**Computer Assignment 1 (Langevin sampling algorithm)**

In [1]:
```
import torch
import matplotlib.pyplot as plt
import numpy as np

torch.manual_seed(42)
```

Out[1]:
```
<torch._C.Generator at 0x24ba2721910>
```

Plotting options (do not change)

In [27]:
```
plotting_range = np.array([[-4, 6], [-4, 6]])
nbins = 50
density = False
```

Specify mean and covariance

In [28]:
```
mean = torch.tensor([1., 1.])
cov = torch.tensor([[1., 0.9],
        [0.9, 1.]])
cov_inv = torch.linalg.inv(cov) # needed for the score function
```

The log density of a multivariate gaussian is

$$\log p(\boldsymbol{x}) = -\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}) + C,$$

where $C$ is a constant. The score function is then

$$\phi(\boldsymbol{x}) = \nabla_{\boldsymbol{x}} \log p(\boldsymbol{x}) = -\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}).$$

In [29]:
```
def score(x):
    return -cov_inv @ (x - mean)
```

The Langevin iteration is defined as

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \mu\phi(\boldsymbol{x}_t) + \sqrt{2\mu}\boldsymbol{n}_t,$$

where $\boldsymbol{n} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and $\mu$ is the step size.

In [30]:
```
def langevin_dynamics(x0, T, mu):
    samples = []
    x = x0
    for t in range(T):
        n = torch.randn(x0.size())
        x = x + mu*score(x) + torch.sqrt(torch.tensor(2*mu))*n
        samples.append(x)
    return np.array(samples)
```

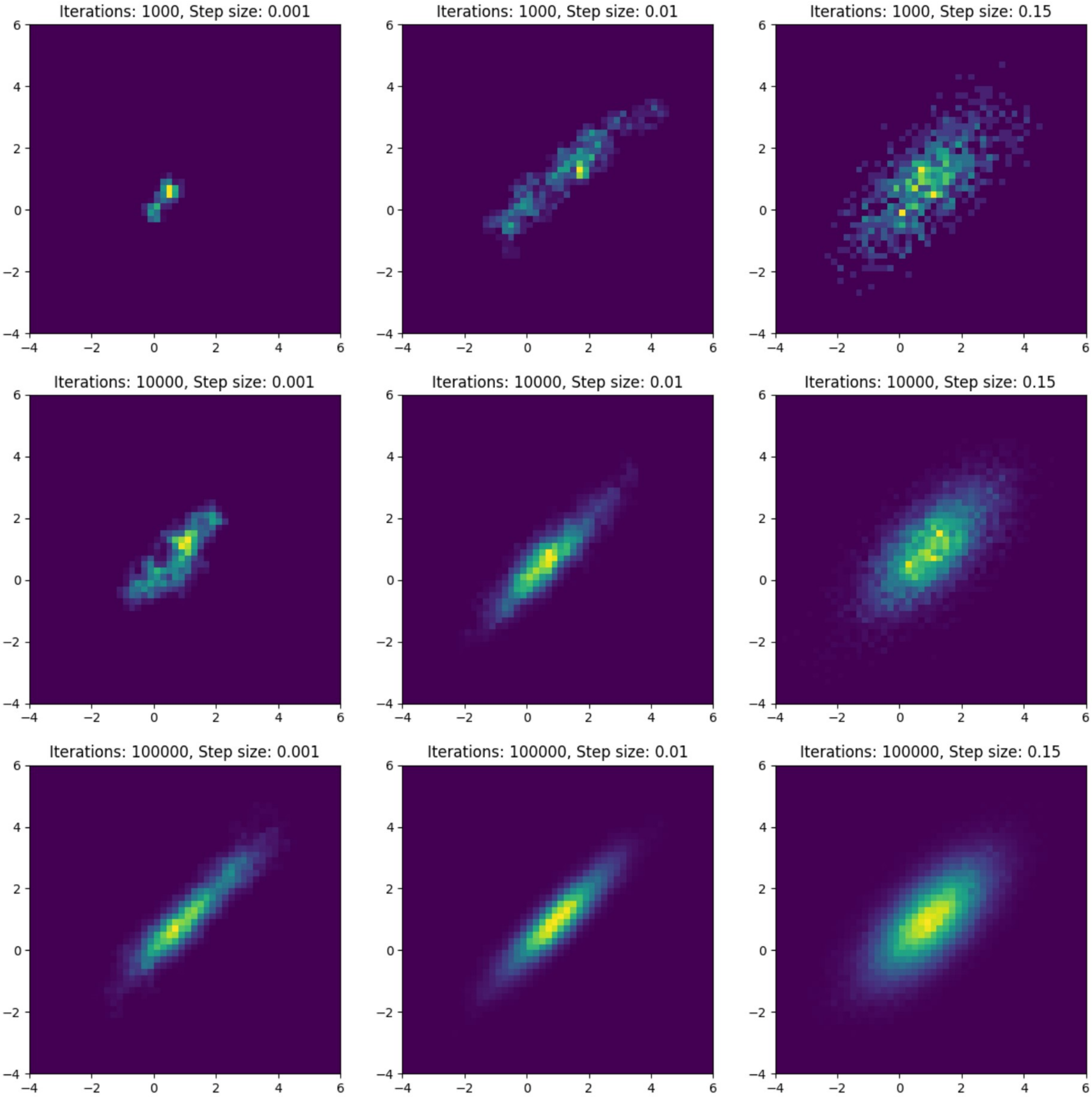
Considered scenarios

In [31]:
```
Ts = [1_000, 10_000, 100_000]
mus = [0.001, 0.01, 0.15]
```

```
x0 = torch.tensor([0,0]) # start from origin
```

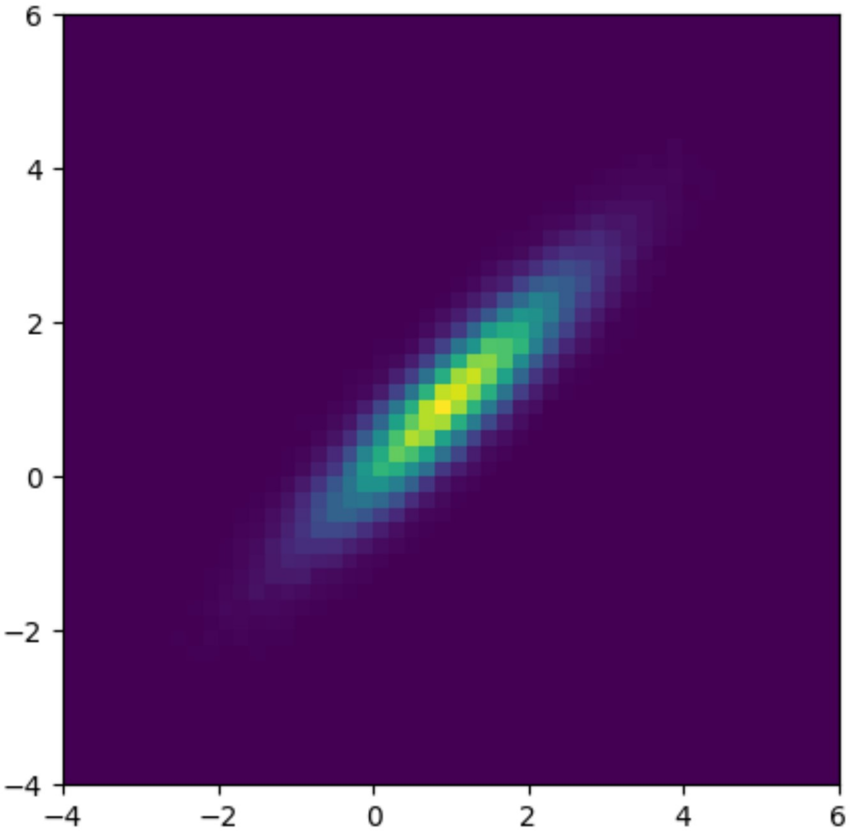
Run sampling

In [32]:
```
fig, axes = plt.subplots(3, 3, figsize=(15, 15))
for i,T in enumerate(Ts):
    for j,mu in enumerate(mus):
        samples = langevin_dynamics(x0,T,mu)
        axes[i,j].hist2d(samples[:, 0], samples[:, 1], bins=nbins, range=plottin
        axes[i,j].set_title(f'Iterations: {T}, Step size: {mu}')
plt.show()
```




Compare results to the target

In [33]:
```
target = torch.distributions.multivariate_normal.MultivariateNormal(mean, cov)
target_samples = target.sample((Ts[2],))
plt.figure(figsize=(5, 5))
plt.hist2d(target_samples[:, 0], target_samples[:, 1], bins=nbins, range=plottin
plt.show()
```



The best hyperparameters were $T = 100000$ and $\mu = 0.01$. All runs started from the origin, i.e. $\boldsymbol{x}_0 = \mathbf{0}$. The step size needs to be small similarly as in SGD so that we don't make too drastic jumps e.g. bouncing between modes, but continue going towards increasing probability mass, while the noise term allows us not to get stuck in modes. The number of steps needs to be high, since the Langevin algorithm is a Markov process. This allows us to explore the whole distribution and also to move out from the starting position.

**Computer Assignment 2 (GAN)**

In [59]:
```
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
from tqdm import tqdm
torch.manual_seed(42)
```

Out[59]: <torch._C.Generator at 0x24ba2721910>

In [54]:
```
noise_dim = 100
lr = 2e-4
batch_size = 64
num_epochs = 30
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

In [55]:
```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize([0.5], [0.5]) # scales images to [-1, 1] which is appar
])
```

```python
dataset = datasets.MNIST("data", train=True, download=True, transform=transform)

train_loader = DataLoader(dataset, batch_size=batch_size)
```

In [56]:
```python
class Generator(nn.Module):
    def __init__(self, noise_dim):
        super(Generator, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(noise_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, 784),
            nn.Tanh() # output needs to be in [-1, 1] as well
            )

    def forward(self, x):
        x = self.layers(x)
        return x.view(-1, 1, 28, 28)

class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.layers = nn.Sequential(
            nn.Flatten(),
            nn.Linear(784, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 1),
            nn.Sigmoid() # output in [0, 1]
        )

    def forward(self, x):
        return self.layers(x)
```

In [57]:
```python
generator = Generator(noise_dim).to(device)
discriminator = Discriminator().to(device)

criterion = nn.BCELoss()
opt_gen = optim.Adam(generator.parameters(), lr=lr, betas=(0.5, 0.999))
opt_disc = optim.Adam(discriminator.parameters(), lr=lr, betas=(0.5, 0.999))
```

In [60]:
```python
for epoch in range(num_epochs):
    for real_imgs, _ in tqdm(train_loader):
        real_imgs = real_imgs.to(device)
        real_labels = torch.ones(real_imgs.size(0), 1, device=device)
        fake_labels = torch.zeros(real_imgs.size(0), 1, device=device)

        # train the generator first
        opt_gen.zero_grad()
        noise = torch.randn(real_imgs.size(0), 1, noise_dim, device=device)
        generated_imgs = generator(noise)
        gen_loss = criterion(discriminator(generated_imgs), real_labels)
        gen_loss.backward()
        opt_gen.step()

        # then the discriminator
        opt_disc.zero_grad()
```

```
        real_loss = criterion(discriminator(real_imgs), real_labels)
        fake_loss = criterion(discriminator(generated_imgs.detach()), fake_label
        disc_loss = real_loss + fake_loss
        disc_loss.backward()
        opt_disc.step()

    print(f"[Epoch {epoch+1}/{num_epochs}] Discriminator loss: {disc_loss.item()
```

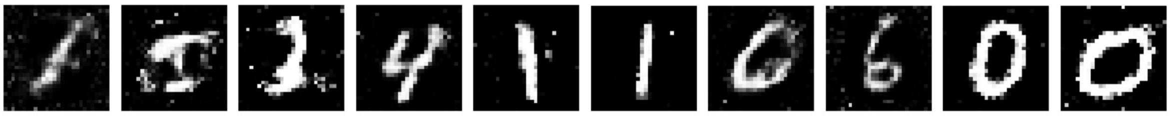
```
100%|██████████| 938/938 [00:20<00:00, 46.30it/s]
[Epoch 1/30] Discriminator loss: 1.6516 Generator loss: 0.2228
100%|██████████| 938/938 [00:20<00:00, 46.68it/s]
[Epoch 2/30] Discriminator loss: 0.1623 Generator loss: 1.9703
100%|██████████| 938/938 [00:20<00:00, 46.76it/s]
[Epoch 3/30] Discriminator loss: 0.8965 Generator loss: 0.7263
100%|██████████| 938/938 [00:20<00:00, 45.22it/s]
[Epoch 4/30] Discriminator loss: 0.2520 Generator loss: 1.9755
100%|██████████| 938/938 [00:21<00:00, 44.52it/s]
[Epoch 5/30] Discriminator loss: 0.2239 Generator loss: 2.8118
100%|██████████| 938/938 [00:20<00:00, 46.71it/s]
[Epoch 6/30] Discriminator loss: 0.2339 Generator loss: 3.9131
100%|██████████| 938/938 [00:19<00:00, 47.09it/s]
[Epoch 7/30] Discriminator loss: 0.1656 Generator loss: 2.3488
100%|██████████| 938/938 [00:19<00:00, 47.01it/s]
[Epoch 8/30] Discriminator loss: 0.1023 Generator loss: 4.0226
100%|██████████| 938/938 [00:19<00:00, 46.96it/s]
[Epoch 9/30] Discriminator loss: 0.2323 Generator loss: 2.9202
100%|██████████| 938/938 [00:20<00:00, 46.62it/s]
[Epoch 10/30] Discriminator loss: 0.8166 Generator loss: 1.1091
100%|██████████| 938/938 [00:20<00:00, 46.22it/s]
[Epoch 11/30] Discriminator loss: 0.4115 Generator loss: 3.4981
100%|██████████| 938/938 [00:20<00:00, 46.16it/s]
[Epoch 12/30] Discriminator loss: 0.7873 Generator loss: 1.0700
100%|██████████| 938/938 [00:20<00:00, 46.05it/s]
[Epoch 13/30] Discriminator loss: 0.5325 Generator loss: 1.7074
100%|██████████| 938/938 [00:20<00:00, 46.24it/s]
[Epoch 14/30] Discriminator loss: 0.8876 Generator loss: 1.5214
100%|██████████| 938/938 [00:20<00:00, 46.27it/s]
[Epoch 15/30] Discriminator loss: 0.6623 Generator loss: 1.3686
100%|██████████| 938/938 [00:20<00:00, 46.30it/s]
[Epoch 16/30] Discriminator loss: 0.8796 Generator loss: 1.1974
100%|██████████| 938/938 [00:20<00:00, 45.99it/s]
[Epoch 17/30] Discriminator loss: 1.1790 Generator loss: 0.8370
100%|██████████| 938/938 [00:20<00:00, 46.45it/s]
[Epoch 18/30] Discriminator loss: 0.4375 Generator loss: 2.3606
100%|██████████| 938/938 [00:20<00:00, 46.05it/s]
[Epoch 19/30] Discriminator loss: 0.7311 Generator loss: 1.1450
100%|██████████| 938/938 [00:20<00:00, 46.07it/s]
[Epoch 20/30] Discriminator loss: 0.7714 Generator loss: 1.4216
100%|██████████| 938/938 [00:20<00:00, 46.32it/s]
[Epoch 21/30] Discriminator loss: 0.8048 Generator loss: 1.0607
100%|██████████| 938/938 [00:20<00:00, 46.29it/s]
[Epoch 22/30] Discriminator loss: 0.7095 Generator loss: 1.2142
100%|██████████| 938/938 [00:20<00:00, 46.31it/s]
[Epoch 23/30] Discriminator loss: 0.6679 Generator loss: 1.3167
```

```
100%|██████████| 938/938 [00:20<00:00, 46.40it/s]
[Epoch 24/30] Discriminator loss: 0.7676 Generator loss: 1.3774
100%|██████████| 938/938 [00:20<00:00, 46.53it/s]
[Epoch 25/30] Discriminator loss: 0.5914 Generator loss: 1.9441
100%|██████████| 938/938 [00:20<00:00, 46.19it/s]
[Epoch 26/30] Discriminator loss: 0.5392 Generator loss: 1.8826
100%|██████████| 938/938 [00:20<00:00, 45.07it/s]
[Epoch 27/30] Discriminator loss: 0.7850 Generator loss: 1.4204
100%|██████████| 938/938 [00:23<00:00, 40.51it/s]
[Epoch 28/30] Discriminator loss: 0.6004 Generator loss: 1.4113
100%|██████████| 938/938 [00:23<00:00, 40.71it/s]
[Epoch 29/30] Discriminator loss: 0.7256 Generator loss: 1.3258
100%|██████████| 938/938 [00:20<00:00, 46.07it/s]
[Epoch 30/30] Discriminator loss: 0.9946 Generator loss: 0.8654
```

In [63]:
```python
generator.eval()
torch.manual_seed(42)
with torch.no_grad():
    noise = torch.randn(10, noise_dim, device=device)
    samples = generator(noise).cpu()

fig, axes = plt.subplots(1, 10, figsize=(15, 2))
for i in range(10):
    axes[i].imshow(samples[i][0], cmap='gray')
    axes[i].axis('off')
plt.tight_layout()

plt.show()
```



The generation appears to be quite successful and you can distinguish most of the generated digits quite easily.