Beyond Blur: A Fluid Perspective on Generative Diffusion Models Supplementary Material

Grzegorz Gruszczynski[1,2] Jakub Meixner[1,3] Michal Wlodarczyk[1,4]
Przemyslaw Musialski[1,5,6]

[1]IDEAS NCBR [2]Samsung AI Center Warsaw [3]Polish Academy of Sciences
[4]Warsaw University of Technology [5]IDEAS Research Institute [6]New Jersey Institute of Technology
g.gruszczyns@samsung.com, {kubameixner, mwlodarzc}@gmail.com, przem@njit.edu

Appendix

In this appendix, §A contains hyperparameters and experiment setup. §B contains additional samples and interpolations for FFHQ-128, MNIST and LSUN Church datasets. §C contains solver implementation details.

A. Hyperparameter settings

We present the complete experimental configuration for our model architecture, including network topology details and optimization parameters. Our implementation utilizes a modified U-Net architecture with residual blocks and multi-head self-attention layers. Table 1 summarizes the dataset-specific configurations. The spatial resolution and batch size were selected to maximize GPU memory utilization while maintaining stable training dynamics.

Table 1. Neural network hyperparameters used during experiments on different datasets.

Dataset	Network param.	Layer multipliers	Base Channels	Learning rate	Resolution	Batch	Attention lvls
FFHQ	210904835	(1, 2, 3, 4, 5)	128	2e-05	128×128	32	(2, 3, 4)
LSUN	261828227	(1, 2, 3, 4, 5)	128	2e-05	128×128	32	(2, 3, 4)
MNIST	42082049	(1, 2, 2)	128	2e-04	28×28	128	(2,)

All experiments were conducted on NVIDIA A100 GPUs using PyTorch 2.6.0. The FFHQ model trained for 1M iterations (≈146 hours), MNIST converged within 500k iterations (≈32 hours), we additionally train a model for LSUN Church dataset for 500k iterations (≈ 80 hours). We employed random horizontal flipping (p=0.5) for FFHQ augmentation, with no augmentation applied to MNIST and LSUN Church.

B. Additional samples

We present supplementary experimental results from parameter ablations examining the influence of the Peclet number (Pe) on image synthesis quality and sample distribution diversity. Experiments examine Peclet number (Pe) impacts on FFHQ-128 (Pe $\in \{0.0, 0.02, 0.04, 0.06, 0.08, 0.1, 0.12, 0.14\}$), MNIST (Pe $\in \{0.0, 0.02, 0.04, 0.06, 0.08, 0.1\}$) and LSUN Church (Pe $\in \{0.0, 0.02, 0.04, 0.06, 0.08\}$). This appendix documents comparative studies conducted on the FFHQ-128, MNIST and LSUN Church datasets, organized as follows: (i) single-initial-state sampling through generation, where ground truth (GT) images are propagated through the forward diffusion process with Pe-specific dynamics and reconstructed through model inference (Figs. 1, 7, 12); (ii) interpolation trajectories with Peclet ablations, illustrating transition dynamics under varying diffusion constraints (Figs. 2, 8, 13); (iii) multiple initial-state sampling via stochastic generation from diversified initial states, emphasizing Pe's role in governing output variability across distinct trajectory initializations (Figs. 3, 4, 9, 14); and (iv) uncurated interpolation demonstrating raw model behavior (Figs. 5, 6, 10, 11, 15, 16).

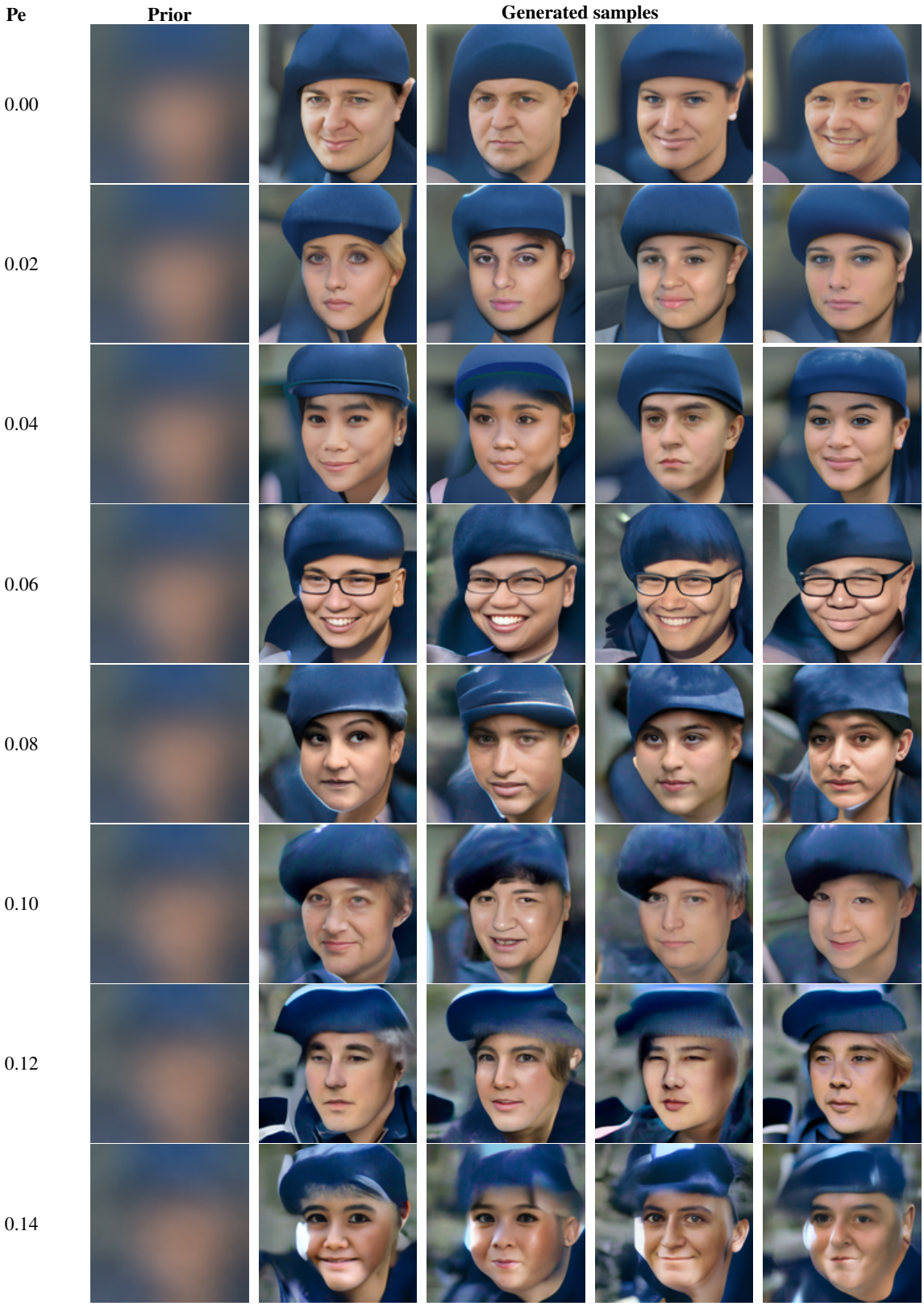


Figure 1. Results for $\sigma = 20$, showing inverse processes with varying Pe numbers. The image prior is consistent across rows for visual comparison, preserving the color palette.

Figure 2. Visual comparison of interpolations between two FFHQ samples. Each undergoes the forward process up to $\sigma = 20$, followed by linear interpolation and denoising with SLERP-interpolated generative noise added at each step.

Figure 3. Visual comparison of the results of our method and the IHD method on the FFHQ dataset.

Figure 4. Visual comparison of the results of our method and the IHD method on the FFHQ dataset.

Figure 5. Interpolations between two random images on FFHQ 128 $\times$ 128. σ = 16, Pe=0.8

Figure 6. Interpolations between two random images on FFHQ 128 $\times$ 128. σ = 20, Pe=0.8

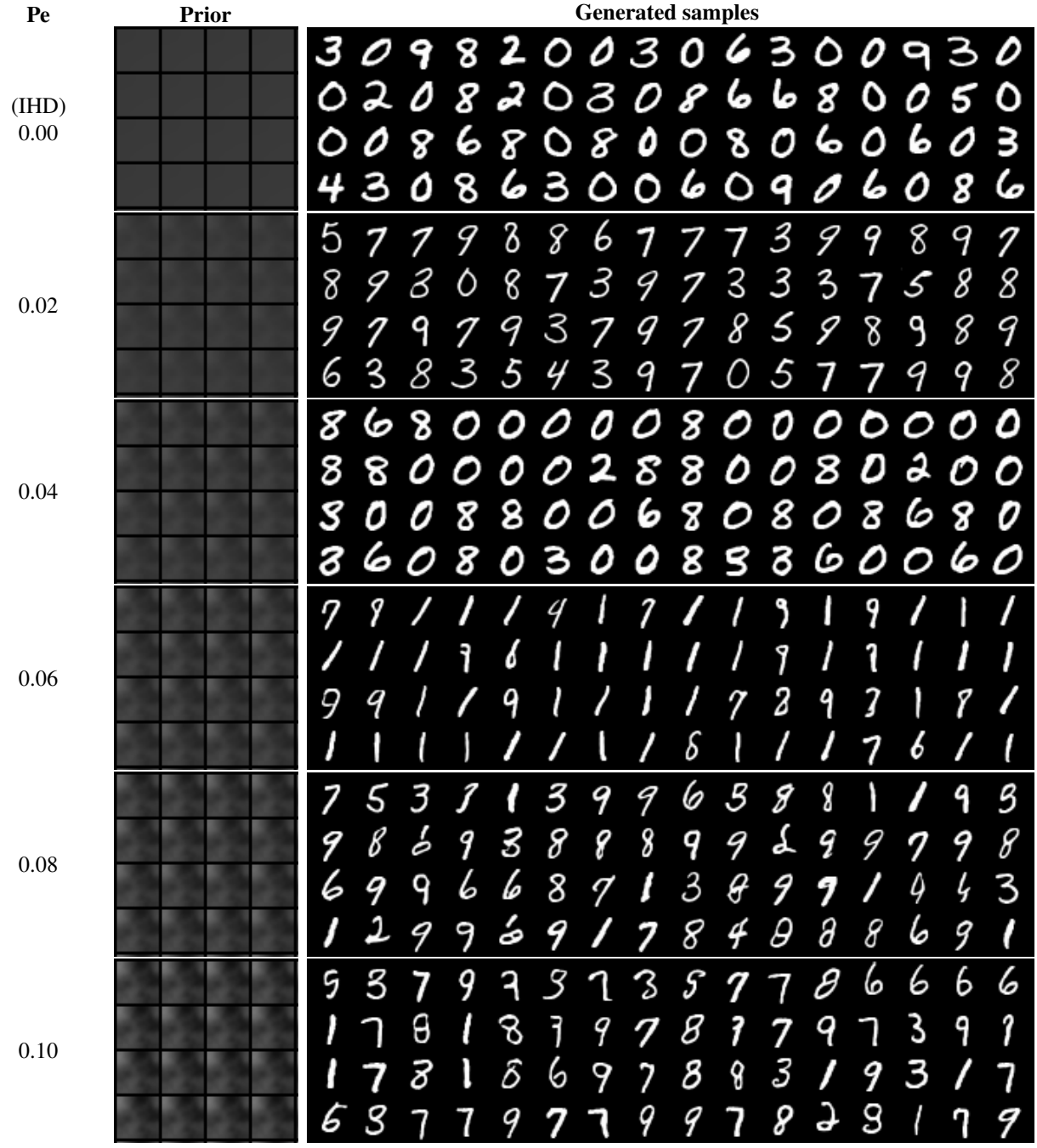


Figure 7. Additional samples with corresponding initial images from MNIST dataset, comparing different Peclet numbers. We can observe the impact of advection in forward process final step, that is the initial state for sampling.

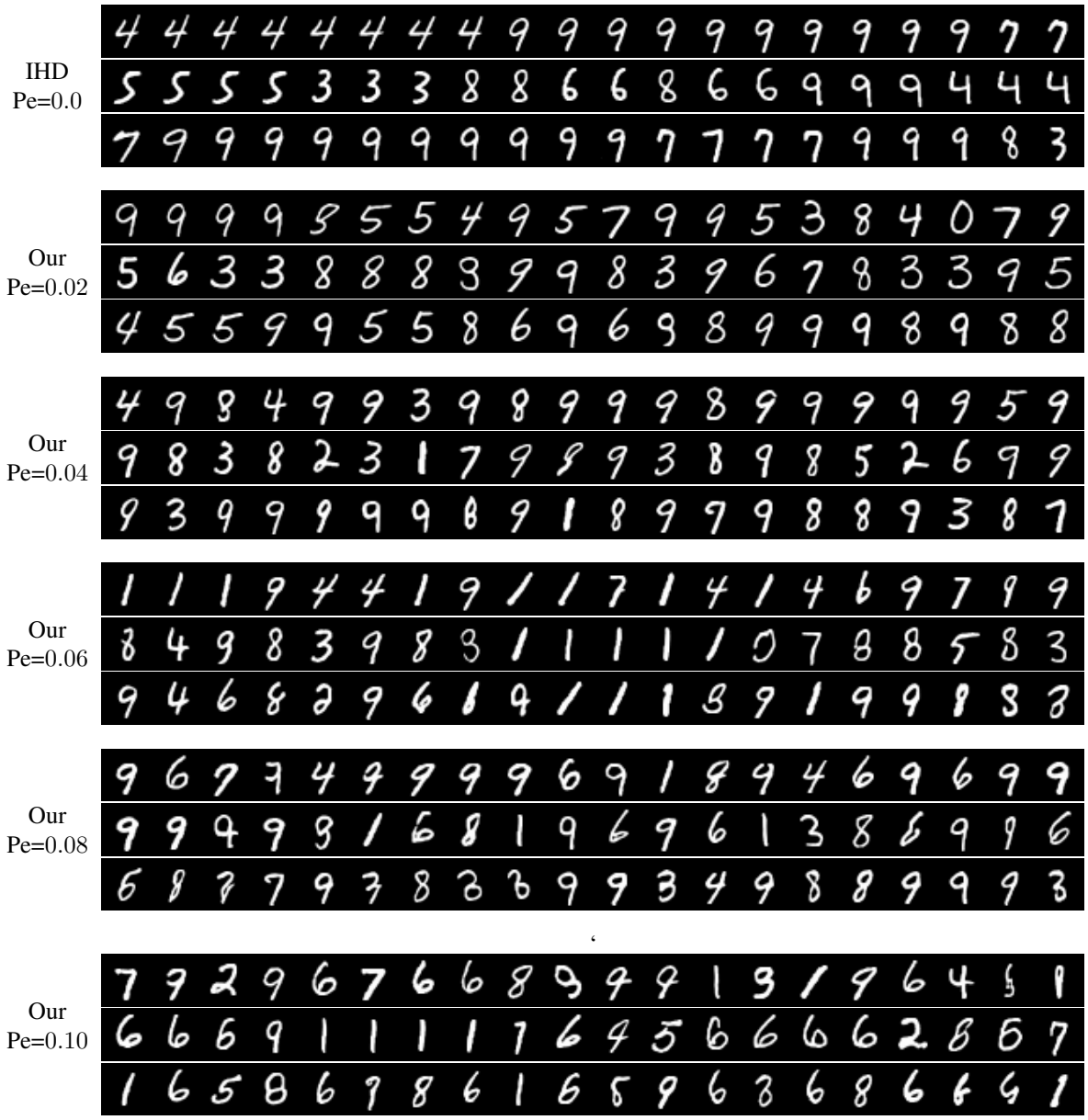
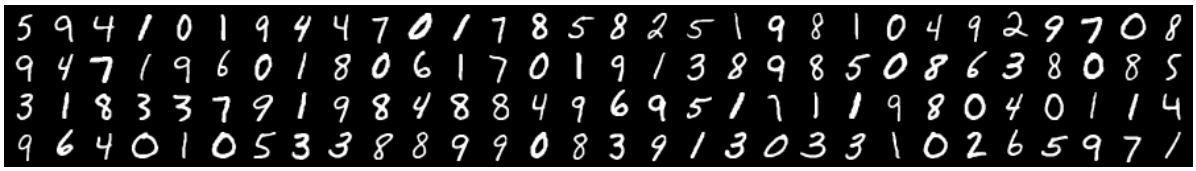
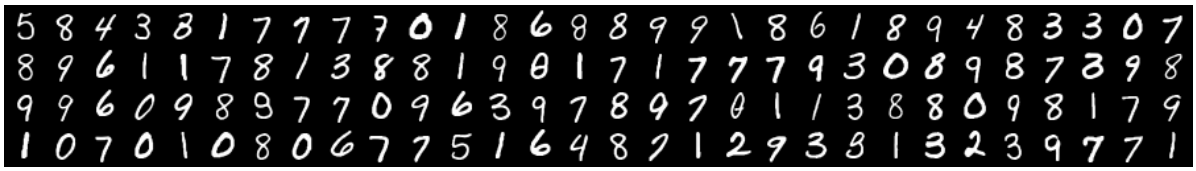


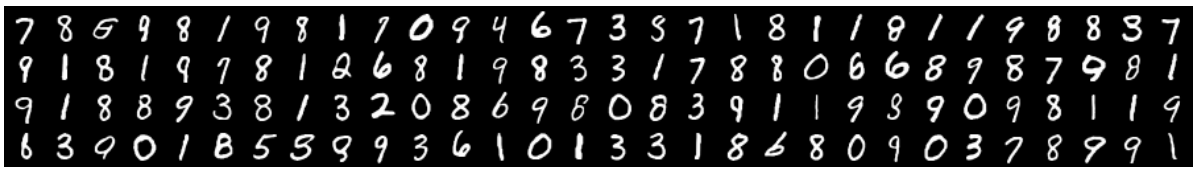
Figure 8. Visual comparison of interpolations between two MNIST samples. Each undergoes the forward process up to $\sigma = 20$, followed by linear interpolation and denoising with SLERP-interpolated generative noise added at each step.



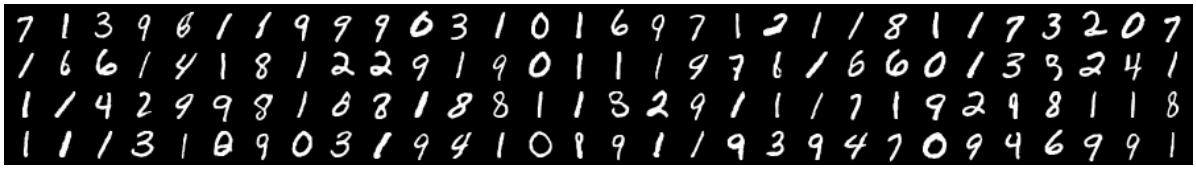
(a) (IHD) $Pe = 0.00$



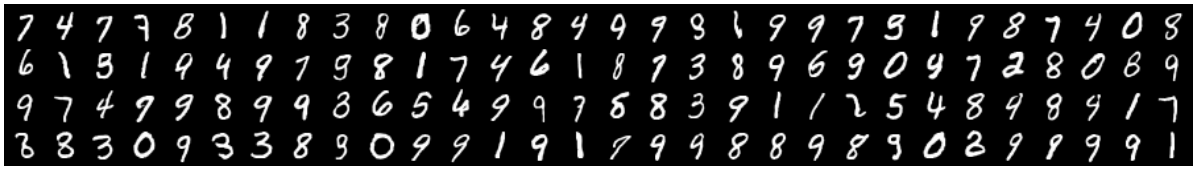
(b) $Pe = 0.02$



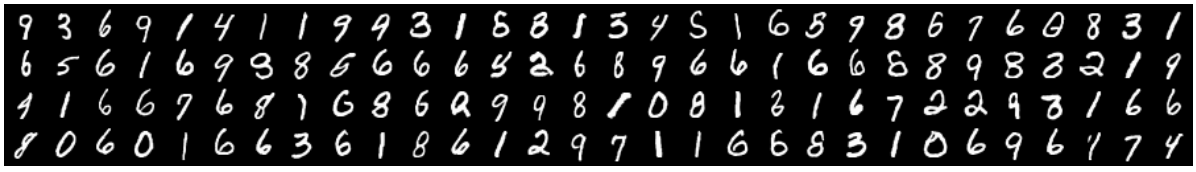
(c) $Pe = 0.04$



(d) $Pe = 0.06$



(e) $Pe = 0.08$



(f) $Pe = 0.10$

Figure 9. Visual comparison of the results of our method and the IHD method on the MNIST dataset.

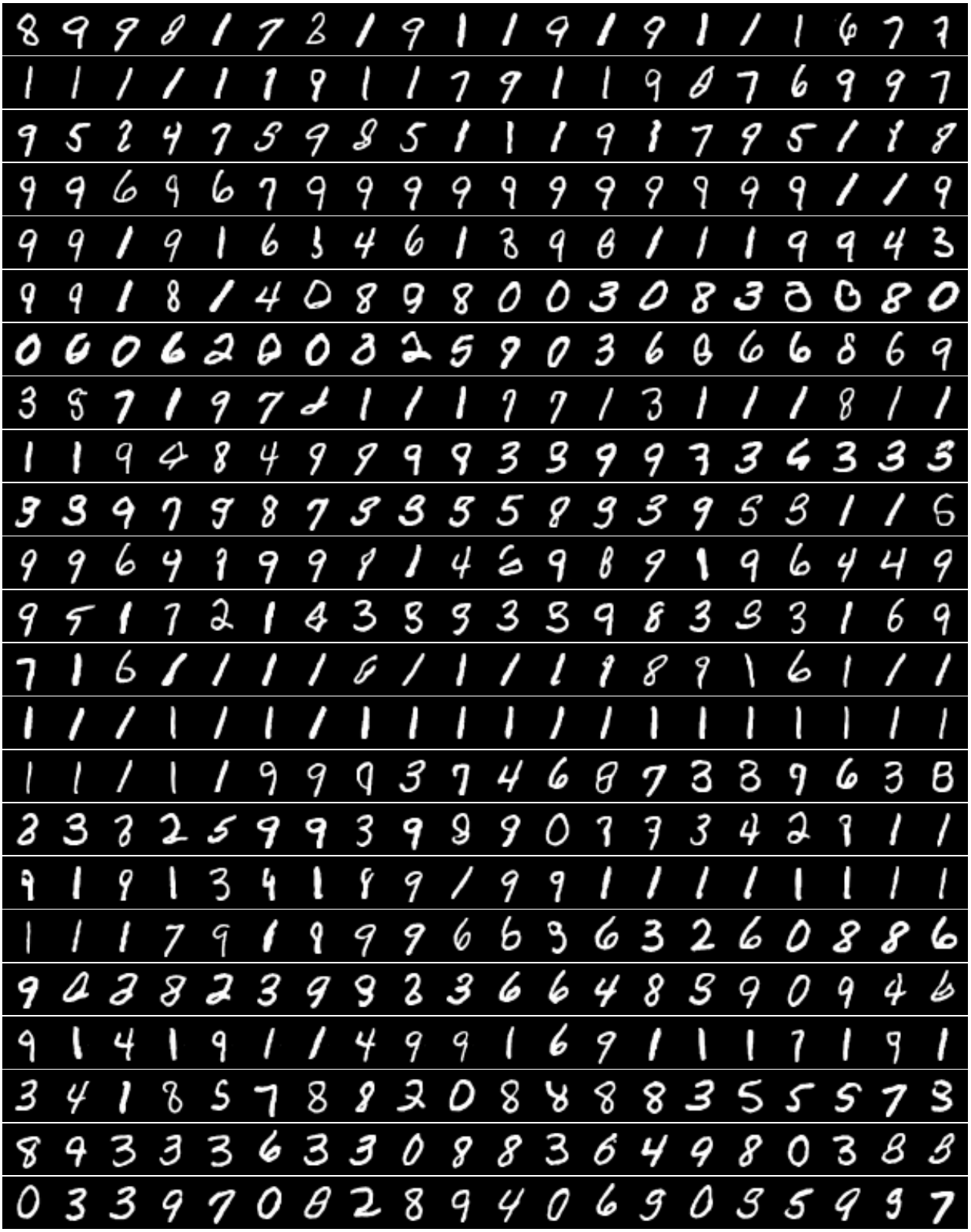


Figure 10. Interpolations between two random images on MNIST, $\sigma = 20$, $Pe = 0.6$

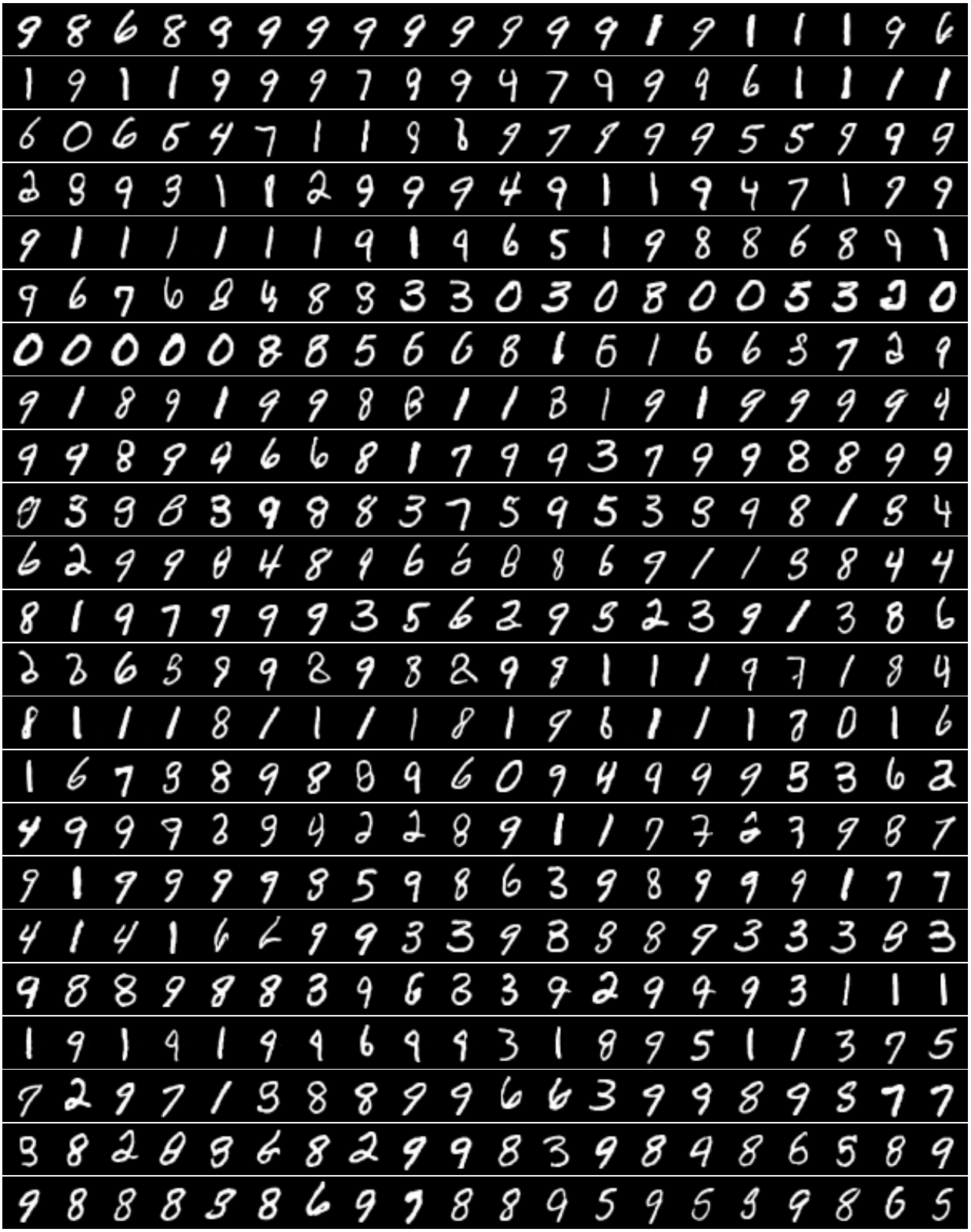


Figure 11. Interpolations between two random images on MNIST, $\sigma = 20$, $Pe = 0.8$

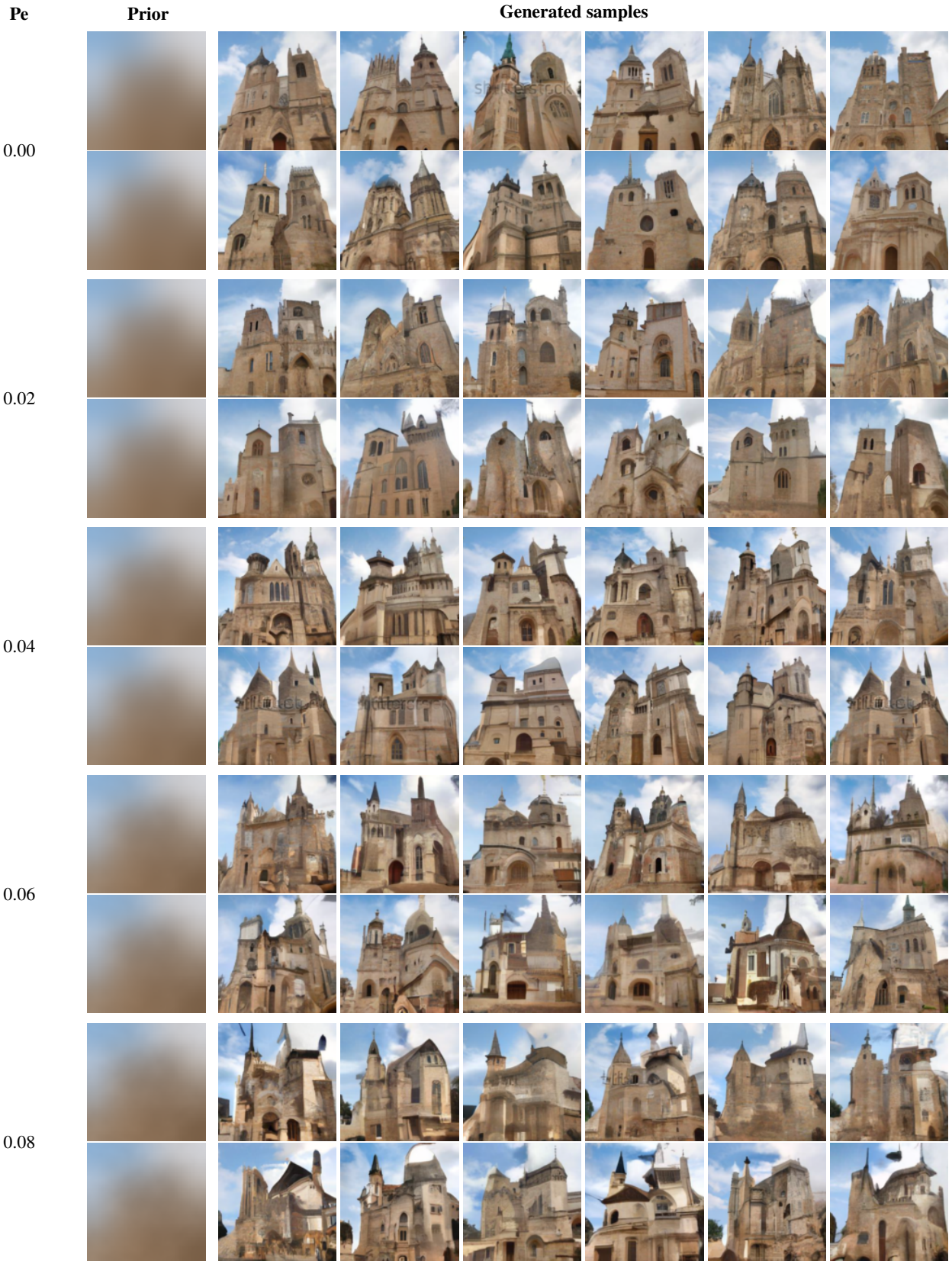


Figure 12. Results for $\sigma = 20$, on the LSUN Church dataset, showing inverse processes with varying Pe numbers. The image prior is consistent across rows for visual comparison, preserving the color palette.

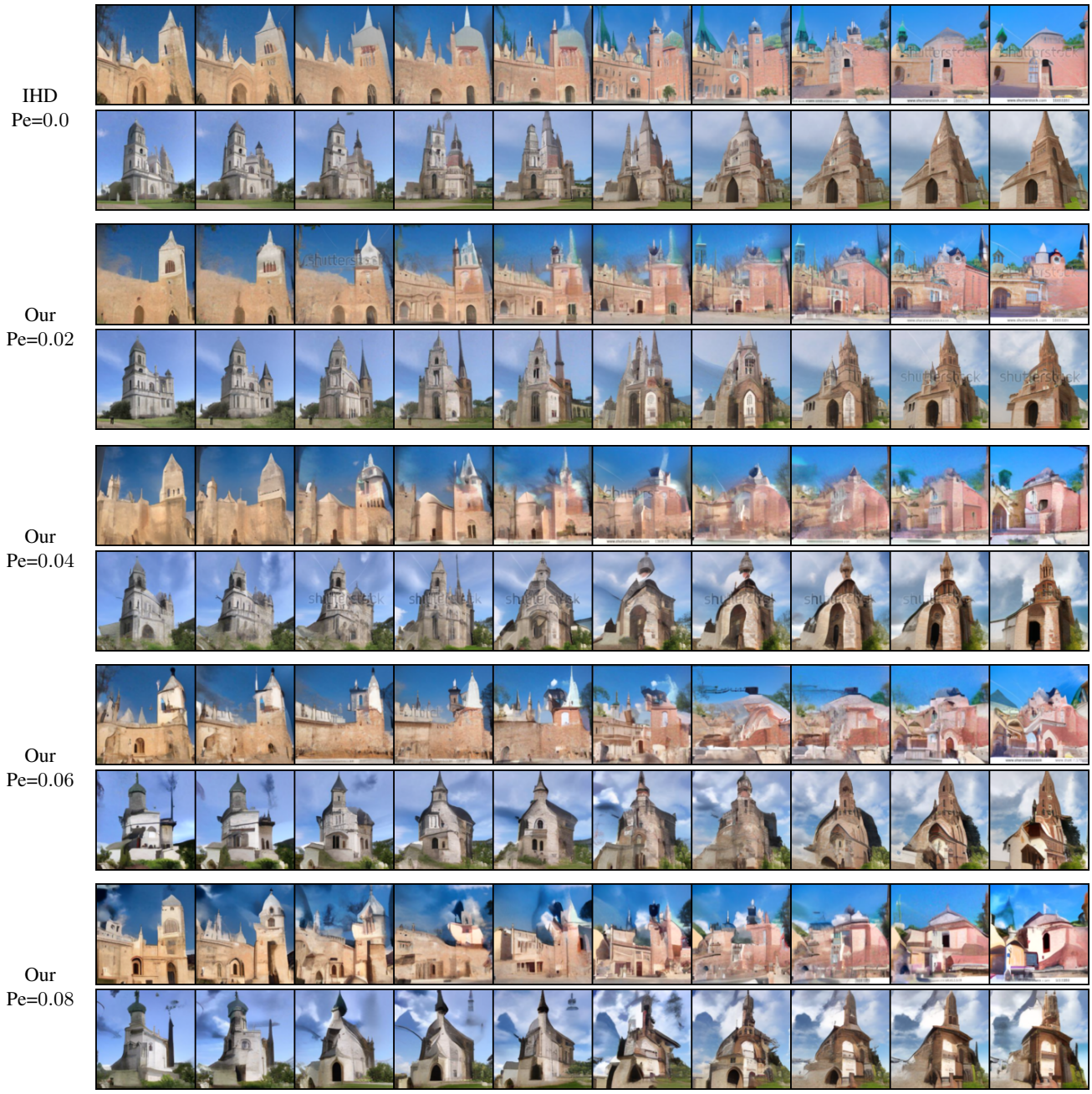


Figure 13. Visual comparison of interpolations between two LSUN Church samples. Each undergoes the forward process up to $\sigma = 20$, followed by linear interpolation and denoising with SLERP-interpolated generative noise added at each step.

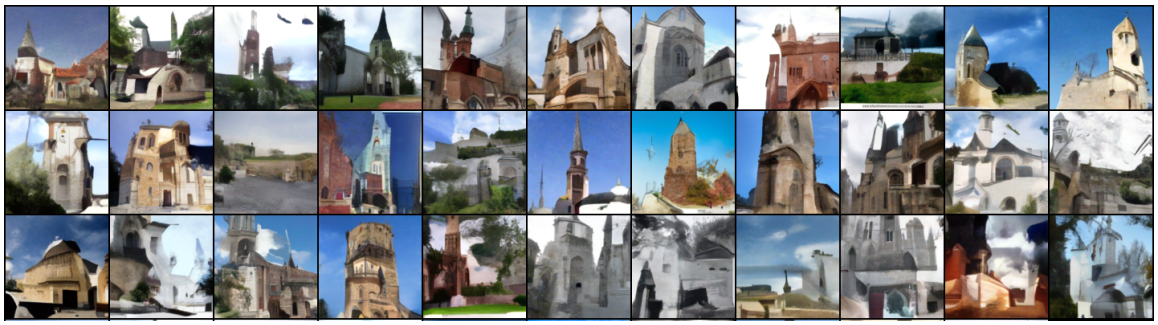
$\sigma = 20$

(IHD)
Pe=0.00

Pe=0.02

Pe=0.04

Pe=0.06



Pe=0.08

Figure 14. Visual comparison of the results of our method and the IHD method on the LSUN Church dataset.

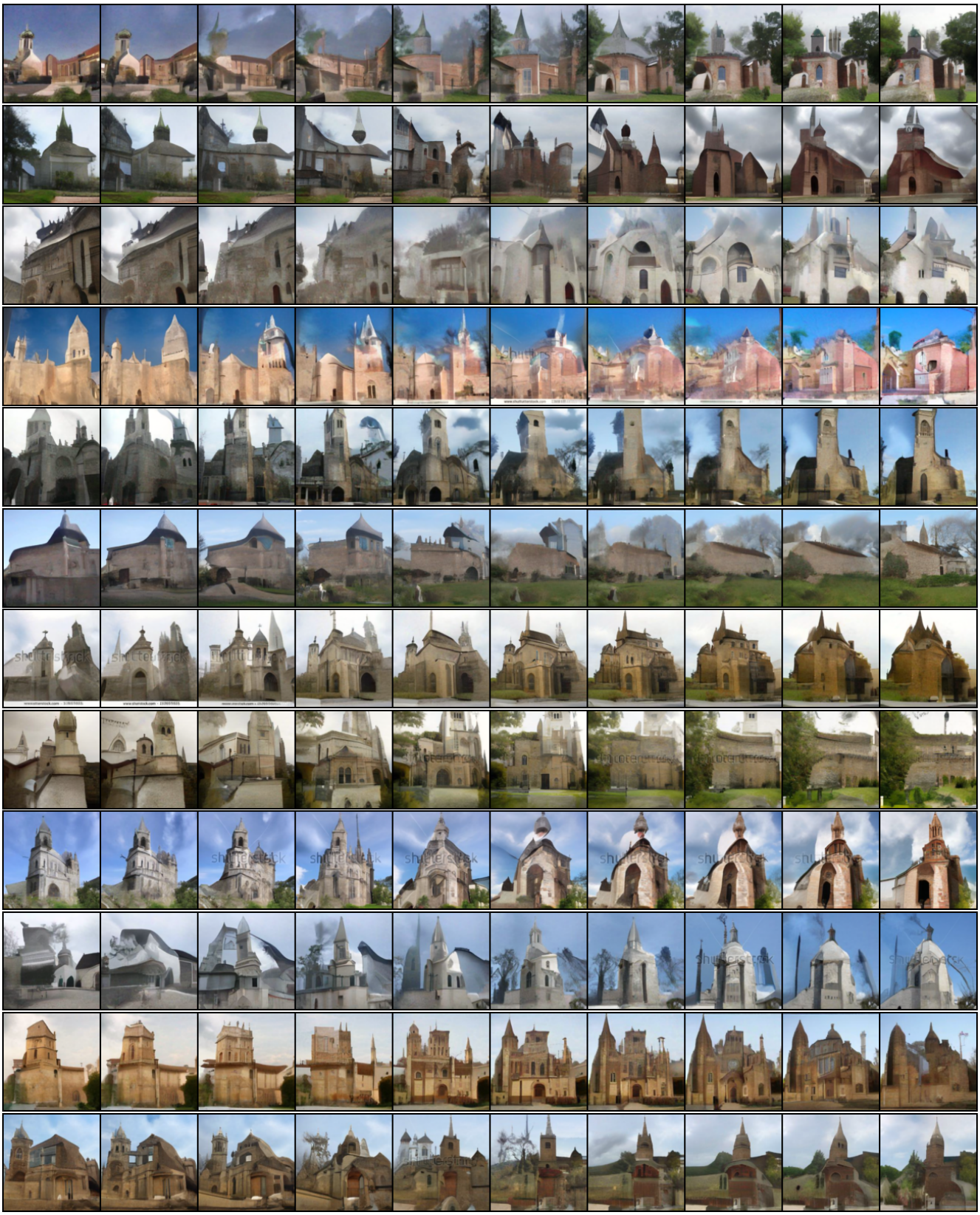


Figure 15. Interpolations between two random images on LSUN Church 128 $\times$ 128. σ = 20, Pe=0.4

Figure 16. Interpolations between two random images on LSUN Church 128 $\times$ 128. σ = 20, Pe=0.6

C. Pseudocode

We insert a pseudocode in Python to illustrate how the Spectral Turbulence Generator and Lattice Boltzmann Method (LBM) work internally.

```python
import torch as t

def tanh_limiter(x, min_val, max_val, sharpness=1.0):
    mid_val = (max_val + min_val) / 2
    range_val = (max_val - min_val) / 2
    return mid_val + range_val * t.tanh(sharpness * (x - mid_val) / range_val)

def limit_velocity_field(u, v, min_val, max_val):
    velocity_magnitude = t.sqrt(u**2 + v**2)
    limited_magnitude = tanh_limiter(velocity_magnitude, min_val, max_val)

    small_factor = 1E-9
    direction_factor = t.where(velocity_magnitude < small_factor, small_factor, limited_magnitude /
        velocity_magnitude)

    # Adjust u and v components to match the new limited magnitude while preserving direction
    upscale = 1.
    direction_factor *=upscale
    u_limited = u * direction_factor
    v_limited = v * direction_factor

    return u_limited, v_limited

class SpectralTurbulenceGenerator(t.nn.Module):
    def __init__(self, std_dev_schedule):
        grid_size = (128, 128)
        self.std_dev = std_dev_schedule #turbulence intensity scaling factor

        energy_slope = -2.0
        self.energy_spectrum = lambda k: t.where(t.isinf(k ** (energy_slope)), 0, k ** (energy_slope))
        self.frequency_range = {'k_min': 2.0 * t.pi / min(grid_size), 'k_max': 2.0 * t.pi /
            (min(self.domain_size) / 1024)}

        # Fourier transform wave numbers
        kx = (t.fft.fftfreq(grid_size[0], d=1/grid_size[0]) * 2 * t.pi).to('gpu')
        ky = (t.fft.fftfreq(grid_size[1], d=1/grid_size[1]) * 2 * t.pi).to('gpu')
        KX, KY = t.meshgrid(kx, ky)
        self.K = t.sqrt(KX**2 + KY**2).to('gpu')

        # Initialize the phases once and use them in each run
        self.phase_u = (t.rand(grid_size) * 2 * t.pi).to('gpu')
        self.phase_v = (t.rand(grid_size) * 2 * t.pi).to('gpu')

        self.amplitude = (t.where(self.K != 0, (self.energy_spectrum(self.K)), 0)).to('gpu')
        self.amplitude = (t.where((self.K >= self.frequency_range['k_min']) & (self.K <=
            self.frequency_range['k_max']), self.amplitude, 0.0)).to("gpu")

        dt_turb = 1E-4
        self.omega = dt_turb*self.K

    def generate_turbulence(self, time: int):
        u_hat = self.amplitude * t.exp(1j * (self.phase_u + self.omega * time))
        v_hat = self.amplitude * t.exp(1j * (self.phase_v + self.omega * time))
        u = t.real(t.fft.ifft2(u_hat))
        v = t.real(t.fft.ifft2(v_hat))

        if self.std_dev[time]< 1E-14:
            u,v = 0*self.K, 0*self.K #avoid division by 0 in t.std(u)
        else:
            u *= self.std_dev[time] / t.std(u)
            v *= self.std_dev[time] / t.std(v)
```

```
        u, v = limit_velocity_field(u, v, min_val=-1E-3, max_val=1E-3)

        return u.float(), v.float()
```

```
import taichi as ti
import taichi.math as tm

# Fluid solver based on lattice boltzmann method using taichi language
# Inspired by: https://github.com/hietwll/LBM_Taichi

@ti.data_oriented
class LBM_ADE_Solver():
    def __init__(self, config, turbulenceGenerator):
        self.nx, self.ny = config.domain_size
        self.turbulenceGenerator = turbulenceGenerator

        self.cs2 = ti.field(ti.f32)(1./3.)
        self.omega_kin = ti.field(ti.f32, shape=self.max_iter[None])
        self.omega_kin.from_numpy(1.0 / (3.0* config.kin_visc + 0.5))

        self.rho = ti.field(float, shape=(self.nx, self.ny))
        self.vel = ti.Vector.field(2, float, shape=(self.nx, self.ny))

        self.f = ti.Vector.field(9, float, shape=(self.nx, self.ny))
        self.f_new = ti.Vector.field(9, float, shape=(self.nx, self.ny))

        self.Force = ti.Vector.field(2, float, shape=(self.nx, self.ny))
        self.w = ti.types.vector(9, float)(4, 1, 1, 1, 1, 1 / 4, 1 / 4, 1 / 4, 1 / 4) / 9.0
        self.e = ti.types.matrix(9, 2, int)(
            [0, 0],  [1, 0], [0, 1], [-1, 0],[0, -1],[1, 1], [-1, 1], [-1, -1], [1, -1])

    def init(self, np_image):
        self.rho.from_numpy(np_image)
        self.vel.fill(0)

    def solve(self, iterations):
        for iteration in range(iterations):
            self.stream()
            self.update_macro_var()
            self.collide_srt()
            self.vel = self.turbulenceGenerator.generate_turbulence(iteration)
            self.apply_bounceback_boundary_condition()

    @ti.kernel
    def stream(self):
        for i, j in ti.ndrange(self.nx, self.ny):
            for k in ti.static(range(9)):
                ip = i - self.e[k, 0]
                jp = j - self.e[k, 1]
                self.f[i, j][k] = self.f_new[ip, jp][k]

    @ti.kernel
    def update_macro_var(self):
        for i, j in ti.ndrange((1, self.nx-1), (1,self.ny-1)):
            self.rho[i, j] = 0
            for k in ti.static(range(9)):
                self.rho[i, j] += self.f[i, j][k]

    @ti.kernel
    def collide_srt(self):
        omega_kin = self.omega_kin[self.iterations_counter[None]]
        for i, j in ti.ndrange((1, self.nx - 1), (1, self.ny - 1)):
            for k in ti.static(range(9)):
                feq = self.f_eq(i, j)
                self.f_new[i, j][k] = (1. - omega_kin) * self.f[i, j][k] + feq[k] * omega_kin

    @ti.func
```

```
    def f_eq(self, i, j):
        eu = self.e @ self.vel[i, j]
        uv = tm.dot(self.vel[i, j], self.vel[i, j])
        return self.w * self.rho[i, j] * (1 + 3 * eu + 4.5 * eu * eu - 1.5 * uv)

    @ti.func
    def apply_bounceback_core(self, i: int, j: int):
        tmp = ti.f32(0.0)
        for k in ti.static([1,2,5,6]):
            tmp = self.f[i, j][k]
            self.f[i, j][k] = self.f[i, j][k+2]
            self.f[i, j][k+2] = tmp

        for k in ti.static(range(9)):
             self.f_new[i, j][k] =  self.f[i, j][k]

    @ti.kernel
    def apply_bounceback_boundary_condition(self):
        for i in range(0, self.nx):
            self.apply_bounceback_core(i, 0)
            self.apply_bounceback_core(i, self.ny-1)

        for j in range(1, self.ny-1):
            self.apply_bounceback_core(0, j)
            self.apply_bounceback_core(self.nx-1, j)
```