Unraveling Normal Anatomy via Fluid-Driven Anomaly Randomization
(Appendix)

This Appendix provides additional context regarding:
A: Computing Derivatives of Perlin Noise;
B: Datasets and Metrics;
C: Implementation Details.

A. Computing Derivatives of Perlin Noise

Perlin noise is a gradient noise function invented by Ken Perlin [41]. Unlike traditional random noise, which produces entirely chaotic, discontinuous patterns, Perlin noise is smooth and continuous, it generates a field of smoothly varying values that appear random but maintain a continuous flow, without abrupt jumps or visible seams. These properties make Perlin noise ideal for generating natural-looking patterns that have rich textures. In UNA, we resort to Perlin noise for generating random shape profiles for anomaly probability initialization as well as the incompressible flow and non-negative diffusion fields in Sec. 3.

A.1. Perlin Noise and Random Anomaly Initialization

Here, we present the implementation details of our Perlin noise generation and thresholding for random anomaly shape synthesis. As shown in the code below, the generation of the random anomaly probability map can be summarized into six steps:
1. Generate a grid of random gradients at lattice points (Line 25-30).
2. Compute the relative position of the point inside the grid cell (Line 31-44).
3. Calculate the dot product of the gradients and the relative position vectors (45-52).
4. Apply the fade function to smooth the interpolation (Line 54-61).
5. Interpolate between dot products to get a smooth value (Line 62-69).
6. Threshold to get a random shape of anomaly profile (Line 71-78).

```
1  import os, time
2  import numpy as np
3
4  def interpolant(t):
5      return t*t*t*(t*(t*6 - 15) + 10)
6
7  def generate_perlin_noise_3d(shape, res, tileable=(False, False, False), interpolant=interpolant,
       percentile=None,):
8      """Generate a 3D numpy array of perlin noise.
9
10     Args:
11         shape: The shape of the generated array (tuple of three ints). This must be a multiple of res.
12         res: The number of periods of noise to generate along each axis (tuple of three ints). Note
       shape must be a multiple of res.
13         tileable: If the noise should be tileable along each axis (tuple of three bools). Defaults to (
       False, False, False).
14         interpolant: The interpolation function, defaults to t*t*t*(t*(t*6 - 15) + 10).
15         percentile: The percentile for random shape thresholding.
16
17     Returns:
18         A numpy array of shape with the generated noise.
19         (Optional) A numpy array of thresholded noise given an input percentile.
20     """
21     seed = int(time.time())
22     os.environ['PYTHONHASHSEED'] = str(seed)
23     np.random.seed(seed)
24
25     # Initialize the grid
26     delta = (res[0] / shape[0], res[1] / shape[1], res[2] / shape[2])
27     d = (shape[0] // res[0], shape[1] // res[1], shape[2] // res[2])
28     grid = np.mgrid[0:res[0]:delta[0],0:res[1]:delta[1],0:res[2]:delta[2]]
```

```
    grid = np.mgrid[0:res[0]:delta[0],0:res[1]:delta[1],0:res[2]:delta[2]]
    grid = grid.transpose(1, 2, 3, 0) % 1
    # Gradients
    theta = 2*np.pi*np.random.rand(res[0] + 1, res[1] + 1, res[2] + 1)
    phi = 2*np.pi*np.random.rand(res[0] + 1, res[1] + 1, res[2] + 1)
    gradients = np.stack(
        (np.sin(phi)*np.cos(theta), np.sin(phi)*np.sin(theta), np.cos(phi)),
        axis=3
    )
    if tileable[0]:
        gradients[-1,:,:] = gradients[0,:,:]
    if tileable[1]:
        gradients[:,-1,:] = gradients[:,0,:]
    if tileable[2]:
        gradients[:,:,-1] = gradients[:,:,0]
    gradients = gradients.repeat(d[0], 0).repeat(d[1], 1).repeat(d[2], 2)
    g000 = gradients[    :-d[0],    :-d[1],    :-d[2]]
    g100 = gradients[d[0]:     ,    :-d[1],    :-d[2]]
    g010 = gradients[    :-d[0],d[1]:     ,    :-d[2]]
    g110 = gradients[d[0]:     ,d[1]:     ,    :-d[2]]
    g001 = gradients[    :-d[0],    :-d[1],d[2]:     ]
    g101 = gradients[d[0]:     ,    :-d[1],d[2]:     ]
    g011 = gradients[    :-d[0],d[1]:     ,d[2]:     ]
    g111 = gradients[d[0]:     ,d[1]:     ,d[2]:     ]
    # Ramps
    n000 = np.sum(np.stack((grid[:,:,:,0]  , grid[:,:,:,1]  , grid[:,:,:,2]  ), axis=3) * g000, 3)
    n100 = np.sum(np.stack((grid[:,:,:,0]-1, grid[:,:,:,1]  , grid[:,:,:,2]  ), axis=3) * g100, 3)
    n010 = np.sum(np.stack((grid[:,:,:,0]  , grid[:,:,:,1]-1, grid[:,:,:,2]  ), axis=3) * g010, 3)
    n110 = np.sum(np.stack((grid[:,:,:,0]-1, grid[:,:,:,1]-1, grid[:,:,:,2]  ), axis=3) * g110, 3)
    n001 = np.sum(np.stack((grid[:,:,:,0]  , grid[:,:,:,1]  , grid[:,:,:,2]-1), axis=3) * g001, 3)
    n101 = np.sum(np.stack((grid[:,:,:,0]-1, grid[:,:,:,1]  , grid[:,:,:,2]-1), axis=3) * g101, 3)
    n011 = np.sum(np.stack((grid[:,:,:,0]  , grid[:,:,:,1]-1, grid[:,:,:,2]-1), axis=3) * g011, 3)
    n111 = np.sum(np.stack((grid[:,:,:,0]-1, grid[:,:,:,1]-1, grid[:,:,:,2]-1), axis=3) * g111, 3)
    # Interpolation
    t = interpolant(grid)
    n00 = n000*(1-t[:,:,:,0]) + t[:,:,:,0]*n100
    n10 = n010*(1-t[:,:,:,0]) + t[:,:,:,0]*n110
    n01 = n001*(1-t[:,:,:,0]) + t[:,:,:,0]*n101
    n11 = n011*(1-t[:,:,:,0]) + t[:,:,:,0]*n111
    n0 = (1-t[:,:,:,1])*n00 + t[:,:,:,1]*n10
    n1 = (1-t[:,:,:,1])*n01 + t[:,:,:,1]*n11

    noise = ((1-t[:,:,:,2])*n0 + t[:,:,:,2]*n1)
    if percentile is None:
        return noise
    shres = np.percentile(noise, percentile)
    mask = np.zeros_like(noise)
    mask[noise >= shres] = 1.
    noise *= mask
    return noise, mask
```

A.2. Flow and Diffusion Initialization

As discussed in Sec. 3.1, we further utilize Perlin noise for creating the random potentials $\mathbf{\Psi}$ for $\mathbf{V}$, and Φ for D. The random map of L initialization, as a scalar field, could be directly obtained from the above function "generate_perlin_noise_3d". Here, we show details on the implementation of $\mathbf{\Psi}$ initialization, which is a 3-dimensional vector field. As shown in the code below, the generation of the random incompressible flow fields can be summarized into three steps:

1. Generate three individual Perlin noise maps for the potential ($\mathbf{\Psi}$) construction (Line 6-9).
2. Reshape the noise map to match the current subject sample's patch size (Line 11-23).
3. Surjectively map the random potential to its corresponding incompressible flow space (Line 25-28).

```
1 import torch
2
3 def generate_velocity_3d(shape, perlin_res, V_multiplier, device, save_orig_for_visualize = False):
4     pad_shape = [ 200, 200, 200 ]
5
```

```
    # Generate random potentials (back to original shape)
    curl_a = generate_perlin_noise_3d(pad_shape, perlin_res, tileable=(True, False, False))
    curl_b = generate_perlin_noise_3d(pad_shape, perlin_res, tileable=(True, False, False))
    curl_c = generate_perlin_noise_3d(pad_shape, perlin_res, tileable=(True, False, False))

    # Back to original shape
    curl_a = curl_a[(pad_shape[0] - shape[0]) // 2 : (pad_shape[0] - shape[0]) // 2 + shape[0], \
                    (pad_shape[1] - shape[1]) // 2 : (pad_shape[1] - shape[1]) // 2 + shape[1], \
                    (pad_shape[2] - shape[2]) // 2 : (pad_shape[2] - shape[2]) // 2 + shape[2]
                    ]
    curl_b = curl_b[(pad_shape[0] - shape[0]) // 2 : (pad_shape[0] - shape[0]) // 2 + shape[0], \
                    (pad_shape[1] - shape[1]) // 2 : (pad_shape[1] - shape[1]) // 2 + shape[1], \
                    (pad_shape[2] - shape[2]) // 2 : (pad_shape[2] - shape[2]) // 2 + shape[2]
                    ]
    curl_c = curl_c[(pad_shape[0] - shape[0]) // 2 : (pad_shape[0] - shape[0]) // 2 + shape[0], \
                    (pad_shape[1] - shape[1]) // 2 : (pad_shape[1] - shape[1]) // 2 + shape[1], \
                    (pad_shape[2] - shape[2]) // 2 : (pad_shape[2] - shape[2]) // 2 + shape[2]
                    ]

    # Surjective mapping to incompressible flow space
    Vx, Vy, Vz = stream_3D(torch.from_numpy(curl_a).to(device),
                           torch.from_numpy(curl_b).to(device),
                           torch.from_numpy(curl_c).to(device))

    return {'Vx': (Vx * V_multiplier), 'Vy': (Vy * V_multiplier).to(device), 'Vz': (Vz * V_multiplier)}
```

B. Datasets and Metrics

B.1. Datasets and Preprocessing

We test and compare UNA over various datasets including modalities of MR and CT, the MR images further contain T1-weighted, T2-weighted, and FLAIR (fluid-attenuated inversion recovery) images.

- ADNI [25]: we use T1-weighted (2045 cases) MRI scans from the Alzheimer's Disease Neuroimaging Initiative (ADNI). All scans are acquired at 1 mm isotropic resolution from a wide array of scanners and protocols. The dataset contains aging subjects, some diagnosed with mild cognitive impairment (MCI) or Alzheimer's Disease (AD). Many subjects present strong atrophy patterns and white matter lesions.
- HCP [12]: we use T1-weighted (897 cases) and T2-weighted (897 cases) MRI scans of young subjects from the Human Connectome Project, acquired at 0.7 mm resolution.
- ADNI3 [55]: we use T1-weighted (331 cases) and FLAIR (331 cases) MRI scans from ADNI3, which continues the previously funded ADNI1, ADNI-GO, and ADNI2 studies to determine the relationships between the clinical, cognitive, imaging, genetic and biochemical biomarker characteristics of the entire spectrum of sporadic late-onset AD.
- ADHD200 [7]: we use T1-weighted (961 cases) MRI scans from ADH200 Sample, which is a grassroots initiative dedicated to the understanding of the neural basis of Attention Deficit Hyperactivity Disorder (ADHD).
- AIBL [15]: we use T1-weighted (668 cases), T2-weighted (302 cases) and FLAIR (336 cases) MRI scans from The Australian Imaging, Biomarkers and Lifestyle (AIBL) Study, which is a study of cognitive impairment (MCI) and Alzheimer's disease dementia.
- OASIS3 [27]: we use CT (885 cases) scans from OASIS3, which is a longitudinal neuroimaging, clinical, and cognitive dataset for normal aging and AD. For our experiments, we use CT and T1-weighted MRI pair with the earliest date, from each subject.
- ATLAS [31]: we use T1-weighted (655 cases) MRI scans and the provided gold-standard stroke lesion segmentations, from Anatomical Tracings of Lesions After Stroke (ATLAS), which is a study of subacute/chronic stroke.
- ISLES [19] we use FLAIR (152 cases) MRI scans and the provided gold-standard stroke lesion segmentation, from ISLES 2022, which is a MICCAI challenge in 2022 for acute/subacute stroke lesion detection and segmentation.

Among the above eight datasets, ADNI [25], ADNI3 [55], HCP [12], ADHD200 [7], AIBL [15], OASIS3 [27] contain subjects with healthy anatomy. ATLAS [31], ISLES [19]. ATLAS and ISLES include stroke patients, with gold-standard manual segmentations of stroke lesions provided in both datasets.

For all datasets, we skull-strip all the images using SynthStrip [21], and resample them to 1 mm isotropic resolution. For all the modalities other than T1-weighted MRI, we use NiftyReg [39] rigid registration to register all images to their

Category	**Param**	**Corruption Level**		
		Mild	Medium	Severe
Deformation	affine-rotation$_{max}$	15	=	=
	affine-shearing$_{max}$	0.2	=	=
	affine-scaling$_{max}$	0.2	=	=
	nonlinear-scale μ_{min}	0.03	=	=
	nonlinear-scale μ_{max}	0.06	=	=
	nonlinear-scale σ_{max}	4	=	=
Resolution	$p_{\text{low-field}}$	0.1	0.3	0.5
	$p_{\text{anisotropic}}$	0	0.1	0.25
Bias Field	μ_{min}	0.01	0.02	0.02
	μ_{max}	0.02	0.03	0.04
	σ_{min}	0.01	0.05	0.1
	σ_{max}	0.05	0.3	0.6
Noises	σ_{min}	0.01	0.5	5
	σ_{max}	1	5	15

Table B.1. `UNA` synthetic generator setups: mild, medium, and severe levels. p denotes probability, μ and σ refer to the mean and variance of the Gaussian distributions, respectively.

same-subject T1-weighted MRI counterparts. The brain segmentation label maps are obtained by performing SynthSeg [5] on the T1-weighted MR images of all the subjects.

To ease the computation burden during training, the deformations between gold-standard pathology segmentation maps and all healthy training subjects are pre-computed via NiftyReg [39] during data pre-processing. During training, the randomly selected anomaly profiles for `UNA`'s anomaly randomization initialization are registered to the current training subject on the fly, using the pre-computed deformation fields.

B.2. Contrast Synthesis and Contralateral-Paired Input

`UNA`'s synthetic generator uses brain segmentation labels from FreeSurfer [13], for random-modality generation. In this work, we use the segmentation maps of training subjects from all the healthy datasets (`ADNI` [25], `ADNI3` [55], `HCP` [12], `ADHD200` [7], `AIBL` [15], `OASIS3` [27]). We follow Brain-ID [33]'s mild-to-severe data corruption strategy for enriching the training sample variations in resolution, orientation, and external artifacts. In Tab. B.1, we list the generator parameters for mild, medium, and severe data corruption levels, respectively. Note that for each level, the setup parameters control the corruption value ranges, since the simulation is randomized, there could still be mildly corrupted samples generated under the "severe" settings. In addition, the random deformation fields are independent of data corruption levels.

To ease the burden of computing the deformation between original and hemisphere-flipped images for our contralateral-pair input, we preprocess the correspondence from the flipped to the original image for each subject during pre-processing. Specifically, to reduce the effects of the pathological regions for registration, we first use SynthSR [23] to estimate the T1-weighted counterpart of both the hemisphere-flipped image and the original image. Then, we use NiftyReg [39] to compute the deformation fields from the hemisphere-flipped image to the original image. During training, the flipped sample is first registered to the domain of the original image using pre-computed deformations, then the contra-lateral paired inputs undergo the same deformation augmentations simultaneously.

B.3. Metrics

We resort to various metrics for evaluating individual tasks across multiple aspects:

- `L1`: the average $L1$ distance, it is used for the voxel-wise prediction correctness of anatomy reconstruction.
- `PSNR`: the peak signal-to-noise ratio (PSNR) that indicates the fidelity of predictions. It is used in anatomy reconstruction.
- `SSIM`: the structural similarity scores between the generated and real images. It is used in anatomy reconstruction evaluation,

Method	`L1` (↓)			`PSNR` (↑)			`SSIM` (↑)		
	F	H	D	F	H	D	F	H	D
$\lambda_p = 0$	0.0165	0.0153	0.0004	28.62	29.94	42.03	0.959	0.970	0.982
$\lambda_p = 0.5$	0.0150	0.0147	0.0004	31.01	32.85	44.20	0.973	0.989	0.995
`UNA` ($\lambda_p = 1$)	**0.0147**	**0.0143**	**0.0003**	**31.98**	**33.25**	45.61	**0.981**	**0.992**	**0.998**
$\lambda_p = 1.5$	0.0149	0.0150	**0.0003**	30.61	31.27	45.73	0.979	0.986	**0.998**
$\lambda_p = 2$	0.0152	0.0152	**0.0003**	30.29	32.43	**45.78**	0.973	0.989	0.995
$\lambda_{\text{contrast}} = 0$	0.0195	0.0182	0.0005	27.13	28.04	42.97	0.931	0.950	0.969
$\lambda_{\text{contrast}} = 1$	0.0158	0.0163	0.0004	30.78	31.82	44.05	0.953	0.961	0.981
`UNA` ($\lambda_{\text{contrast}} = 2$)	**0.0147**	**0.0143**	0.0003	**31.98**	**33.25**	45.61	**0.981**	**0.992**	**0.998**
$\lambda_{\text{contrast}} = 3$	0.0150	0.0155	**0.0002**	31.82	32.59	**45.63**	0.974	0.984	0.996
$\lambda_{\text{contrast}} = 4$	0.0154	0.0156	0.0003	31.76	32.40	**45.63**	0.970	0.981	0.996

Table C.2. Hyperparameter search of `UNA`. Testing images are real T1w MRI encoded with simulated pathology (same as the first-row group in Tab. 1). (F: full brain; H: healthy region; D: diseased region.)

- `Dice`: the similarity score between predicted and ground truth segmentations, and it is used in anomaly detection evaluation.

C. Implementation Details and Additional Experiments

As a general learning framework, `UNA` can use any backbone to extract brain features. For fair comparisons, we adopt the same 3D UNet [43] as utilized in themodels [23, 33, 34] we compare with, with 64 feature channels in the last layer. A linear regression layer is added following the feature outputs for anatomy reconstruction.

`UNA` is trained on the combination of synthetic and real data, with a probability of 50% and 50%, respectively. The training sample images are sized at 160^3, with a batch size of 4. We use the AdamW optimizer, beginning with a learning rate of 10^{-4} for the first 300,000 iterations, which is then reduced to 10^{-5} for the subsequent 100,000 iterations. The entire training process took approximately 14 days on an NVIDIA A100 GPU.

The additional attention parameter (λ_p in Eq. 7) is set to 1 for healthy anatomy reconstruction in pathological regions. The intra-subject contrastive learning weight ($\lambda_{\text{contrast}}$ in Eq. 9) is set to 2. Tab. C.2 provide additional experiments on hyperparameter search of the anomaly attention weight (λ_p) and the intra-subject contrastive learning weight ($\lambda_{\text{contrast}}$). Specifically, we observe that greater attention to anomalies helps improve the reconstruction of pathology regions, yet it harms the overall performance of originally healthy tissues.