

ДОДАТОК А

Код розробленого алгоритму

А.1 Клас Agent

Лістинг А.1 – Код розробленого класу Agent

```

"""
Agent for MAS consensus-ADMM image denoising.

Each agent owns one image patch and is responsible for ALL three
ADMM
update steps that concern it – x, z, and u – acting only on
information
it has received from neighbours via message passing.

Message protocol (simulated, synchronous):
    - After the x-update each agent deposits a per-edge outbox
message:
        { 'x_overlap': x_i[O_ij], 'u': u_ij }
    - Before the z/u-update it reads the matching inbox message
from j:
        { 'x_overlap': x_j[O_ij], 'u': u_ji }
    - With both sides known it computes z_ij and u_ij locally.
    - The Coordinator's only job is to route messages (copy outbox
→ inbox)
      and collect residuals; it performs no ADMM arithmetic.

ADMM variables owned by this agent:
    x_i    -- local patch estimate
    z_ij   -- consensus value on the overlap with neighbour j
              (computed here, same result at both ends by
symmetry)
    u_ij   -- scaled dual variable for edge (i, j)
"""

import numpy as np
from concurrent.futures import ProcessPoolExecutor

from .prox_tv import proximal_tv_l2
from .weight_map import structure_tensor_weights

def _solve_patch(args):
    """Standalone function for parallel x-update.

    Receives a plain-data snapshot of agent state so it is fully
    picklable. Returns (agent_id, x_new, p_new).
    """
    (agent_id, noisy, x_init, p_init,

```



```

        # Warm-start dual variable for the inner Chambolle-Pock
solver.
        self._p_dual = np.zeros((2,) + noisy_patch.shape)

        # Last computed TV weight map (stored so Coordinator can
reconstruct
        # a full-image map for visualisation).
        self._tv_weight_map = tv_weight

        # neighbors[neighbor_id] = {
        #   'local_slice' : (slice_r, slice_c) into THIS agent's
patch
        #   'z'           : consensus value on the shared overlap
        #   'u'           : this agent's scaled dual for edge
(i, j)
        #   'outbox'      : message this agent will send after
x-update
        #   'inbox'      : message received from neighbour
before z/u-update
        # }
        self.neighbors = {}

        # Graph wiring (called by Coordinator during setup only)

        def add_neighbor(self, neighbor_id, local_slice,
overlap_shape):
            self.neighbors[neighbor_id] = {
                'local_slice': local_slice,
                'z': np.zeros(overlap_shape),
                'u': np.zeros(overlap_shape),
                'outbox': None,
                'inbox': None,
            }

        # Helpers

        def _compute_tv_weight_map(self):
            """Per-pixel TV weight map recomputed from the current
estimate."""
            if not self._use_weight_map:
                return self.tv_weight
            raw = structure_tensor_weights(self.x,
**self._wm_params)
            return self.tv_weight * raw

        def _build_solver_args(self, rho):
            """Assemble the plain-data tuple needed by
_solve_patch."""
            self._tv_weight_map = self._compute_tv_weight_map()

            consensus_weight_map = np.zeros_like(self.x)
            consensus_sum = np.zeros_like(self.x)

```

```

        for nb in self.neighbors.values():
            sr, sc = nb['local_slice']
            target = nb['z'] - nb['u']
            consensus_weight_map[sr, sc] += rho
            consensus_sum[sr, sc] += rho * target

        target_map = np.zeros_like(self.x)
        mask = consensus_weight_map > 0
        target_map[mask] = consensus_sum[mask] /
        consensus_weight_map[mask]

        return (
            self.id,
            self.noisy,
            self.x,
            self._p_dual,
            consensus_weight_map,
            target_map,
            self._tv_weight_map,
            self.n_inner_iter,
        )

    # x-update (sequential path - used when n_workers == 1)

    def x_update(self, rho):
        """Solve the local ADMM subproblem and update self.x in-
        place."""
        args = self._build_solver_args(rho)
        _, x_new, p_new = _solve_patch(args)
        self.x = x_new
        self._p_dual = p_new

    # Message passing

    def fill_outbox(self):
        """After x-update: write current overlap values and
        duals to outbox.
        The Coordinator copies each outbox to the matching
        neighbour's inbox.
        """
        for nid, nb in self.neighbors.items():
            sr, sc = nb['local_slice']
            nb['outbox'] = {
                'x_overlap': self.x[sr, sc].copy(),
                'u': nb['u'].copy(),
            }

    def receive(self, neighbor_id, message):
        """Coordinator delivers neighbour j's outbox message to
        our inbox."""
        self.neighbors[neighbor_id]['inbox'] = message

    # z-update and u-update (agent-local, after inbox is filled)

```

```

def zu_update(self):
    """Compute z_ij and u_ij for every edge using only local
state
    and the message received from each neighbour.

    Returns (primal_res_sq, dual_res_sq) summed over all
edges,
    for convergence monitoring.
    """
    primal_sq = 0.0
    dual_sq = 0.0

    for nid, nb in self.neighbors.items():
        msg = nb['inbox']
        assert msg is not None, f"Agent {self.id}: inbox
from {nid} is empty"

        xi = self.x[nb['local_slice']]
        xj = msg['x_overlap']
        ui = nb['u']
        uj = msg['u']
        z_old = nb['z']

        # z-update: closed-form minimiser of
        # (rho/2)||xi - z + ui||^2 + (rho/2)||xj - z +
uj||^2
        z_new = 0.5 * ((xi + ui) + (xj + uj))
        ui_new = ui + (xi - z_new)

        primal_sq += np.sum((xi - z_new) ** 2)
        dual_sq += np.sum((z_new - z_old) ** 2)

        nb['z'] = z_new
        nb['u'] = ui_new
        nb['inbox'] = None # consume message

    return primal_sq, dual_sq

# Accessor for Coordinator reconstruction
def get_overlap_value(self, neighbor_id):
    sr, sc = self.neighbors[neighbor_id]['local_slice']
    return self.x[sr, sc]

```

A.2 Клас Coordinator

Лістинг A.2 – Код розробленого класу Coordinator

```

"""
Coordinator for MAS consensus-ADMM image denoising.

```

Responsibilities:

- Build agents and wire the neighbour graph (setup only).
- Each round: dispatch x-updates (optionally parallel), route messages between agents, collect residuals, check convergence.
- Reconstruct the full image and weight map from agent estimates.

The Coordinator performs NO ADMM arithmetic. All x-, z-, and u-updates are computed by the agents themselves. The Coordinator's message-routing role is the only coupling between agents at runtime.

Parallelism:

```

    n_workers=1    Sequential loop (default, good for debugging
and small images).
    n_workers>1    x-updates run in a ProcessPoolExecutor across
`n_workers`
                    processes. Each worker receives a plain-data
snapshot of         the agent state (fully picklable), solves the
subproblem,         and returns (agent_id, x_new, p_new). z/u-
updates and         message routing remain sequential in the main
process             (they are cheap  $O(|E|*k)$  operations).
"""

```

```

import numpy as np
from concurrent.futures import ProcessPoolExecutor

from .partition import build_patch_grid, bbox_intersection,
global_to_local
from .agent import Agent, _solve_patch

```

class Coordinator:

```

    def __init__(self, noisy_image, patch_size, stride,
                  tv_weight=0.1, rho=1.0, n_inner_iter=30,
                  use_weight_map=False,
                  wm_sigma=2.0, wm_alpha=2.5, wm_min=0.2,
wm_max=0.7,
                  n_workers=1):
        """
        Parameters
        -----
        noisy_image : 2D np.ndarray
        patch_size : (ph, pw)
        stride : (sh, sw)
        tv_weight : float

```

```

        Global TV regularisation strength (lambda).
    rho : float
        ADMM penalty parameter, shared by all edges.
    n_inner_iter : int
        Chambolle-Pock iterations per agent per ADMM
round.
    use_weight_map : bool
        Each agent uses a per-pixel adaptive TV weight map
if True.
    wm_sigma, wm_alpha, wm_min, wm_max : float
        Weight-map hyperparameters forwarded to each
agent.
    n_workers : int
        Number of parallel worker processes for the x-
update.
        1 = sequential (default). Set to os.cpu_count()
for full
        parallelism; a value of 2-4 is usually sufficient.
    """
    self.image_shape = noisy_image.shape
    self.patch_size = patch_size
    self.stride = stride
    self.rho = rho
    self.n_inner_iter = n_inner_iter
    self.use_weight_map = use_weight_map
    self.n_workers = n_workers
    self.wm_kwargs = dict(wm_sigma=wm_sigma,
wm_alpha=wm_alpha,
                        wm_min=wm_min, wm_max=wm_max)

    self.agents = []
    self.edges = []
    self.history = {'primal_residual': [],
'dual_residual': []}

    self._build(noisy_image, tv_weight)

def _build(self, noisy_image, tv_weight):
    bboxes, _, grid_shape = build_patch_grid(
        self.image_shape, self.patch_size, self.stride
    )
    self.grid_shape = grid_shape
    self.bboxes = bboxes

    if np.isscalar(tv_weight):
        tv_weights = np.full(len(bboxes),
float(tv_weight))
    else:
        tv_weights = np.asarray(tv_weight, dtype=float)
        if tv_weights.shape != (len(bboxes),):
            raise ValueError(
                f"tv_weight array must have shape
({len(bboxes)}), "

```

```

        f"got {tv_weights.shape}"
    )

    for pid, bbox in enumerate(bboxes):
        r0, c0, r1, c1 = bbox
        self.agents.append(Agent(
            pid, bbox, noisy_image[r0:r1, c0:c1],
            tv_weight=tv_weights[pid],
            n_inner_iter=self.n_inner_iter,
            use_weight_map=self.use_weight_map,
            **self.wm_kwargs,
        ))

    for i in range(len(bboxes)):
        for j in range(i + 1, len(bboxes)):
            ov = bbox_intersection(bboxes[i], bboxes[j])
            if ov is None:
                continue
            li = global_to_local(ov, bboxes[i])
            lj = global_to_local(ov, bboxes[j])
            shape = (li[2] - li[0], li[3] - li[1])
            self.agents[i].add_neighbor(j, (slice(li[0],
            li[2]), slice(li[1], li[3])), shape)
            self.agents[j].add_neighbor(i, (slice(lj[0],
            lj[2]), slice(lj[1], lj[3])), shape)
            self.edges.append((i, j))

    def step(self):
        """One ADMM round:
        1. x-update - all agents solve their local
subproblem
                                (parallel if n_workers > 1).
        2. Routing - Coordinator delivers each agent's
outbox to
                                each neighbour's inbox (pure message
passing,
                                no arithmetic).
        3. z/u-update - each agent updates its own z and u
variables
                                using only the received message.
        """

        # 1. x-update
        if self.n_workers > 1:
            args_list = [a._build_solver_args(self.rho) for a
            in self.agents]
            with
            ProcessPoolExecutor(max_workers=self.n_workers) as pool:
                results = list(pool.map(_solve_patch,
            args_list))
            # write results back into agents (main process)
            for agent_id, x_new, p_new in results:

```

```

        self.agents[agent_id].x          = x_new
        self.agents[agent_id]._p_dual = p_new
        # recompute and cache the weight map (was
computed inside
        # _build_solver_args which ran in the worker
process)
        self.agents[agent_id]._tv_weight_map = \

self.agents[agent_id]._compute_tv_weight_map()
    else:
        for agent in self.agents:
            agent.x_update(self.rho)

    # 2. Message routing (Coordinator's only arithmetic-
free role)
    # Each agent writes its overlap values + duals to its
outbox;
    # Coordinator copies each outbox to the matching
neighbour's inbox.
    for agent in self.agents:
        agent.fill_outbox()

    for (i, j) in self.edges:
        self.agents[j].receive(i,
self.agents[i].neighbors[j]['outbox'])
        self.agents[i].receive(j,
self.agents[j].neighbors[i]['outbox'])

    # 3. z/u-update (agent-local)
    primal_sq = 0.0
    dual_sq   = 0.0
    for agent in self.agents:
        ps, ds = agent.zu_update()
        primal_sq += ps
        dual_sq   += ds

    # Residuals are summed over agents (each edge counted
once per side).
    # Divide dual_sq by 2 to avoid double-counting (both
agents observe
    # the same  $z_{old} \rightarrow z_{new}$  transition).
    primal_res = float(np.sqrt(primal_sq))
    dual_res   = float(np.sqrt(dual_sq / 2.0))
    self.history['primal_residual'].append(primal_res)
    self.history['dual_residual'].append(dual_res)
    return primal_res, dual_res

def run(self, n_iters, tol=1e-4, verbose=False):
    """Run up to n_iters synchronous rounds."""
    for it in range(n_iters):
        pr, dr = self.step()
        if verbose:

```

```

        print(f"iter {it + 1:3d}:  primal_res={pr:.6f}
dual_res={dr:.6f}")
        if pr < tol and dr < tol:
            if verbose:
                print(f"Converged after {it + 1}
iterations.")
            break
        return self.history

    def reconstruct(self):
        """Overlap-average all agent estimates into a full
image."""
        H, W = self.image_shape
        acc  = np.zeros((H, W))
        count = np.zeros((H, W))
        for agent in self.agents:
            r0, c0, r1, c1 = agent.bbox
            acc[r0:r1, c0:c1] += agent.x
            count[r0:r1, c0:c1] += 1.0
        return acc / count

    def reconstruct_weight_map(self):
        """Overlap-average each agent's TV weight map into a
full image.
Returns None when use_weight_map is False."""
        if not self.use_weight_map:
            return None
        H, W = self.image_shape
        acc  = np.zeros((H, W))
        count = np.zeros((H, W))
        for agent in self.agents:
            r0, c0, r1, c1 = agent.bbox
            wm = agent._tv_weight_map
            if np.isscalar(wm):
                wm = np.full((r1 - r0, c1 - c0), wm)
            acc[r0:r1, c0:c1] += wm
            count[r0:r1, c0:c1] += 1.0
        return acc / count

```

A.3 Файл prox_tv.py

Лістинг А.3 – Код розробленого файлу prox_tv.py

```

"""
Proximal solver for problems of the form

    min_x  (1/2) * sum_p (x_p - y_p)^2
           + sum_p lambda_p * |grad x|_p          ← spatially-
varying TV
           + sum_p (c_p / 2) * (x_p - t_p)^2      ← ADMM
consensus term

```

where:

- λ_p is a per-pixel TV weight (the structure-tensor weight map),
controlling how strongly TV regularisation is applied at each pixel.
- (c_p, t_p) are the ADMM consensus penalty weight and target ($c_p = \rho * \text{number-of-neighbors-covering-pixel-p}$, $t_p = \text{their z-u average}$).

This is solved with the Chambolle-Pock primal-dual algorithm:

$$F(Kx) = \sum_p \lambda_p * ||[Kx]_p||_2 \quad (K = \text{discrete gradient})$$

$$G(x) = (1/2) ||x - y||^2 + \sum_p (c_p/2) (x_p - t_p)^2$$

Iterations:

$$p^{k+1} = \text{proj}_{\{\text{radius_map}\}}(p^k + \sigma * K(x_{\text{bar}}^k))$$

$$x^{k+1} = \text{prox}_{\{\tau G\}}(x^k + \tau * \text{div}(p^{k+1}))$$

$$x_{\text{bar}}^{k+1} = x^{k+1} + \theta * (x^{k+1} - x^k)$$

$\text{proj}_{\{\text{radius_map}\}}$ projects each 2-vector $p[:,i,j]$ onto the l_2 ball of

$\text{radius_map}[i,j] = \lambda_p[i,j]$, allowing the TV strength to vary

per pixel. $\text{prox}_{\{\tau G\}}$ is closed-form because G is a separable quadratic.

"""

```
import numpy as np
```

```
def gradient(x):
```

```
    """Forward-difference discrete gradient with Neumann
    boundary
    conditions. Returns an array of shape (2, H, W): [d/dx,
    d/dy]."""
```

```
    gx = np.zeros_like(x)
    gy = np.zeros_like(x)
    gx[:, :-1] = x[:, 1:] - x[:, :-1]
    gy[:-1, :] = x[1:, :] - x[:-1, :]
    return np.stack([gx, gy])
```

```
def divergence(p):
```

```
    """Backward-difference divergence; adjoint of -gradient
    (i.e.
```

```
    <grad(x), p> = -<x, div(p)>)."""
    px, py = p[0], p[1]
    dx = np.zeros_like(px)
    dy = np.zeros_like(py)
```

```

dx[:, 1:] = px[:, 1:] - px[:, :-1]
dx[:, 0] = px[:, 0]
dy[1:, :] = py[1:, :] - py[:, :-1]
dy[0, :] = py[0, :]
return dx + dy

def _project_l2_ball(p, radius_map):
    """Pointwise projection of each 2-vector p[:, i, j] onto
    the l2 ball
    of radius radius_map[i, j].

    radius_map may be a scalar (uniform TV, original
    behaviour) or a 2D
    array of shape (H, W) for spatially-varying TV strength.
    """
    norm = np.sqrt(p[0] ** 2 + p[1] ** 2)
    scale = np.maximum(norm / np.maximum(radius_map, 1e-12),
1.0)
    return p / scale[np.newaxis, :, :]

def proximal_tv_l2(
    y,
    consensus_weight_map,
    target_map,
    tv_weight_map,
    n_iter=50,
    x_init=None,
    p_init=None,
    tau=None,
    sigma=None,
    theta=1.0,
):
    """Solve the per-agent ADMM subproblem with spatially-
    varying TV.

    min_x (1/2)||x-y||^2
          + sum_p tv_weight_map[p] * |grad x|_p
          + sum_p (consensus_weight_map[p]/2) * (x[p] -
target_map[p])^2

    Parameters
    -----
    y : np.ndarray, shape (H, W)
        Noisy observation (data fidelity target).
    consensus_weight_map : np.ndarray, shape (H, W)
        c_p: ADMM penalty weight per pixel. Zero outside
overlap regions.
    target_map : np.ndarray, shape (H, W)
        t_p: consensus target per pixel. Ignored where
consensus_weight_map==0.
    tv_weight_map : np.ndarray or float

```

```

        lambda_p: per-pixel TV regularisation strength. Pass a
        scalar for
            uniform TV (backwards-compatible), or a 2D array for
            spatially-
                varying TV (structure-tensor weighted).
        n_iter : int
            Number of Chambolle-Pock iterations.
        x_init, p_init : optional warm-start primal/dual
        variables.
        tau, sigma : optional step sizes (default:
        tau=sigma=1/sqrt(8),
            ensuring  $\tau \cdot \sigma \cdot \|K\|^2 \leq 1$ ).
        theta : float
            Extrapolation parameter (1.0 = standard PDHG).

    Returns
    -----
    x : np.ndarray, the solution.
    p : np.ndarray, the final dual variable (for warm-starting
    next call).
    """
    if tau is None or sigma is None:
        L2 = 8.0
        tau = sigma = 1.0 / np.sqrt(L2)

    x = y.copy() if x_init is None else x_init.copy()
    x_bar = x.copy()
    p = np.zeros((2,) + y.shape) if p_init is None else
    p_init.copy()

    # radius_map for the dual projection: lambda_p (scalar or
    2D)
    radius_map = tv_weight_map # may be scalar or array;
    _project handles both

    # Precompute prox_G denominator and linear term (both
    pixel-wise)
    denom = 1.0 + consensus_weight_map
    linear = y + consensus_weight_map * target_map

    for _ in range(n_iter):
        x_old = x

        # dual update: project onto per-pixel l2 ball of
        radius lambda_p
        p = _project_l2_ball(p + sigma * gradient(x_bar),
        radius_map)

        # primal update: closed-form prox of the separable
        quadratic G
        v = x + tau * divergence(p)
        x = (v + tau * linear) / (1.0 + tau * denom)

```

```

# over-relaxation / extrapolation
x_bar = x + theta * (x - x_old)

return x, p

```

A.4 Файл weight_map.py

Лістинг A.4 – Код розробленого файлу prox_tv.py

```

"""
Classifier-free per-pixel TV weight map based on local
variance.

    High weight    →    smooth more    (low local variance:
flat/uniform regions)
    Low weight     →    preserve        (high local variance: edges,
texture)

    The weight map is computed per-patch from the agent's own
noisy
    observation, which is why it differs slightly across
overlapping patches –
    that difference is exactly what the consensus mechanism
resolves.
"""

import numpy as np
from scipy.ndimage import gaussian_filter

def structure_tensor_weights(img, sigma=1.0, alpha=25,
w_min=0.4, w_max=0.7):
    """Per-pixel TV weight map via local variance of img.

    Parameters
    -----
    img : 2D np.ndarray
        The patch image (agent's local observation).
    sigma : float
        Gaussian scale for measuring local variance.
    alpha : float
        Sensitivity to variance: larger → weight drops
faster near edges.
    w_min, w_max : float
        Output range after linear normalisation.

    Returns
    -----
    w : 2D np.ndarray, same shape as img, values in [w_min,
w_max].
    """
    mean = gaussian_filter(img, sigma)

```

```

mean_sq = gaussian_filter(img ** 2, sigma)
local_var = np.clip(mean_sq - mean ** 2, 0, None)

raw = 1.0 / (1.0 + alpha * local_var)
lo, hi = raw.min(), raw.max()
w = w_min + (w_max - w_min) * (raw - lo) / (hi - lo +
1e-12)

return w

```

A.5 Файл partition.py

Лістинг A.5 – Код розробленого файлу partition.py

```

"""
Utilities for splitting an image into a regular grid of
overlapping
patches, and for computing the geometry of the overlap
regions between
neighboring patches.
"""

import numpy as np

def compute_patch_origins(length, patch_size, stride):
    """Return a list of starting indices (origins) along
one axis such
that every patch has size `patch_size`, consecutive
patches are
`stride` apart, and the final patch is shifted so it
ends exactly
at `length` (i.e. the image is fully covered, with the
last patch
possibly overlapping its predecessor by more than
`stride`).
"""
    if patch_size >= length:
        return [0]

    origins = list(range(0, length - patch_size + 1,
stride))
    if origins[-1] != length - patch_size:
        origins.append(length - patch_size)
    return origins

def build_patch_grid(image_shape, patch_size, stride):
    """Build the list of patch bounding boxes for an image.

Parameters
-----

```

```

    image_shape : (H, W)
    patch_size : (ph, pw)
    stride : (sh, sw)

    Returns
    -----
    bboxes : list of (r0, c0, r1, c1) in global image
coordinates
    grid_index : dict mapping (row_idx, col_idx) -> patch
id
    grid_shape : (n_rows, n_cols)
    """
    H, W = image_shape
    ph, pw = patch_size
    sh, sw = stride

    row_origins = compute_patch_origins(H, ph, sh)
    col_origins = compute_patch_origins(W, pw, sw)

    bboxes = []
    grid_index = {}
    pid = 0
    for r_idx, r0 in enumerate(row_origins):
        for c_idx, c0 in enumerate(col_origins):
            bbox = (r0, c0, r0 + ph, c0 + pw)
            bboxes.append(bbox)
            grid_index[(r_idx, c_idx)] = pid
            pid += 1

    grid_shape = (len(row_origins), len(col_origins))
    return bboxes, grid_index, grid_shape

def bbox_intersection(b1, b2):
    """Intersection of two (r0, c0, r1, c1) bounding boxes,
or None if
    they don't overlap (touching edges with zero area don't
count)."""
    r0 = max(b1[0], b2[0])
    c0 = max(b1[1], b2[1])
    r1 = min(b1[2], b2[2])
    c1 = min(b1[3], b2[3])
    if r0 < r1 and c0 < c1:
        return (r0, c0, r1, c1)
    return None

def global_to_local(bbox, patch_bbox):
    """Express a global bounding box `bbox` in coordinates
local to
    `patch_bbox` (i.e. relative to the patch's top-left
corner)."""
    r0, c0, r1, c1 = bbox

```

```
pr0, pc0, _, _ = patch_bbox
return (r0 - pr0, c0 - pc0, r1 - pr0, c1 - pc0)
```

A.6 Файл `__init__.py`

Лістинг A.6 – Код розробленого файлу `__init__.py`

```
from .partition import build_patch_grid, bbox_intersection,
global_to_local
from .agent import Agent
from .coordinator import Coordinator
from .prox_tv import proximal_tv_l2
from .weight_map import structure_tensor_weights
from .visualise import plot_agent_graph

__all__ = [
    "build_patch_grid",
    "bbox_intersection",
    "global_to_local",
    "Agent",
    "Coordinator",
    "proximal_tv_l2",
    "structure_tensor_weights",
    "plot_agent_graph",
]
```

A.7 Файл `demo.py`

Лістинг A.7 – Код розробленого файлу `demo.py`

```
"""
Demo: compare three denoising modes
1. Plain whole-image TV (skimage baseline)
2. MAS consensus-ADMM with uniform TV
3. MAS consensus-ADMM with per-patch structure-tensor weight
map

Also renders the agent communication graph and the weight map
from a
representative central agent.
"""

import numpy as np
from skimage import data, img_as_float
from skimage.restoration import denoise_tv_chambolle
from skimage.metrics import peak_signal_noise_ratio as psnr
from skimage.metrics import structural_similarity as ssim
import matplotlib.pyplot as plt
import os
```

```

from mas_denoising import Coordinator,
structure_tensor_weights, plot_agent_graph

INNER_ITER = 20
OUTER_ITER = 20
PATCH_SIZE = (64, 64)
STRIDE = (32, 32)

def run_mas(noisy, use_weight_map, tv_weight, rho, sigma, alpha,
w_min, w_max):
    coord = Coordinator(
        noisy,
        patch_size=PATCH_SIZE,
        stride=STRIDE,
        tv_weight=tv_weight,
        rho=rho,
        n_inner_iter=INNER_ITER,
        use_weight_map=use_weight_map,
        wm_sigma=sigma,
        wm_alpha=alpha,
        wm_min=w_min,
        wm_max=w_max,
        n_workers=os.cpu_count()
    )
    coord.run(n_iters=OUTER_ITER, tol=1e-6, verbose=True)
    return coord

def add_gaussian_noise(img, sigma, seed):
    rng = np.random.default_rng(seed)
    noisy = img.copy()
    noisy = np.clip(noisy + rng.normal(0, sigma, noisy.shape),
0, 1)
    return noisy

def add_sp_noise(img, p, seed):
    rng = np.random.default_rng(seed)
    noisy = img.copy()

    mask_rand = rng.random(noisy.shape)

    noise_mask = mask_rand < p

    noise_mask_broadcast = noise_mask

    noisy[mask_rand < p / 2] = 0.0
    noisy[(mask_rand >= p / 2) & (mask_rand < p)] = 1.0

    return noisy, noise_mask_broadcast

def main():

```

```

image = img_as_float(data.camera())
seed = 42
noisy = add_gaussian_noise(image, 0.1, seed)

plain_tv = denoise_tv_chambolle(noisy, weight=0.12)

use_weight_map=False
tv_weight      = 0.1
rho            = 1
sigma          = 1.0
alpha          = 20.0
w_min          = 0.4
w_max          = 0.7

print("Running MAS (uniform TV)...")
coord_uniform = run_mas(noisy, False, tv_weight, rho, sigma,
alpha, w_min, w_max)
result_uniform = coord_uniform.reconstruct()

use_weight_map=True
tv_weight      = 0.12
rho            = 1
sigma          = 1.0
alpha          = 20.0
w_min          = 0.4
w_max          = 0.7
print("Running MAS (weight-map TV)...")
coord_wm = run_mas(noisy, use_weight_map, tv_weight, rho,
sigma, alpha, w_min, w_max)
result_wm = coord_wm.reconstruct()

print()
print(f"PSNR noisy                      : {psnr(image, noisy):.2f}
dB")
print(f"PSNR plain TV                    : {psnr(image, plain_tv):.2f}
dB")
print(f"PSNR MAS uniform TV                : {psnr(image,
result_uniform):.2f} dB")
print(f"PSNR MAS weight-map TV           : {psnr(image,
result_wm):.2f} dB")

print()
print(f"SSIM noisy                          : {ssim(image, noisy,
data_range=1.0):.2f}")
print(f"SSIM plain TV                      : {ssim(image, plain_tv,
data_range=1.0):.2f}")
print(f"SSIM MAS uniform TV                : {ssim(image,
result_uniform, data_range=1.0):.2f}")
print(f"SSIM MAS weight-map TV           : {ssim(image, result_wm,
data_range=1.0):.2f}")

```

```

# --- Figure 1: denoising comparison ---
fig, axes = plt.subplots(1, 5, figsize=(20, 6))
for ax, im, title in zip(
    axes,
    [image, noisy, plain_tv, result_uniform, result_wm],
    ["Original", "Noisy", "Plain TV",
     "MAS uniform TV", "MAS weight-map TV"],
):
    ax.imshow(im, cmap="gray", vmin=0, vmax=1)
    ax.set_title(title, fontsize=9)
    ax.axis("off")
plt.tight_layout()
plt.show()
#plt.savefig("result.png", dpi=150)
print("\nSaved result.png")

# --- Figure 2: convergence (both runs) ---
fig2, ax2 = plt.subplots(figsize=(6, 4))
ax2.plot(coord_uniform.history["primal_residual"],
label="uniform - primal")
ax2.plot(coord_uniform.history["dual_residual"],
label="uniform - dual", linestyle="--")
ax2.plot(coord_wm.history["primal_residual"],
label="weight-map - primal")
ax2.plot(coord_wm.history["dual_residual"],
label="weight-map - dual", linestyle="--")
ax2.set_xlabel("outer iteration")
ax2.set_ylabel("residual norm")
ax2.set_yscale("log")
ax2.legend(fontsize=8)
ax2.set_title("ADMM convergence: uniform vs weight-map TV")
plt.tight_layout()
#plt.show()
plt.savefig("convergence.png", dpi=150)
print("Saved convergence.png")

if __name__ == "__main__":
    main()

```

ДОДАТОК Б

Відомість кваліфікаційної роботи

Позначення					Найменування		Дод. відомості	
					Текстові документи			
1.					Пояснювальна записка		98 с.	