Lab 2b: Plug-and-Play ADMM for Pixel-wise Inpainting

Advanced Image Processing

Goal. Solve a pixel-wise inpainting problem with Plug-and-Play ADMM, using a classical denoiser as the prior. Run the blocks, edit the `# EDIT` lines, and collate PSNR/LPIPS and figures yourself.

Dependencies. PyTorch, torchvision, matplotlib, scikit-image, PIL, optional `lpips`. If needed: `pip install scikit-image lpips`.

Block 0: Setup and helpers (run once)

```
# Block 0: imports, metrics, denoisers, reproducibility
import os; os.makedirs("results", exist_ok=True)
import math, numpy as np, torch, torch.nn.functional as F
import torchvision.transforms as T
import matplotlib.pyplot as plt
from skimage import data, restoration, filters, morphology
from PIL import Image
import random

# --- Reproducibility ---
SEED = 12345   # EDIT if desired
random.seed(SEED); np.random.seed(SEED); torch.manual_seed(SEED)
if torch.cuda.is_available(): torch.cuda.manual_seed_all(SEED)

device = "cuda" if torch.cuda.is_available() else "cpu"

# Optional LPIPS
USE_LPIPS = True
try:
    import lpips
    lpips_fn = lpips.LPIPS(net='alex').eval().to(device) if USE_LPIPS else None
except Exception:
    lpips_fn = None
    print("LPIPS disabled or missing. pip install lpips to enable.")

def load_rgb01():
    img = Image.fromarray(data.astronaut())
    x = T.ToTensor()(img).unsqueeze(0)  # [1,3,H,W] in [0,1]
    return x.to(device)

def psnr(ref, out, eps=1e-12):
    mse = F.mse_loss(out, ref)
    return float(10.0 * torch.log10(1.0 / (mse + eps)))

def lpips_score(ref, out):
    if lpips_fn is None: return None
    with torch.no_grad():
        return float(lpips_fn(ref.clamp(0,1)*2-1, out.clamp(0,1)*2-1).mean().item())

# Denoisers (numpy HWC -> numpy HWC), then back to torch
```

```
def to_numpy_img(x):
    return x.detach().cpu().permute(0,2,3,1).numpy()[0]  # HWC in [0,1]

def to_torch_img(arr, like):
    t = torch.from_numpy(arr).permute(2,0,1).unsqueeze(0).to(like.device, dtype=like.
    dtype)
    return t.clamp(0,1)

DRUNET = None

def get_drunet(device):
    global DRUNET
    if DRUNET is not None:
        return DRUNET
    try:
        import deepinv as dinv
        from deepinv.models import DRUNet
        DRUNET = DRUNet(pretrained="download", device=device).eval()  # auto-download
    weights
        return DRUNET
    except Exception as e:
        print("DRUNet unavailable. Install with: pip install deepinv")
        print("Error:", e)
        return None

def denoise_image(img_np, kind="bilateral", strength=0.1, median_size=3):
    """
    kind: "median" | "bilateral" | "nlm" | "drunet"
    strength:
        - bilateral: sigma_color ( 0.020.2)
        - nlm: h ( 0.050.2)
        - drunet: noise sigma in [0,1] (e.g., 0.05 for 5% of 0-1 range)
    """
    import numpy as np
    from skimage import restoration, filters, morphology

    if kind == "median":
        selem = morphology.disk(max(1, int(median_size)))
        out = np.zeros_like(img_np)
        for c in range(img_np.shape[2]):
            out[:, :, c] = filters.median(img_np[:, :, c], footprint=selem)
        return out

    elif kind == "bilateral":
        return restoration.denoise_bilateral(img_np, sigma_color=strength, sigma_spatial
    =3, channel_axis=-1)

    elif kind == "nlm":
        patch_kw = dict(patch_size=3, patch_distance=5, channel_axis=-1, fast_mode=True)
        return restoration.denoise_nl_means(img_np, h=strength, **patch_kw)

    elif kind == "drunet":
        # DRUNet expects torch tensor in NCHW, [0,1], and a noise level sigma in [0,1]
        import torch
        x = torch.from_numpy(img_np).permute(2,0,1).unsqueeze(0).float()  # 1xCxHxW
        x = x.to(device)
        model = get_drunet(device)
        if model is None:
            return img_np  # fallback no-op if not installed
        with torch.no_grad():
```

```
            # DRUNet signature: y = model(x, sigma) ; sigma \in [0,1]
            y = model(x, float(strength)).clamp(0,1)
        return y.squeeze(0).permute(1,2,0).cpu().numpy()

    else:
        return img_np  # no-op
```

Block 1: Build random inpainting mask and observations

```
# Block 1: pixel-wise mask with p% dropped pixels (same mask for all channels)
x_clean = load_rgb01()
B,C,H,W = x_clean.shape
N = H*W

p = 20.0            # EDIT: percentage of pixels to drop (0..100)
noise_sigma = 0.00  # EDIT: optional AWGN on observed pixels (0.0 = off)

# Construct mask: 1 = observed, 0 = missing
num_drop = int(max(0.0, min(100.0, p)) * 0.01 * N)
idx = np.arange(N); np.random.shuffle(idx)
drop = idx[:num_drop]
M = np.ones((N,), dtype=np.float32); M[drop] = 0.0
M = torch.from_numpy(M.reshape(H, W)).to(device)  # [H,W]
M3 = M.unsqueeze(0).unsqueeze(0).repeat(1,C,1,1)  # [1,C,H,W] for broadcasting

# Observations: keep observed pixels, missing set to 0
y = x_clean.clone()
if noise_sigma > 0:
    y = (y + noise_sigma * torch.randn_like(y)).clamp(0,1)
y = y * M3

kept_pct = 100.0 * (1.0 - num_drop / max(1, N))
print(f"Mask coverage: kept {kept_pct:.1f}% pixels (p = {p:.1f}% masked)")
print("Observed   PSNR (with zeros at missing):", psnr(x_clean, y), "LPIPS:", lpips_score
    (x_clean, y))

plt.figure(figsize=(12,3))
plt.subplot(1,3,1); plt.imshow(x_clean[0].permute(1,2,0).cpu().clip(0,1));  plt.axis('off
    '); plt.title("Clean")
plt.subplot(1,3,2); plt.imshow(M.cpu(), vmin=0, vmax=1);                      plt.axis('
    off'); plt.title("Mask (white=obs)")
plt.subplot(1,3,3); plt.imshow(y[0].permute(1,2,0).cpu().clip(0,1));         plt.axis('
    off'); plt.title("Observed (masked)")
plt.tight_layout(); plt.savefig("results/lab2b_inpaint_data.png", dpi=200); plt.close()
```

Block 2: PnP-ADMM for inpainting (closed-form x-update + convergence plot)

```
# Block 2: PnP-ADMM with data term 0.5*|| M * (x - y) ||^2  (elementwise *)
rho    = 0.2        # EDIT if desired (ADMM penalty)
iters  = 40         # EDIT if desired (iterations)
kind   = "drunet"  # EDIT: "median" | "bilateral" | "nlm" | "drunet"
strength = 0.08     # EDIT: denoiser strength
median_size = 3     # EDIT: for median only
```

```
# init
x = y.clone()
z = y.clone()
u = torch.zeros_like(y)

psnr_hist = []
r_norm, s_norm = [], []  # ADMM residuals

for t in range(1, iters+1):
    # x-update: (M + rho I) x = M * y + rho * (z - u)  --> element-wise
    rhs = y + rho * (z - u)
    denom = M3 + rho       # broadcast over channels
    x = (rhs / denom).clamp(0,1)

    # z-update: PnP denoiser on x + u
    x_plus_u = (x + u).clamp(0,1)
    img_np = to_numpy_img(x_plus_u)
    den_np = denoise_image(img_np, kind=kind, strength=strength, median_size=median_size)
    z_new = to_torch_img(den_np, x_plus_u)

    # residuals
    r = (x - z_new).reshape(-1)
    s = (rho * (z_new - z)).reshape(-1)
    r_norm.append(float(torch.norm(r).item()))
    s_norm.append(float(torch.norm(s).item()))

    z = z_new
    u = u + x - z

    # metric
    psnr_hist.append(psnr(x_clean, x))

print("PnP-ADMM (inpaint) PSNR:", psnr(x_clean, x), "LPIPS:", lpips_score(x_clean, x))

# Convergence plots
plt.figure()
plt.plot(range(1,len(psnr_hist)+1), psnr_hist, marker='o')
plt.xlabel("Iteration"); plt.ylabel("PSNR (dB)")
plt.tight_layout(); plt.savefig("results/lab2b_inpaint_psnr_curve.png", dpi=200); plt.
    close()

plt.figure()
plt.plot(range(1,len(r_norm)+1), r_norm, label="primal ||x - z||")
plt.plot(range(1,len(s_norm)+1), s_norm, label="dual ||rho*(z - z_prev)||")
plt.yscale('log'); plt.xlabel("Iteration"); plt.ylabel("Residual norm")
plt.legend(); plt.tight_layout(); plt.savefig("results/lab2b_inpaint_residuals.png", dpi
    =200); plt.close()

plt.figure(figsize=(12,3))
plt.subplot(1,4,1); plt.imshow(y[0].permute(1,2,0).cpu().clip(0,1));       plt.axis('off'
    ); plt.title("Input y (masked)")
plt.subplot(1,4,2); plt.imshow(z[0].permute(1,2,0).cpu().clip(0,1));       plt.axis('off'
    ); plt.title("Denoised z")
plt.subplot(1,4,3); plt.imshow(x[0].permute(1,2,0).cpu().clip(0,1));       plt.axis('off'
    ); plt.title("PnP x (inpaint)")
plt.subplot(1,4,4); plt.imshow(x_clean[0].permute(1,2,0).cpu().clip(0,1)); plt.axis('off'
    ); plt.title("Clean")
plt.tight_layout(); plt.savefig("results/lab2b_inpaint_pnp.png", dpi=200); plt.close()
```

Block 3: Simple baselines for context

```
# Baseline A: zero-fill (just the observed y)
rec_a = y.clone()

# Baseline B: nearest-neighbor fill using a small blur on missing (quick-and-dirty)
from scipy.ndimage import distance_transform_edt

def nn_fill(img_np, mask_np):
    """
    Fill missing pixels (mask=0) with the value of the nearest known pixel (mask=1).
    Nearest neighbour in Euclidean distance.
    img_np: HxWxC, floats in [0,1] or uint8
    mask_np: HxW, 1=known, 0=missing
    """
    out = img_np.copy()
    # Compute distance transform: returns indices of nearest known pixel
    dist, (idx_y, idx_x) = distance_transform_edt(1 - mask_np,
                                                  return_indices=True)
    # Fill missing pixels for each channel from nearest known pixel
    for c in range(img_np.shape[2]):
        out[..., c] = img_np[idx_y, idx_x, c]
    return out

mask_np = M.cpu().numpy()
img_np  = to_numpy_img(y)
fill_np = nn_fill(img_np, mask_np)
rec_b   = to_torch_img(fill_np, y)

print("Zero-fill   PSNR:", psnr(x_clean, rec_a), "LPIPS:", lpips_score(x_clean, rec_a))
print("NN-fill     PSNR:", psnr(x_clean, rec_b), "LPIPS:", lpips_score(x_clean, rec_b))

plt.figure(figsize=(12,3))
plt.subplot(1,3,1); plt.imshow(rec_a[0].permute(1,2,0).cpu().clip(0,1)); plt.axis('off');
     plt.title("Baseline: zero-fill")
plt.subplot(1,3,2); plt.imshow(rec_b[0].permute(1,2,0).cpu().clip(0,1)); plt.axis('off');
     plt.title("Baseline: nn-fill")
plt.subplot(1,3,3); plt.imshow(x[0].permute(1,2,0).cpu().clip(0,1));     plt.axis('off');
     plt.title("PnP x")
plt.tight_layout(); plt.savefig("results/lab2b_inpaint_baselines.png", dpi=200); plt.
    close()
```

Tasks

This lab familiarizes you with an important iterative solver: the alternating direction method of multipliers (ADMM). Regularization is provided by different denoisers. The theory for this will be explored in future lectures.

- How is the ADMM parametrization here different from the definitions in the lecture?
- Choose a value of `p` (percentage). Run PnP-ADMM for different denoisers (`median`, `bilateral`, `nlm`).
- If you have a GPU (or want to try it anyways), try "drunet" as denoiser (this is a very powerful deep denoiser that can lead to impressive restorations).
- Sweep denoiser strength (start with powers of 10:1, 0.1, 0.01,...) and ρ (start with default value and then deviate until it breaks). Choose the smallest values that remove artifacts without over-smoothing. For drunet try strength = sigma.
- Try different levels of inpainting (different p) and noise (different sigma).

- Save figures and record PSNR (and LPIPS if enabled). Compare against zero-fill and nearest neighbor (NN)-fill baselines.
- Inspect the convergence plots (PSNR curve and ADMM residuals). Increase iterations if curves have not plateaued.
- How do the results compare to nearest neighbor inpainting? When would we prefer ADMM?