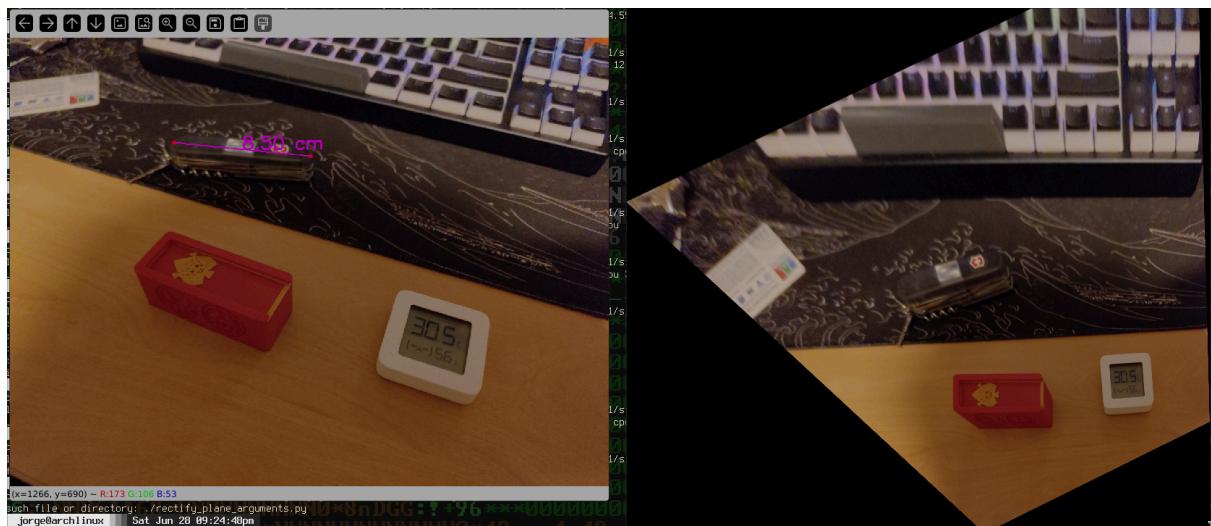


Imagen 42: Ejemplo de ejecución de `rectify_plane.py` con la imagen `rectify_5.jpg`.

Para esta imagen, la referencia ha sido una caja de dados impresa en 3D con una medida de  $6 \times 2.8\text{cm}$ . La medida marcada en este caso ha sido la misma navaja, que ha dado  $8.5\text{cm}$ . Como ya sabemos, la navaja mide unos  $9\text{cm}$ , así que el porcentaje de error ha sido de  $\approx 5.6\%$ .

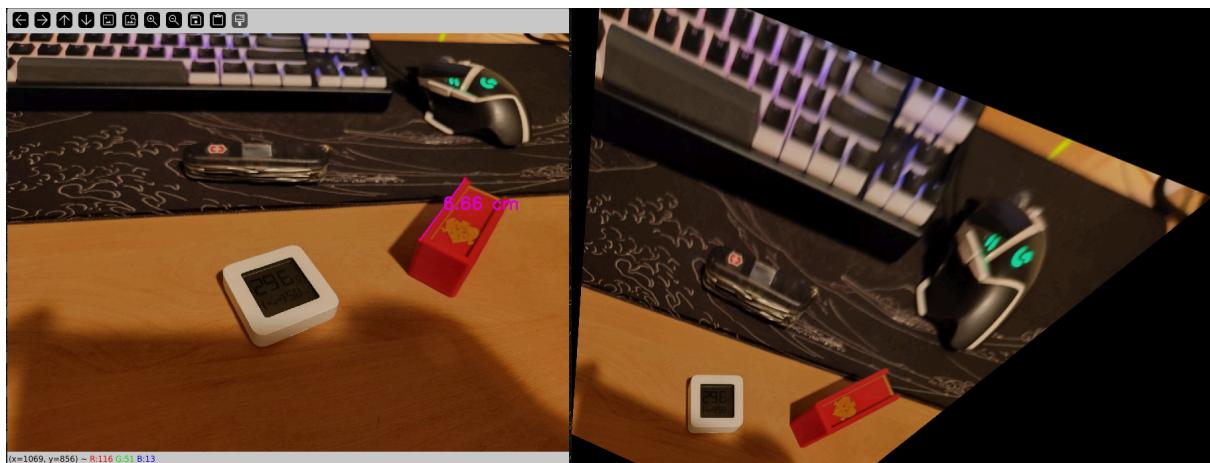


Imagen 43: Ejemplo de ejecución de rectify\_plane.py con la imagen rectify\_6.jpg.

Esta imagen vuelve a tener el Zigbee como referencia, y se ha medido la caja de datos, que ha dado 6.66cm curiosamente.

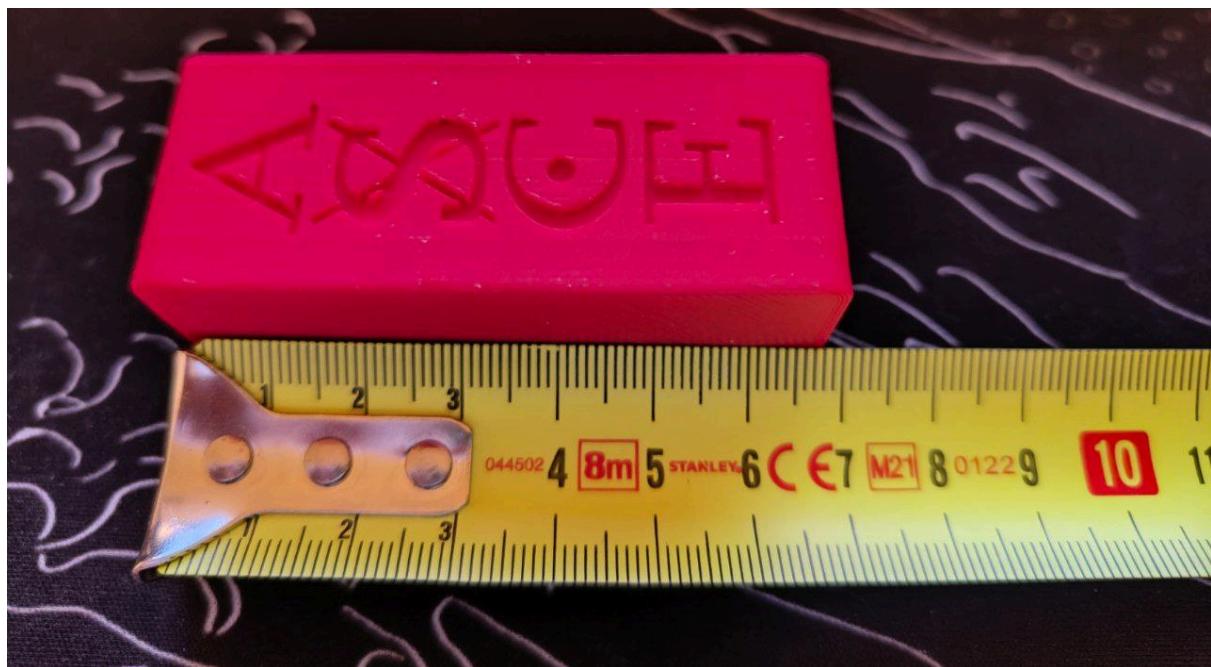


Imagen 44: Medida real datos.

Como vemos en la imagen, la caja de datos mide 6.8cm, así que el porcentaje de error es de  $\approx 2\%$ , que está genial.