Comentários finais:

Ruído Gaussiano

O ruído Gaussiano é caracterizado por uma variação suave dos valores de intensidade ao redor dos pixels, sem criar pontos isolados muito discrepantes.

Filtro Gaussiano aplicado:

- Funciona muito bem nesse caso, pois o filtro Gaussiano é essencialmente uma suavização ponderada.
- Ele reduz a intensidade das variações randômicas sem distorcer muito os contornos.
- Com um ksize maior, observa-se maior suavização, mas também maior borramento da imagem (perda de detalhes finos).

Filtro Mediano aplicado:

- Também consegue reduzir o ruído Gaussiano, mas não é o mais adequado.
- Como o ruído afeta quase todos os pixels, a mediana não consegue eliminar o ruído de forma tão eficiente quanto o Gaussiano.
- Resultado: redução parcial do ruído, mas com artefatos tipo "posterização" em áreas com gradientes suaves.

Combinação Gaussiano + Mediana:

- Pode suavizar ainda mais, mas tende a perder nitidez.
- Funciona melhor se quisermos reduzir ruído + remover pequenos outliers.

Ruído Sal-e-Pimenta

O ruído sal-e-pimenta altera pixels de forma extrema (ficam pretos ou brancos isolados).

Filtro Gaussiano aplicado:

- Não é eficiente, pois o filtro Gaussiano faz média ponderada, e os pixels muito diferentes (0 ou 255) "contaminam" a vizinhança.
- Resultado: borramento dos pontos brancos e pretos, mas sem removê-los totalmente.

Filtro Mediano aplicado:

- Muito mais eficaz nesse tipo de ruído.
- O pixel com valor extremo (0 ou 255) é substituído pela mediana da vizinhança, que geralmente representa melhor os pixels ao redor.
- Resultado: remoção quase total dos pontos isolados, preservando as bordas da imagem.

Combinação Gaussiano + Mediana:

- Nesse caso, o filtro Mediano já resolve grande parte do problema.
- Aplicar Gaussiano depois pode apenas suavizar um pouco mais, mas também reduzir bordas.

Comparação pelo tamanho da máscara (3x3, 5x5)

Máscaras pequenas (3x3):

- Preservam mais detalhes finos, porém podem não remover totalmente o ruído intenso.
- Bom equilíbrio entre nitidez e redução de ruído leve.

Máscaras grandes (5x5):

- Removem mais ruído, mas aumentam o borramento.
- Bordas ficam menos nítidas.
- No caso de sal-e-pimenta, quanto maior a janela, mais eficaz a filtragem, mas sempre à custa da perda de detalhes.

Comparação implementação vs cv2

as imagens parecem muito semelhantes, mostrando que a implementação foi feita corretamente.

Conclusão:

- Para ruído Gaussiano, o Filtro Gaussiano é mais adequado.
- Para ruído Sal-e-Pimenta, o Filtro Mediano é o mais eficaz.
- A escolha do tamanho da máscara depende do balanço desejado entre remoção de ruído e preservação de detalhes:
  - 3x3 ⟶ bom para preservar detalhes, mas remove pouco ruído.
  - 5x5 ou maior ⟶ remove melhor o ruído, mas borra a imagem.
- A combinação de filtros pode ser útil, mas geralmente aplica-se cada filtro de acordo com o tipo de ruído predominante.

---

c)

Use uma rotina qualquer para o algoritmo CANNY e produza diversas imagens indicando os efeitos de escala (uma escala menor se refere a uma imagem vista de perto ou zoom em uma determinada área) e os limiares de contraste nos contornos detectados. Implemente os detectores de Robert e Sobel nas mesmas imagens, para uma única escala, e compare os três algoritmos. Mostre as imagens originais. Sugestão: Use a função icanny fornecida pela Machine Vison ToolBox, que pode ser baixada do site www.petercorke.com. Podem ser usadas quaisquer funções de outras bibliotecas que correspondam ao algoritmo Canny. Comente os resultados.

```
import cv2
import numpy as np
from PIL import Image
from skimage.transform import resize

def linear_filter(I: np.ndarray, A: np.ndarray) -> np.ndarray:
    """
```

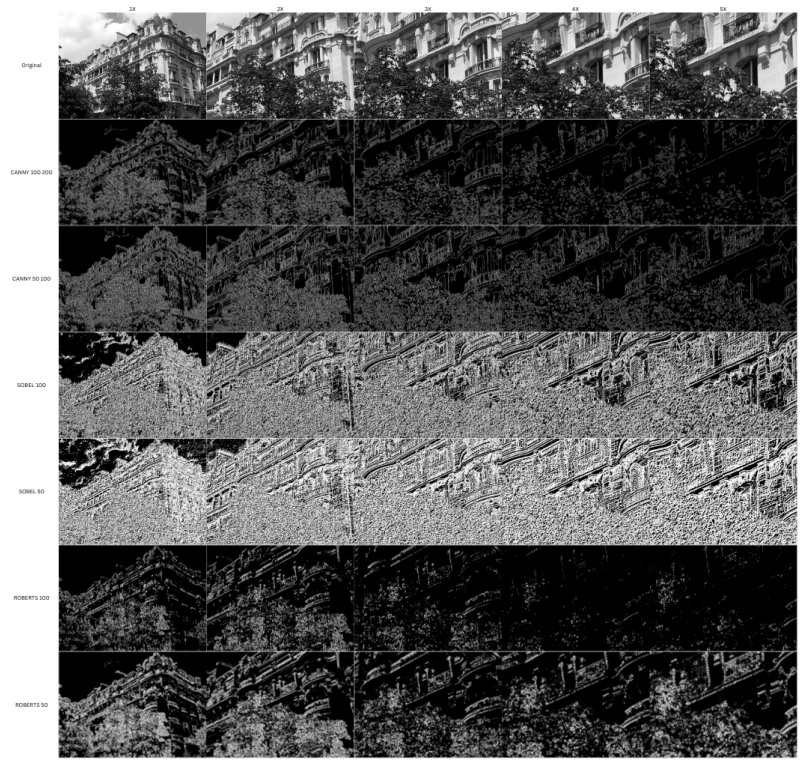
```
    Implementação manual de filtro linear 2D por convolução.
    I: imagem (numpy array 2D)
    A: máscara do filtro (numpy array m x m, pode ser par ou ímpar)
    """
    m = A.shape[0]  # tamanho do kernel (supomos quadrado)
    N, M = I.shape
    IA = np.zeros_like(I, dtype=np.float64)

    if m % 2 == 1:
        # Caso ímpar: centro está bem definido
        offset = m // 2
        i_start, i_end = offset, N - offset
        j_start, j_end = offset, M - offset

        for i in range(i_start, i_end):
            for j in range(j_start, j_end):
                region = I[i - offset:i + offset + 1, j - offset:j + offset + 1]
                IA[i, j] = np.sum(A * region)

    else:
        # Caso par: usar offset "meio a meio"
        offset = m // 2
        i_start, i_end = offset - 1, N - offset
        j_start, j_end = offset - 1, M - offset

        for i in range(i_start, i_end):
            for j in range(j_start, j_end):
                region = I[i - offset + 1:i + offset + 1,
                           j - offset + 1:j + offset + 1]
                IA[i, j] = np.sum(A * region)

    # Normalizar para [0,255] se imagem
    IA = np.clip(IA, 0, 255)
    return IA.astype(np.uint8)

def roberts_step_1(img: np.ndarray):
    # filtro feijão com arroz
    img_s = cv2.GaussianBlur(img, (5, 5), 1)
    return img_s


def roberts_step_2(img_s: np.ndarray):
    # mascaras
    kx = np.array([
        [1, -1],
        [-1, 1]
    ], dtype=np.float32)
    ky = np.array([
        [-1, 1],
        [1, -1]
    ], dtype=np.float32)
    gx = linear_filter(img_s, kx)
    gy = linear_filter(img_s, ky)
    return gx, gy


def sobel_step_2(img_s: np.ndarray):
    # mascaras
    kx = np.array([
```

```
        [-1, -2, -1],
        [0, 0, 0],
        [1, 2, 1]
    ], dtype=np.float32)
    ky = np.array([
        [-1, 0, 1],
        [-2, 0, 2],
        [-1, 0, 1]
    ], dtype=np.float32)
    gx = linear_filter(img_s, kx)
    gy = linear_filter(img_s, ky)
    return gx, gy


def roberts_step_3(gx, gy):
    # Magnitude
    g = np.sqrt(gx**2 + gy**2)

    # Normalizar para 0-255
    g = (g / g.max()) * 255
    g = g.astype(np.uint8)
    return g


def roberts_step_4(g, tau):
    edges = np.zeros_like(g, dtype=np.uint8)
    edges[g > tau] = 255
    return edges


def roberts_edge_det(img: np.ndarray, tau: float) -> np.ndarray:
    img_s = roberts_step_1(img)

    gx, gy = roberts_step_2(img_s)

    g = roberts_step_3(gx, gy)

    return roberts_step_4(g, tau)


def sobel_edge_det(img: np.ndarray, tau: float) -> np.ndarray:
    img_s = roberts_step_1(img)

    gx, gy = sobel_step_2(img_s)

    g = roberts_step_3(gx, gy)

    return roberts_step_4(g, tau)

def image_to_array(file, width=None, height=None):
    if file.lower().endswith(".npz"):
        # Carregar npz
        data = np.load(file)
        # se tiver a chave "resultado", usa ela. Se não, pega a primeira existent
        arr = data["resultado"] if "resultado" in data else data[list(data.keys()

        # garantir dtype float64
        np_img = np.array(arr, dtype=np.float64)
```

```
        if width is not None and height is not None:
            # Se precisar forçar resize (padrão igual ao das imagens PIL)
            if (np_img.shape[1], np_img.shape[0]) != (width, height):
                np_img = resize(
                    np_img, (height, width, np_img.shape[2]), preserve_range=True
                ).astype(np.float64)

    else:
        # Imagem regular
        img = Image.open(file).convert("RGB")
        if width is not None and height is not None:
            img = img.resize((width, height))
        np_img = np.array(img, dtype=np.float64)

    return np_img


def save_image(output, name, img_type):
    name = f"{name}.{img_type}"
    if img_type.lower() == "npz":
        np.savez_compressed(name, resultado=output)
    else:
        output = np.clip(output, 0, 255).astype(np.uint8)
        # Salva a imagem média
        if len(output.shape) == 2:
            img_final = Image.fromarray(output, mode="L")
        else:
            img_final = Image.fromarray(output, mode="RGB")
        img_final.save(name)
    print(f"Imagem salva como {name}")

def generate_scales(path, scales):
    img = Image.open(path)
    W, H = img.size
    images = []

    for scale in scales:
        if scale < 1:
            # Zoom out → reduz resolução (igual você já fazia)
            new_size = (int(W * scale), int(H * scale))
            scaled_img = img.resize(new_size, Image.Resampling.LANCZOS)

        else:
            # Zoom in → crop central e depois resize de volta
            crop_w, crop_h = int(W / scale), int(H / scale)
            left = (W - crop_w) // 2
            top = (H - crop_h) // 2
            right = left + crop_w
            bottom = top + crop_h

            cropped = img.crop((left, top, right, bottom))
            scaled_img = cropped.resize((W, H), Image.Resampling.LANCZOS)

        save_name = f"{path}_{str(scale).replace('.', '_')}x.jpg"
        scaled_img.save(save_name)
        images.append(save_name)

        print(f"Salvo: {save_name} ({scaled_img.width}x{scaled_img.height} px)")
```

```
    return images

# Diferentes escalas para gerar
scales = [1, 2, 3, 4, 5]  # 100%, 200%, 300%, 400%, 500%

paths: list[str] = generate_scales("building2-1.png", scales)

for path in paths:
    # 1. Carregar imagem (em RGB e depois converter para grayscale)
    arr = image_to_array(path)
    # arr = image_to_array("average.png")
    img_gray = cv2.cvtColor(arr.astype(np.uint8), cv2.COLOR_RGB2GRAY)

    # 2. Aplicar detectores
    edges_roberts_50 = roberts_edge_det(img_gray, tau=50)
    edges_sobel_50 = sobel_edge_det(img_gray, tau=50)

    edges_roberts_100 = roberts_edge_det(img_gray, tau=100)
    edges_sobel_100 = sobel_edge_det(img_gray, tau=100)

    # Canny com dois thresholds
    edges_canny_1 = cv2.Canny(img_gray, 50, 100)
    edges_canny_2 = cv2.Canny(img_gray, 100, 200)

    path = path.replace('.png', '').replace('.jpg', '')
    # 3. Salvar resultados
    save_image(img_gray, f"{path}_gray", "png")
    save_image(edges_roberts_50, f"{path}_edges_roberts_50", "png")
    save_image(edges_sobel_50, f"{path}_edges_sobel_50", "png")
    save_image(edges_roberts_100, f"{path}_edges_roberts_100", "png")
    save_image(edges_sobel_100, f"{path}_edges_sobel_100", "png")
    save_image(edges_canny_1, f"{path}_edges_canny_50_100", "png")
    save_image(edges_canny_2, f"{path}_edges_canny_100_200", "png")
```

Resultados:



Comentários finais:

Imagens Originais e Escalas

- A geração de escalas permitiu visualizar os efeitos de zoom-in (escala > 1) e zoom-out (escala < 1, se fosse incluída).
- Em escala maior (zoom-in), os detalhes ficam mais evidentes, mas o número de pixels aumenta e pequenas variações de intensidade ficam mais perceptíveis para os detectores. Isso tende a gerar mais bordas (inclusive ruído).
- Em escala menor (zoom-out, menos pixels), as bordas ficam mais "suaves" e várias bordas finas podem desaparecer, pois se perdem ao reduzir a resolução.

Detector de Roberts

- O Roberts utiliza máscaras pequenas (2×2), logo é muito sensível a ruídos e pequenas variações na intensidade.
- Em limites baixos de $\tau = 50$, aparece bastante ruído, especialmente em superfícies com pequenas texturas.
- Com $\tau = 100$, parte do ruído é removida, mas também perde-se contorno de detalhes finos (bordas de janelas, detalhes arquitetônicos).
- Ele tende a gerar bordas mais fragmentadas e menos contínuas.

Detector de Sobel

- O Sobel utiliza máscaras maiores (3×3), suavizando mais a imagem e respondendo melhor a gradientes.

- Em τ = 50, detecta muitas bordas, mas mantendo maior continuidade do que Roberts.
- Em τ = 100, ainda preserva os contornos principais (ex.: borda de prédios, telhado), mas elimina detalhes menores.
- Comparado a Roberts, o Sobel reduz ruído e produz bordas mais fortes e "limpas", embora possa engordar um pouco as linhas.

Detector de Canny

- O Canny é mais sofisticado: aplica suavização Gaussiana + gradiente + supressão de não-máximos + histerese (usa dois limiares).
- Para limiares (50, 100):
  - Detecta bordas muito mais finas e definidas.
  - Mantém continuidade mesmo em contornos mais longos (linhas de paredes, telhado).
- Para limiares (100, 200):
  - As bordas ficam ainda mais limpas e restritas apenas às mais fortes, praticamente eliminando ruído.
  - Entretanto, alguns detalhes secundários da cena somem, restando somente os contornos estruturais principais do prédio.

Comparação Geral

- Roberts ⟶ Rápido e simples, mas muito sensível a ruídos, gera bordas fragmentadas.
- Sobel ⟶ Um bom equilíbrio: mais robusto, gera bordas fortes e contínuas, adequado quando não se precisa de sofisticação.
- Canny ⟶ Melhor qualidade entre os três: bordas finas, contínuas e bem localizadas, com tratamento extra de ruído. É o que mais se aproxima de uma extração limpa de contornos estruturais.

Conclusão:

- Escala: quanto maior o zoom, mais bordas e ruídos aparecem; quanto menor, mais se perdem detalhes.
- Limiar: limiares baixos capturam muitos detalhes (inclusive ruído), enquanto limiares altos extraem apenas bordas fortes.
- Comparação: Canny > Sobel > Roberts em termos de robustez e precisão.

d)

Implemente o algoritmo CORNERS (mostrado nos slides) e use uma interface que mostre os cantos superpostos às imagens originais. Construa uma imagem sintética com um quadrado branco sobre um fundo preto e teste o algoritmo. Compare os resultados com o algoritmo de Harris, nas mesmas imagens. Em seguida use a imagem disponível no site do curso (building2-1.png). Rode a

função icorner no Matlab, fornecida pela Machine Vison ToolBox que pode ser baixada do site www.petercorke.com (algoritmo de detector de cantos de Harris) ou qualquer outra função que realize operação semelhante, sobre a mesma imagem (o limiar tem de ter escolha adequada para fazer comparação com os outros algoritmos). Faça o mesmo para os métodos SURF (pode ser o ORB que é de livre instalação. O SURF pode ter restrições na versão free) e SIFT na plataforma que você estiver utilizando. Discuta os resultados.

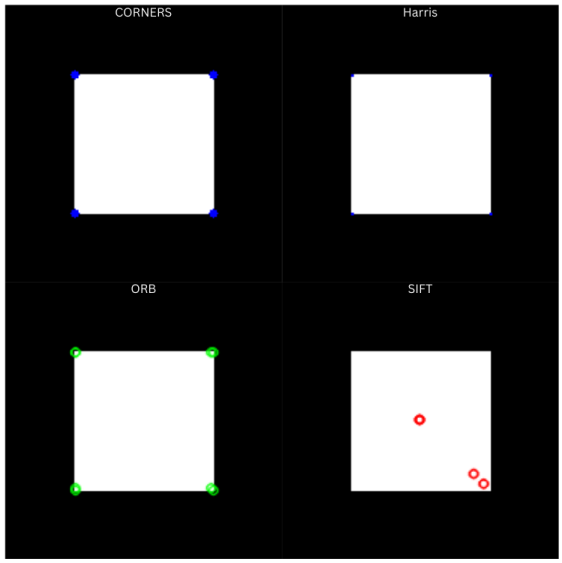
```
import cv2
import numpy as np
from PIL import Image
from skimage.transform import resize

def image_to_array(file, width=None, height=None):
    if file.lower().endswith(".npz"):
        # Carregar npz
        data = np.load(file)
        # se tiver a chave "resultado", usa ela. Se não, pega a primeira existent
        arr = data["resultado"] if "resultado" in data else data[list(data.keys()

        # garantir dtype float64
        np_img = np.array(arr, dtype=np.float64)

        if width is not None and height is not None:
            # Se precisar forçar resize (padrão igual ao das imagens PIL)
            if (np_img.shape[1], np_img.shape[0]) != (width, height):
                np_img = resize(
                    np_img, (height, width, np_img.shape[2]), preserve_range=True
                ).astype(np.float64)

    else:
        # Imagem regular
        img = Image.open(file).convert("RGB")
        if width is not None and height is not None:
            img = img.resize((width, height))
        np_img = np.array(img, dtype=np.float64)

    return np_img


def save_image(output, name, img_type):
    name = f"{name}.{img_type}"
    if img_type.lower() == "npz":
        np.savez_compressed(name, resultado=output)
    else:
        output = np.clip(output, 0, 255).astype(np.uint8)
        # Salva a imagem média
        if len(output.shape) == 2:
            img_final = Image.fromarray(output, mode="L")
        else:
            img_final = Image.fromarray(output, mode="RGB")
        img_final.save(name)
    print(f"Imagem salva como {name}")

def gradients(img, N):
    Ix = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=3)  # Derivada em x
    Iy = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=3)  # Derivada em y
```

```
    Ix2 = Ix * Ix
    Iy2 = Iy * Iy
    Ixy = Ix * Iy

    # Suavização
    ksize = 2 * N + 1
    Sx2 = cv2.GaussianBlur(Ix2, (ksize, ksize), 1)
    Sy2 = cv2.GaussianBlur(Iy2, (ksize, ksize), 1)
    Sxy = cv2.GaussianBlur(Ixy, (ksize, ksize), 1)
    return Sx2, Sy2, Sxy

def corners_detector(img, N=3, tau=1e-4):
    # 1. Converter para escala de cinza (caso seja RGB)
    if len(img.shape) == 3:
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    else:
        gray = img

    gray = np.float32(gray) / 255.0
    h, w = gray.shape

    # 2. Gradientes
    Sx2, Sy2, Sxy = gradients(gray, N)

    # Resposta λ2 em cada pixel
    lambda2 = np.zeros((h, w))
    pontos = []

    # 2. Para cada pixel p
    for y in range(h):
        for x in range(w):
            # a) Construir matriz C
            C = np.array([[Sx2[y, x], Sxy[y, x]], [Sxy[y, x], Sy2[y, x]]])

            # b) Calcular menor autovalor
            eigvals = np.linalg.eigvalsh(C)
            l2 = np.min(eigvals)
            lambda2[y, x] = l2

            # c) Testar com τ
            if l2 > tau:
                pontos.append((y, x, l2))

    # 3. Ordenar lista em ordem decrescente de λ2
    pontos.sort(key=lambda p: p[2], reverse=True)

    # 4. Supressão de não máximos: evitar vizinhanças sobrepostas
    final_points = []
    marcado = np.zeros((h, w), dtype=bool)

    for (y, x, val) in pontos:
        if not marcado[y, x]:
            final_points.append((y, x, val))
            # marcar vizinhança como ocupada
            y1, y2 = max(0, y - N), min(h, y + N + 1)
            x1, x2 = max(0, x - N), min(w, x + N + 1)
            marcado[y1:y2, x1:x2] = True
```

```
    return final_points, lambda2


# ============================================================
# Imagem Sintética
# ============================================================
img_synthetic = np.zeros((200, 200), dtype=np.uint8)
cv2.rectangle(img_synthetic, (50, 50), (150, 150), 255, -1)

corners, response = corners_detector(img_synthetic, N=3, tau=1e-4)

# Visualização
img_color = cv2.cvtColor(img_synthetic, cv2.COLOR_GRAY2BGR)
for y, x, val in corners:
    cv2.circle(img_color, (x, y), 3, (0, 0, 255), -1)

# Salvar resultados
save_image(img_color, "synthetic_corners", "png")
save_image(response, "synthetic_lambda2", "npz")

# --- Harris ---
harris = cv2.cornerHarris(np.float32(img_synthetic) / 255.0, 2, 3, 0.04)
harris_img = cv2.cvtColor(img_synthetic, cv2.COLOR_GRAY2BGR)
harris_img[harris > 0.01 * harris.max()] = [0, 0, 255]
save_image(harris_img, "synthetic_harris", "png")
save_image(harris, "synthetic_response", "npz")

# --- ORB ---
orb = cv2.ORB_create()
kp_orb = orb.detect(img_synthetic, None)
orb_img = cv2.drawKeypoints(img_synthetic, kp_orb, None, color=(0, 255, 0))
save_image(orb_img, "synthetic_orb", "png")

# --- SIFT ---
sift = cv2.SIFT_create()
kp_sift, _ = sift.detectAndCompute(img_synthetic, None)
sift_img = cv2.drawKeypoints(img_synthetic, kp_sift, None, color=(255, 0, 0))
save_image(sift_img, "synthetic_sift", "png")


# ============================================================
# building2-1.png
# ============================================================
img = cv2.imread("building2-1.png", cv2.IMREAD_GRAYSCALE)


corners, response = corners_detector(img, N=3, tau=0.05)

# Visualização
img_color = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)
for y, x, val in corners:
    cv2.circle(img_color, (x, y), 3, (0, 0, 255), -1)

# Salvar resultados
save_image(img_color, "building_corners", "png")
save_image(response, "building_lambda2", "npz")

# --- Harris ---
harris = cv2.cornerHarris(np.float32(img) / 255.0, 2, 3, 0.04)
```

```
harris_img = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)
harris_img[harris > 0.01 * harris.max()] = [0, 0, 255]
save_image(harris_img, "building_harris", "png")
save_image(harris, "building_harris_response", "npz")

# --- ORB ---
orb = cv2.ORB_create()
kp_orb = orb.detect(img, None)
orb_img = cv2.drawKeypoints(img, kp_orb, None, color=(0, 255, 0))
save_image(orb_img, "building_orb", "png")

# --- SIFT ---
sift = cv2.SIFT_create()
kp_sift, _ = sift.detectAndCompute(img, None)
sift_img = cv2.drawKeypoints(img, kp_sift, None, color=(255, 0, 0))
save_image(sift_img, "building_sift", "png")
```

Resultados:



Comentários finais:

Imagem sintética (quadrado branco em fundo preto)

Algoritmo CORNERS λ2:

- Detecta os quatro cantos do quadrado de forma bastante precisa.
- Como a estrutura é simples (alto contraste entre preto e branco), o autovalor mínimo da matriz de covariância de gradientes λ2 responde bem.
- Poucos pontos, localizados apenas nos cantos.

Harris:

- Também detecta corretamente os quatro vértices, mas pode marcar regiões na borda.
- Poucos pontos, localizados apenas nos cantos.

ORB (baseado em FAST + BRIEF):

- Detecta cantos, mas gera mais keypoints redundantes do que os algoritmos clássicos de canto.

SIFT:

- Detecta pontos de interesse multiescala, então além dos cantos, também pode marcar pontos no interior do quadrado (dependendo da vizinhança gaussiana analisada).
- Bom, mas claramente gera mais pontos que o necessário para essa imagem simples.
- No caso não achou os cantos

Resumo imagem sintética:

O CORNERS e Harris são mais seletivos e precisos (melhor para detectar vértices puros). Já ORB e SIFT marcam mais pontos, até redundantes, mas isso é uma vantagem quando pensamos em correspondência/robustez em imagens reais.

Imagem real (building2-1.png)

Aqui a análise fica mais interessante porque temos detalhes arquitetônicos com muitas estruturas de canto.

Algoritmo CORNERS:

- Detecta diversos pontos de interesse reais, mas tende a ser mais seletivo e não é multiescala.
- Detecta cantos bem definidos em janelas/prédios, mas pode perder detalhes em partes com textura mais fina.
- Como não possui normalização de escala, pode falhar em estruturas menores ou em cantos menos “puros”.

Harris:

- Responde muito bem nos contornos das janelas e bordas do edifício.
- Boa escolha quando o interesse é destacar apenas cantos pronunciados.

ORB:

- Detecta muitos keypoints espalhados pela cena (inclusive cantos menores, texturas repetitivas, detalhes finos).
- É eficiente, robusto e rápido, além de gerar descritores binários fáceis de usar em correspondência.

SIFT:

- Parecido com ORB, mas com maior diversidade de escalas e maior robustez a mudanças de contraste/iluminação.
- Gera keypoints bem distribuídos e estáveis em diferentes escalas.
- Excelente para aplicações de visão computacional robusta, como image stitching (costura de imagens de prédios).

Resumo imagem real:

- CORNERS/Harris: bons para detectar cantos principais, precisos mas não multiescala.
- ORB/SIFT: capturam muito mais detalhes e escalas diferentes, sendo mais adequados para tarefas práticas como reconhecimento e emparelhamento de imagens.

Comparação Geral

Algoritmo Características principais Resultado esperado CORNERS Baseado em λ2. Simples, seletivo e focado em detectar apenas cantos fortes Pega cantos principais, mas perde em robustez multiescala Harris Similar ao CORNERS (usa $R = \det(C) - k \cdot \text{trace}(C)^2$). Bom em cantos pronunciados, mas sensível ao threshold Detecta cantos bem, mas também bordas redundantes ORB Detector + descritor rápido, multiescala, com descritores binários (FAST + BRIEF). Muitos

pontos de interesse, bom para matching SIFT Robustez em várias escalas, invariância a rotação, contraste e iluminação. Excelente, mas mais pesado. Keypoints distribuídos e estáveis, excelente para aplicações reais

---

Conclusão:

- Em imagens sintéticas simples, CORNERS e Harris são mais limpos e precisos, detectando apenas os vértices.
- Em imagens reais e complexas, ORB e SIFT se destacam, pois capturam mais pontos de interesse em múltiplas escalas, o que os torna melhores para aplicações de reconhecimento, registro e matching de imagens.
- Assim, a escolha do detector depende do objetivo:
    - Se o objetivo é apenas detectar cantos puros → CORNERS/Harris. (O problema do CORNERS é que é muto lento, o Harris é mais rápido)
    - Se o objetivo é matching robusto entre imagens → ORB/SIFT.

.

.

.

.

.

.

2)

a)

Explique quais efeitos você observaria em uma imagem caso a distância focal da lente seja alterada para maior ou para menor.

Resposta

Se aumentamos a distância focal:

- Ampliação da imagem → os objetos da cena parecem maiores e "mais próximos".
- Campo de visão menor → a câmera "enxerga" uma área menor da cena (zoom in).
- Profundidade de campo diminui → regiões fora do foco ficam mais borradas, ou seja, só uma faixa mais estreita de distâncias estará realmente nítida.
- Perspectiva comprimida → objetos distantes parecem mais próximos uns dos outros, "achatando" a percepção de profundidade.

Se diminuímos a distância focal:

- Menor ampliação ⟶ os objetos parecem menores e "mais distantes".
- Campo de visão maior ⟶ a câmera captura uma área muito mais ampla da cena (zoom out).
- Profundidade de campo aumenta ⟶ mais objetos a diferentes distâncias aparecem focados ao mesmo tempo.
- Perspectiva expandida ⟶ objetos próximos ficam bem maiores e os distantes parecem bem menores, aumentando a sensação de profundidade.

Resumindo os efeitos:

Distância Focal Campo de Visão Tamanho Aparente Profundidade de Campo Efeito na Perspectiva Maior Menor (estreito) Objetos maiores Menor (foco seletivo) Cena "achatada" Menor Maior (amplo) Objetos menores Maior (mais foco) Cena "alongada"

b)

Explique o conceito de ruído em imagens, como pode ser quantificado, e como pode afetar o processamento das imagens em visão computacional.

Resposta

O que é ruído em imagens?

No contexto de Visão Computacional de Imagens, ruído é definido como qualquer variação ou informação indesejada que se mistura à imagem original, dificultando o cálculo, e assim impactando no resultado.

Ele pode surgir:

- Durante a aquisição da imagem (câmeras, sensores, transmissão, interferência eletromagnética);
- Durante o processamento digital (compressão, arredondamentos, erros numéricos);
- Por limitações físicas (grão em filmes fotográficos, iluminação não uniforme).
- Mas também pode ser só algo que veio de outro passo que atrapalha o passo atual, ou que não é de interesse para o cálculo atual.

Ele pode ser:

- flutuações espúrias nos valores de pixels introduzidas pelo sistema de aquisição de imagens, afetando a detecção de linhas ou contornos em algoritmos de processamento;
- variações aleatórias ou imprecisões nos dados de entrada, incluindo erros de precisão numérica e arredondamentos;
- contornos inexistentes, isto é, que não correspondem a nenhum objeto real, em algoritmos de agrupamento de linhas.

Matematicamente, uma imagem com ruído pode ser descrita como:

$$\hat{I}(i,j) = I(i,j) + n(i,j)$$

onde:

- Imagem corrompida (com ruído):
$$\hat{I}(i,j)$$
- Imagem ideal (sem ruído):
$$I(i,j)$$
- Ruído aditivo e aleatório:
$$n(i,j)$$

Em alguns casos, o ruído pode ser multiplicativo:

$$\hat{I}(i,j) = I(i,j) \cdot n(i,j)$$

Quantificação do ruído:

Um parâmetro importante é a Relação Sinal-Ruído (SNR):

$$SNR = \frac{\mu_s}{\sigma_n}$$

$$SNR_{dB} = 10 \cdot \log_{10}\left(\frac{\mu_s}{\sigma_n}\right)$$

ou

$$SNR_{dB} = 20 \cdot \log_{10}\left(\frac{\mu_s}{\sigma_n}\right)$$

onde:

- Média do sinal (valores médios dos pixels da imagem original):
$$\mu_s$$
- Desvio padrão do ruído:
$$\sigma_n$$

Então:

- Quanto maior o SNR, melhor a qualidade da imagem (menos ruído em relação ao sinal).
- Quanto menor o SNR, mais contaminada a imagem está.

Como o ruído afeta o processamento de imagens

O ruído pode comprometer diretamente algoritmos de visão computacional. Exemplos:

- Detecção de bordas e contornos: pequenas flutuações podem ser confundidas com bordas falsas.
- Segmentação: regiões homogêneas ficam “quebradas” pelo ruído.
- Reconhecimento de padrões: objetos podem ser mal identificados devido à presença de artefatos.
- Compressão: taxas de erro aumentam, pois o ruído ocupa espaço de codificação.

Ou seja, quanto mais ruído, menos confiáveis são os resultados.

c)

Explique por que a precisão de amostragem de um filtro Gaussiano 1-D com σ = 0,6 não pode ser melhorada utilizando mais de três amostras espaciais (largura espacial maior que 3 pixels).

Resposta

1. Máscara Gaussiana e relação entre w' e σ'

- A largura da máscara Gaussiana normalmente é escolhida como

$$w = 5 \cdot \sigma$$
$$5/\sigma <= 2 \cdot \pi$$
$$\sigma >= 5/(2 \cdot \pi)$$
$$\sigma >\approx 0.8$$

pois nesse intervalo já está contida cerca de 98,76% da energia da Gaussiana.

Exemplo:

- Se (w = 3), então (\sigma \approx 0,6)
- Se (w = 5), então (\sigma \approx 1)

---

2. Limite de resolução no domínio da frequência

A transformada de Fourier de uma Gaussiana é também uma Gaussiana, mas no domínio da frequência:

$$\text{T.F.}\{g(x, \sigma)\} = g(x', \sigma')$$

onde:

$$\sigma' = \frac{1}{\sigma}$$

Assim:

- Quanto menor σ no espaço, mais larga é a Gaussiana em frequência.
- Isso significa que filtros Gaussianos de σ pequeno possuem muitos componentes de alta frequência.

Porém, devido à amostragem espacial (1 pixel como passo de amostragem), o máximo componente de frequência perceptível é:

$$\nu_c = \frac{\pi}{d}, \quad d = 1 \text{ pixel}$$

Logo, frequências maiores que π não podem ser representadas corretamente — sofrem aliasing ou simplesmente se perdem.

---

3. Por que aumentar a largura (w > 3) não melhora a precisão em σ = 0,6

- Para σ = 0,6 (caso w=3):
    - A Gaussiana no domínio da frequência já tem bastante energia além do intervalo [-π, π].

    - Ou seja, mesmo que você aumente a largura (w) no espaço (mais amostras), as frequências perdidas não são recuperáveis porque estão fora da banda perceptível (além de π).
- Se você tentar usar w=5 (isto é, máscara mais larga com mesmo σ=0,6):
    - O valor das novas amostras adicionadas (nas extremidades) é próximo de zero (porque a Gaussiana decai muito rápido em ±2.1σ).
    - Portanto, essas amostras extras não trazem informação significativa.
    - O resultado numérico é praticamente idêntico ao filtro com w=3.

---

4. Conclusão

A precisão não melhora porque:

1. No espaço: a contribuição fora da janela de 3 amostras já é praticamente nula devido à concentração da Gaussiana.
2. No domínio da frequência: aumentar w não traz de volta as frequências já perdidas, que estão além de $\pm\pi$.

Portanto, para σ = 0,6, já é suficiente uma máscara de largura w = 3. Usar mais amostras só aumenta o custo computacional sem melhorar a aproximação.

d)

Dadas as definições e classificações de bordas, discuta as diferenças entre bordas em imagens de intensidade e de profundidade. Quais as diferenças nos algoritmos para detectá-las e localizá-las?

Resposta

Bordas em Imagens de Intensidade

Definição:

São regiões da imagem em que existe uma variação brusca de intensidade luminosa (valores de cinza ou cor).

Exemplo: contorno de um objeto iluminado, sombra projetada, mudança de textura.

Fonte da Informação:

A intensidade é derivada do processo fotométrico (iluminação + refletância da superfície + captura pela câmera).

Isso significa que as bordas de intensidade podem refletir não apenas transições reais de objetos, mas também efeitos de iluminação (sombras, reflexos, brilho, texturas).

Pode também ser gerado computacionalmente através de renderização de uma cena 3d.

Modelo matemático:

Normalmente representamos a intensidade como uma função I(x, y).

As bordas aparecem onde o gradiente (primeira derivada) é máximo:

$$\nabla I(x,y) = \left(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y} \right)$$

A magnitude do gradiente dá a "força" da borda, e sua orientação dá a direção.

Algoritmos típicos:

- Filtros de gradiente: Roberts, Prewitt, Sobel.
- Detectores baseados em suavização + gradiente: Canny.
- Métodos de zero-crossing: Laplaciano do Gaussiano (LoG).

Bordas em Imagens de Profundidade

São regiões em uma imagem de profundidade (Depth Map) ou em um modelo 3D reconstruído, onde ocorre variação brusca de distância (Z em relação à câmera).

Exemplo: borda de descontinuidade profunda (ângulo de uma caixa vista pela câmera) ou mudança no plano da cena.

Fonte da Informação:

Diferente das bordas de intensidade, agora a informação vem de um sensor de profundidade (ex.: câmera estéreo, LiDAR, Kinect, estrutura de luz).

Logo, as bordas correspondem mais diretamente à geometria real da cena, e não são tão influenciadas pela iluminação.

Pode também ser gerado computacionalmente através de renderização de uma cena 3d.

Modelo matemático:

Representamos a profundidade como D(x, y).

As bordas aparecem onde o gradiente de profundidade é elevado:

$$\nabla D(x,y) = \left(\frac{\partial D}{\partial x}, \frac{\partial D}{\partial y} \right)$$

Além disso, também podem ser detectadas descontinuidades em superfícies planas, onde não há uma transição suave, mas sim uma quebra abrupta na profundidade.

Algoritmos típicos:

- Gradiente da profundidade (semelhante a intensidade, mas aplicado ao mapa de profundidade).
- Detecção de descontinuidades (thresholding no gradiente de profundidade).
- Cálculo de normais de superfície: mudanças bruscas nas normais definem bordas geométricas.
- Métodos baseados em ajuste de planos locais (detectar onde o plano não se ajusta mais bem — quinas e arestas).

Principais Diferenças entre Bordas de Intensidade e de Profundidade

Aspecto Bordas de Intensidade Bordas de Profundidade Origem Fotometria (mudança de luz, textura, cor, sombra, refletância). Geometria (mudança no valor de profundidade real). Sensibilidade a condições externas Muito sensíveis à iluminação, sombras, reflexos. Menos dependentes de iluminação, mas sensíveis a ruído do sensor de profundidade. Algoritmo básico Gradiente da intensidade I(x, y), suavização + derivadas. Gradiente da profundidade D(x, y), descontinuidade da geometria ou mudança abrupta de normal. Ruído Ruído fotométrico (granulação, compressão da imagem, baixa iluminação). Ruído de range sensing (erros de disparidade em estéreo, ruído estrutural em Kinect ou LiDAR). Resultado esperado Pode detectar bordas falsas causadas por sombras/texturas. Mais fiel às bordas estruturais dos objetos na cena.

Resumindo

- Em intensidade, borda ≈ forte variação de níveis de cinza (dependente de iluminação e textura).

→ Algoritmos: gradiente, Canny, Sobel, LoG.

- Em profundidade, borda ≈ descontinuidade geométrica (mudança brusca de Z ou de normais de superfície).

→ Algoritmos: gradiente de profundidade, detecção de descontinuidades, análise de normais.

Ou seja:

- Bordas em intensidade dependem de luz → podem confundir sombra com objeto.
- Bordas em profundidade dependem de geometria real → mas sofrem com ruído estrutural do sensor.

e)

Explique por que a segmentação H-K não pode ser aplicada a imagens de intensidade na expectativa de se encontrarem cenas de superfícies homogêneas.

Resposta

A segmentação H-K baseia-se em conceitos de geometria diferencial e requer que cada ponto da imagem represente uma coordenada no espaço 3D, ou seja, um ponto sobre uma superfície definida como (x, y, z).

- Em imagens de intensidade (RGB ou em tons de cinza), cada pixel só carrega valores de intensidade da luz refletida (ou luminosidade da cena), sem nenhuma correspondência direta com altura/profundidade.
- A técnica precisa da informação de profundidade (terceira dimensão) para que seja possível calcular as derivadas parciais da superfície (como
  $$h_x, h_y, h_{xx}, h_{yy}, h_{xy}$$
) que permitem estimar as curvaturas média (H) e gaussiana (K).
- Em uma imagem de intensidade, não existe superfície explícita a ser diferenciada — só existem variações de brilho que não correspondem diretamente a variações geométricas da

forma no espaço 3D.

Por isso:

- Se tentamos aplicar H-K em imagens de intensidade, estaríamos medindo "curvatura" sobre o campo 2D de brilho, e não da superfície real do objeto.
- Regiões homogêneas em intensidade não correspondem necessariamente a regiões planas no espaço físico. Um objeto com a mesma cor, mas com relevos diferentes, teria uma intensidade homogênea mas não seria plano em 3D.
- Da mesma forma, diferenças de iluminação ou sombras podem gerar fortes variações de intensidade, que não têm nada a ver com curvatura real da superfície.

Resposta resumida:

A segmentação H-K não pode ser aplicada diretamente em imagens de intensidade porque não existe relação direta entre variações de intensidade e curvaturas geométricas da superfície. A técnica depende de informações de profundidade z = h(x,y) para que seja possível calcular as curvaturas locais (H e K). Em intensidade, só temos brilho/luminosidade, e não a forma 3D, logo não é possível identificar corretamente regiões homogêneas da superfície.

.

.

.

.

.

.

.

.

.

.