Lab 3b: Matching Pursuit (MP)
vs
Orthogonal Matching Pursuit (OMP)

Advanced Image Processing

Goal

In this lab we explore and compare two greedy sparse approximation algorithms, **Matching Pursuit (MP)** and **Orthogonal Matching Pursuit (OMP)**. We will study them on a small toy problem with an orthogonal and a non-orthogonal dictionary, then extend to a full image using an implicit dictionary.

Background

- MP iteratively selects the atom with the highest correlation to the current residual and updates the residual by subtracting its contribution.
- OMP selects the best atom at each step but re-optimizes all coefficients jointly over the selected atoms (orthogonal projection).
- For orthogonal dictionaries, MP and OMP behave very similarly. For non-orthogonal or overcomplete dictionaries, OMP generally achieves higher reconstruction accuracy.

Toy Example Setup

We take the astronaut image, convert it to grayscale, and downsample it to a low resolution (default 32×32). Two dictionaries are used:

1. An orthogonal DCT dictionary.
2. A non-orthogonal, overcomplete dictionary formed by linear combinations of DCT atoms.

We then run MP and OMP with different sparsity levels T (number of atoms) and measure the PSNR.

Code

```
#!/usr/bin/env python3
# Lab 3b: MP vs OMP (Toy Example)
# Astronaut image -> grayscale -> downsample to TOY_SIZE x TOY_SIZE

import os, math, numpy as np, torch
import matplotlib.pyplot as plt
from skimage import data, color, transform

TOY_SIZE = 32   # toy example resolution
JITTER   = 0.0  # plot marker offset
T_SAVE   = 200  # T at which to save reconstructions

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
os.makedirs("results", exist_ok=True)
```

```python
# --- Utilities ---
def psnr_vec(x, xh, eps=1e-12):
    x, xh = x.reshape(-1), xh.reshape(-1)
    mse = torch.mean((xh - x) ** 2)
    return float(10.0 * torch.log10(1.0 / (mse + eps)))

_DCT_CACHE = {}
def _dct_mat(N, device, dtype):
    key = (N, str(device), str(dtype))
    if key in _DCT_CACHE:
        return _DCT_CACHE[key]
    n = torch.arange(N, device=device, dtype=dtype).unsqueeze(0)
    k = torch.arange(N, device=device, dtype=dtype).unsqueeze(1)
    C = torch.cos((math.pi / N) * (n + 0.5) * k)
    C[0, :] *= (1.0 / math.sqrt(N))
    C[1:, :] *= math.sqrt(2.0 / N)
    _DCT_CACHE[key] = C
    return C

def build_orthonormal_dct_dict(N):
    C = _dct_mat(N, torch.device("cpu"), torch.float32)
    U2D = torch.kron(C, C)
    U2D = U2D / (torch.norm(U2D, dim=0, keepdim=True) + 1e-12)
    return U2D

def make_nonorth_overcomplete(U_cpu, alpha=0.95):
    n = U_cpu.shape[1]
    I = torch.eye(n)
    P = torch.roll(I, shifts=1, dims=1)
    V = U_cpu @ (I + alpha * P)
    V = V / (torch.norm(V, dim=0, keepdim=True) + 1e-12)
    return torch.cat([U_cpu, V], dim=1)

# --- Algorithms ---
def mp_explicit(y_vec, D, T):
    y = y_vec.clone()
    n, m = D.shape
    a = torch.zeros(m, device=D.device, dtype=D.dtype)
    r = y.clone()
    Dt = D.t()
    for _ in range(T):
        c = Dt @ r
        j = int(torch.argmax(torch.abs(c)))
        a[j] = a[j] + c[j]
        r = r - D[:, j] * c[j]
    return D @ a, a

def omp_explicit(y_vec, D, T):
    y = y_vec.clone()
    n, m = D.shape
    S = []
    a = torch.zeros(m, device=D.device, dtype=D.dtype)
    Dt = D.t()
    r = y.clone()
    for _ in range(T):
        c = Dt @ r
        j = int(torch.argmax(torch.abs(c)))
        if j not in S: S.append(j)
        Ds = D[:, S]
        a_S, *_ = torch.linalg.lstsq(Ds, y)
        r = y - Ds @ a_S
    a[S] = a_S
    return D @ a, a, S

# --- Run toy block ---
img = transform.resize(color.rgb2gray(data.astronaut()),
```

```
                        (TOY_SIZE, TOY_SIZE), anti_aliasing=True).astype(np.float32)
x_cpu = torch.from_numpy(img).float().cpu()
y_vec = x_cpu.reshape(-1)

U = build_orthonormal_dct_dict(TOY_SIZE)
D_non = make_nonorth_overcomplete(U, alpha=0.95)

T_vals = [5, 10, 40, 80, 200, 400, 600, 1000]
ps_mp_ortho, ps_omp_ortho, ps_mp_non, ps_omp_non = [], [], [], []
for T in T_vals:
    ps_mp_ortho.append(psnr_vec(y_vec, mp_explicit(y_vec, U, T)[0]))
    ps_omp_ortho.append(psnr_vec(y_vec, omp_explicit(y_vec, U, T)[0]))
    ps_mp_non.append(psnr_vec(y_vec, mp_explicit(y_vec, D_non, T)[0]))
    ps_omp_non.append(psnr_vec(y_vec, omp_explicit(y_vec, D_non, T)[0]))

Ta = np.array(T_vals, dtype=float)
plt.figure()
plt.plot(Ta - JITTER, ps_mp_ortho, "o-", label="MP (orthogonal)")
plt.plot(Ta + JITTER, ps_omp_ortho, "^-", label="OMP (orthogonal)")
plt.plot(Ta - JITTER, ps_mp_non, "s--", label="MP (non-orthogonal)")
plt.plot(Ta + JITTER, ps_omp_non, "D--", label="OMP (non-orthogonal)")
plt.xlabel("T (target sparsity)")
plt.ylabel("PSNR (dB)")
plt.legend()
plt.tight_layout()
plt.savefig("results/toy_psnr_vs_T.png", dpi=200)
plt.close()
```

Tasks

1. Run the toy example script and generate the PSNR plot.
2. Review the lecture's description of MP and OMP. Note where the algorithm steps are in the code and how they work.
3. Compare MP and OMP for orthogonal vs non-orthogonal dictionaries.
4. Save and inspect reconstructions at a fixed T (e.g. $T = 800$).
5. Why do OMP and MP differ when the dictionary is non-orthogonal?
6. When is the non-orthogonal dictionary better? Why?
7. What happens for different image-sizes? (try 16, 32, 64 and modify evaluated Ts accordingly)