Lab 3a: PnP-ADMM vs PnP-Forward-Backward (Deterministic vs DRUNet Denoiser)

Advanced Image Processing

Goal. Compare PnP-ADMM and PnP-Forward-Backward (FB) on deblurring with both a deterministic denoiser and a neural denoiser (DRUNet). Study early stopping and step-sizes.

Block 0: Setup and helpers (run once)

```python
# Block 0: imports, metrics, reproducibility, classical denoisers
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

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():
    from skimage import data
    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())

# Classical 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]
```

```
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)

# Optional neural denoiser (DRUNet) via deepinv
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()
        return DRUNET
    except Exception as e:
        print("DRUNet unavailable. Install: 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"
    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=float(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=float(strength), **patch_kw)
    elif kind == "drunet":
        model = get_drunet(device)
        if model is None:
            return img_np
        x = torch.from_numpy(img_np).permute(2,0,1).unsqueeze(0).float().to(device)
        with torch.no_grad():
            y = model(x, float(strength)).clamp(0,1)
        return y.squeeze(0).permute(1,2,0).cpu().numpy()
    else:
        return img_np
```

Block 1: Deblurring data

```
def gaussian_kernel_2d(size=15, std=3.0):
    ax = np.arange(size) - (size-1)/2.0
    xx, yy = np.meshgrid(ax, ax)
    k = np.exp(-(xx**2 + yy**2)/(2.0*std**2)); k = k/k.sum()
    return k.astype(np.float32)

def convolve_circular(x, kernel_np):
    k = torch.from_numpy(kernel_np).to(x.device, dtype=x.dtype).unsqueeze(0).unsqueeze(0)
    pad = kernel_np.shape[0]//2
    out = torch.zeros_like(x)
    for c in range(x.shape[1]):
        chp = F.pad(x[:,c:c+1], (pad,pad,pad,pad), mode="circular")
        out[:,c:c+1] = F.conv2d(chp, k, padding=0)
    return out

x_clean = load_rgb01()
k_np = gaussian_kernel_2d(size=15, std=3.0)  # EDIT
xb = convolve_circular(x_clean, k_np)
sigma = 0.03  # EDIT
y = (xb + sigma*torch.randn_like(xb)).clamp(0,1)

print("Blur-only  PSNR:", psnr(x_clean, xb), "LPIPS:", lpips_score(x_clean, xb))
print("Observed   PSNR:", psnr(x_clean, y),  "LPIPS:", lpips_score(x_clean, y))

# Save context images
plt.imsave("results/lab3a_clean.png", to_numpy_img(x_clean))
plt.imsave("results/lab3a_blurred.png", to_numpy_img(xb))
plt.imsave("results/lab3a_observed.png", to_numpy_img(y))
```

Block 2: PnP-ADMM (x,z,u; Fourier x-update)

```
def psf_fft(kernel_np, shape_hw, device):
    hk, wk = kernel_np.shape
    H, W = shape_hw
    pad = torch.zeros((H,W), dtype=torch.float32, device=device)
    pad[:hk,:wk] = torch.from_numpy(kernel_np).to(device)
    pad = torch.roll(pad, shifts=(-hk//2, -wk//2), dims=(0,1))
    return torch.fft.fft2(pad)

B,C,H,W = y.shape
Kf = psf_fft(k_np, (H,W), device)
Yf = torch.fft.fft2(y)
Hty_f = torch.conj(Kf) * Yf

rho    = 0.2   # EDIT
iters  = 100   # EDIT
kind   = "bilateral"  # EDIT: "median" | "bilateral" | "nlm" | "drunet"
strength = 0.08       # EDIT: denoiser strength

denom = (torch.abs(Kf)**2 + rho)

x = y.clone(); z = y.clone(); u = torch.zeros_like(y)
psnr_hist_admm = []; r_hist=[]; s_hist=[]
for t in range(1, iters+1):
    rhs_f = Hty_f + rho * torch.fft.fft2(z - u)
    x = torch.real(torch.fft.ifft2(rhs_f / denom)).clamp(0,1)

    x_plus_u = (x + u).clamp(0,1)
    den_np = denoise_image(to_numpy_img(x_plus_u), kind=kind, strength=strength,
    median_size=3)
    z_new = to_torch_img(den_np, x_plus_u)

    r = torch.ravel(x - z_new); s = torch.ravel(rho*(z_new - z))
    r_hist.append(float(torch.norm(r))); s_hist.append(float(torch.norm(s)))
    z = z_new; u = u + x - z
    psnr_hist_admm.append(psnr(x_clean, x))

print("PnP-ADMM PSNR:", psnr(x_clean, x), "LPIPS:", lpips_score(x_clean, x))
plt.figure(); plt.plot(psnr_hist_admm); plt.xlabel("iter"); plt.ylabel("PSNR"); plt.
    tight_layout()
plt.savefig("results/lab3a_pnpadmm_psnr.png", dpi=200); plt.close()

# Save final reconstruction
plt.imsave("results/lab3a_pnpadmm_rec.png", to_numpy_img(x))
```

Block 3: PnP-Forward-Backward (FB)

```
def H(x):  return convolve_circular(x, k_np)
def HT(x): return convolve_circular(x, k_np)  # symmetric psf

L = float((torch.abs(Kf)**2).max().item())
tau = 0.9 / L   # EDIT
iters_fb = 100  # EDIT
kind_fb   = kind
strength_fb = strength

x_fb = y.clone()
psnr_hist_fb=[]
for t in range(iters_fb):
    grad = HT(H(x_fb) - y)
    v = (x_fb - tau*grad).clamp(0,1)
    den_np = denoise_image(to_numpy_img(v), kind=kind_fb, strength=strength_fb,
    median_size=3)
    x_fb = to_torch_img(den_np, v)
    psnr_hist_fb.append(psnr(x_clean, x_fb))

print("PnP-FB PSNR:", psnr(x_clean, x_fb), "LPIPS:", lpips_score(x_clean, x_fb))
plt.figure(); plt.plot(psnr_hist_fb); plt.xlabel("iter"); plt.ylabel("PSNR"); plt.
    tight_layout()
plt.savefig("results/lab3a_pnpfb_psnr.png", dpi=200); plt.close()

# Save final reconstruction
plt.imsave("results/lab3a_pnpfb_rec.png", to_numpy_img(x_fb))
```

Tasks (concise)

- Compare PnP-ADMM vs PnP-FB with the same denoiser; inspect early stopping.
- Try deterministic denoisers vs DRUNet. Tune strength.
- Investigate different blur levels (kernel size and sigma) as well as noise-levels
- Sweep rho (ADMM) and tau (FB). Report smallest values that remove artifacts without over-smoothing.