STAT_MECH_ASSIGNMENT_02

September 10, 2025

1 Multivariate Gaussian Integral (n = 1..10) — Algorithm & Pseudocode with Explanations

1.1 Problem

We want to compute, for dimensions $n = 1, 2, \ldots, 10$, the **multivariate Gaussian integral**:

$$I_n = \int_{\mathbb{R}^n} \exp\left(-\frac{1}{2}x^\top Ax + b^\top x\right) dx$$

Here:

- x is a vector of length n,
- A is an $n \times n$ **symmetric positive definite matrix**,
- b is a vector of length n.

We will compare two ways of computing this integral:

1. The **analytic** value — using known formulas from statistics / linear algebra,
2. A **numerical** estimate — using **importance sampling**, a Monte Carlo method.

Finally, we will plot the **relative error** between the two approaches as a function of dimension n.

1.2 Inputs

To keep things simple and reproducible, we fix the following:

- **Dimension:** $n \in \{1, 2, \ldots, 10\}$ — the size of the vector and matrix
- **Matrix** A: A **tridiagonal matrix** with:
  - diagonal elements $d_0 = 3.0$,
  - off-diagonal elements $a = 0.4$,
  - which guarantees A is **symmetric positive definite** (i.e., integral converges).
- **Vector** b: A random vector drawn from a standard normal distribution $\mathcal{N}(0, 1)^n$. We use a fixed seed so the results are the same every time.
- **Monte Carlo samples** N: Number of random samples to estimate the integral numerically. We choose:

$$N = 60{,}000$$

- **Proposal distribution standard deviation** s: The importance sampler uses a Gaussian centered at the mode. We set:

$$s = \frac{1}{\sqrt{d_0}} \approx 0.577$$

1.3 Outputs

For each dimension n, we will record and report:

- The **analytic value** of the integral
- The **numeric estimate** (from sampling)
- The **standard error** (how noisy the numeric estimate is)
- The **relative error** (how close numeric is to analytic)

Then, we will **plot relative error vs. dimension** n.

1.4 Algorithm Overview

1.4.1 Step-by-step for each $n = 1, 2, \ldots, 10$:

1.4.2 Step 1: Construct the matrix A

We create an $n \times n$ matrix that is:

- **Symmetric**: $A = A^\top$
- **Tridiagonal**: only diagonal and nearest neighbors are non-zero
- **Toeplitz**: same values along diagonals

It looks like this (for example, if $n = 4$):

$$A = \begin{bmatrix} 3.0 & 0.4 & 0 & 0 \\ 0.4 & 3.0 & 0.4 & 0 \\ 0 & 0.4 & 3.0 & 0.4 \\ 0 & 0 & 0.4 & 3.0 \end{bmatrix}$$

This matrix has **fixed eigenvalues bounded away from zero**, so the integral is always well-defined.

1.4.3 Step 2: Draw a random vector b

We generate a random vector $b \in \mathbb{R}^n$, where each element is drawn independently from a standard normal distribution $\mathcal{N}(0, 1)$.

This vector gives us a linear term in the exponent: $b^\top x$.

1.4.4 Step 3: Compute the analytic value

We use the closed-form formula for the integral of a multivariate Gaussian of the form:

$$\int \exp\left(-\frac{1}{2}x^\top Ax + b^\top x\right) dx = \exp\left(\frac{n}{2}\log(2\pi) - \frac{1}{2}\log\det A + \frac{1}{2}b^\top A^{-1}b\right)$$

Here's how we compute it:

- Solve $A\mu = b$ to get $\mu = A^{-1}b$ (we never compute A^{-1} directly — just solve the equation)
- Compute the **quadratic term**: $b^\top \mu$
- Compute $\log\det A$ using a stable method (`np.linalg.slogdet`)
- Plug into the formula:

$$\log I = \frac{n}{2}\log(2\pi) - \frac{1}{2}\log\det A + \frac{1}{2}b^\top \mu$$

- Finally, exponentiate to get:

$$I_{\text{analytic}} = \exp(\log I)$$

1.4.5 Step 4: Estimate numerically using importance sampling

We now approximate the integral using **importance sampling** which is Monte Carlo technique.

We sample from a Gaussian proposal distribution:

$$q(x) = \mathcal{N}(\mu, s^2 I)$$

This distribution is centered at the mode of the integrand (i.e., at $\mu = A^{-1}b$), and has variance s^2 in every direction.

For each sample $i = 1$ to N:

1. **Draw sample**:

$$x_i = \mu + sz_i \quad \text{where } z_i \sim \mathcal{N}(0, I)$$

2. **Evaluate the integrand** at x_i:

$$\log f_i = -\frac{1}{2} x_i^\top A x_i + b^\top x_i$$

3. **Evaluate the proposal density** q at x_i:

$$\log q_i = -\frac{n}{2}\log(2\pi) - \frac{n}{2}\log(s^2) - \frac{1}{2s^2}\|x_i - \mu\|^2$$

4. **Compute importance weight**:

$$w_i = \exp(\log f_i - \log q_i)$$

(In practice, we use the log-sum-exp trick to avoid underflow.)

After all N samples:

- Compute the estimate of the integral:

$$I_{\text{numeric}} = \frac{1}{N}\sum_{i=1}^{N} w_i$$

- Compute the standard error:

$$\text{SE} = \sqrt{\left(\frac{1}{N}\sum_{i=1}^{N} w_i^2 - I_{\text{numeric}}^2\right) \Big/ N}$$

This gives us a numerical estimate **with error bars**.

1.4.6 Step 5: Compute relative error

Compare the numerical estimate with the true value:

$$\text{RelErr} = \frac{|I_{\text{numeric}} - I_{\text{analytic}}|}{I_{\text{analytic}}}$$

A small relative error means the Monte Carlo method is accurate.

1.5 Final Step: Plot

We plot the **dimension** n on the x-axis and **relative error** on the y-axis.

This shows how the quality of the numerical estimate behaves as we increase the dimension.

1.6 Summary

Step	Description
Build A	Fixed-diagonal, tridiagonal Toeplitz matrix
Draw b	From standard normal
Compute analytic I	Use known formula from Gaussian integrals
Estimate numeric I	Using importance sampling with Gaussian proposal
Compute error	Relative error between numeric and analytic values
Plot	Error vs dimension

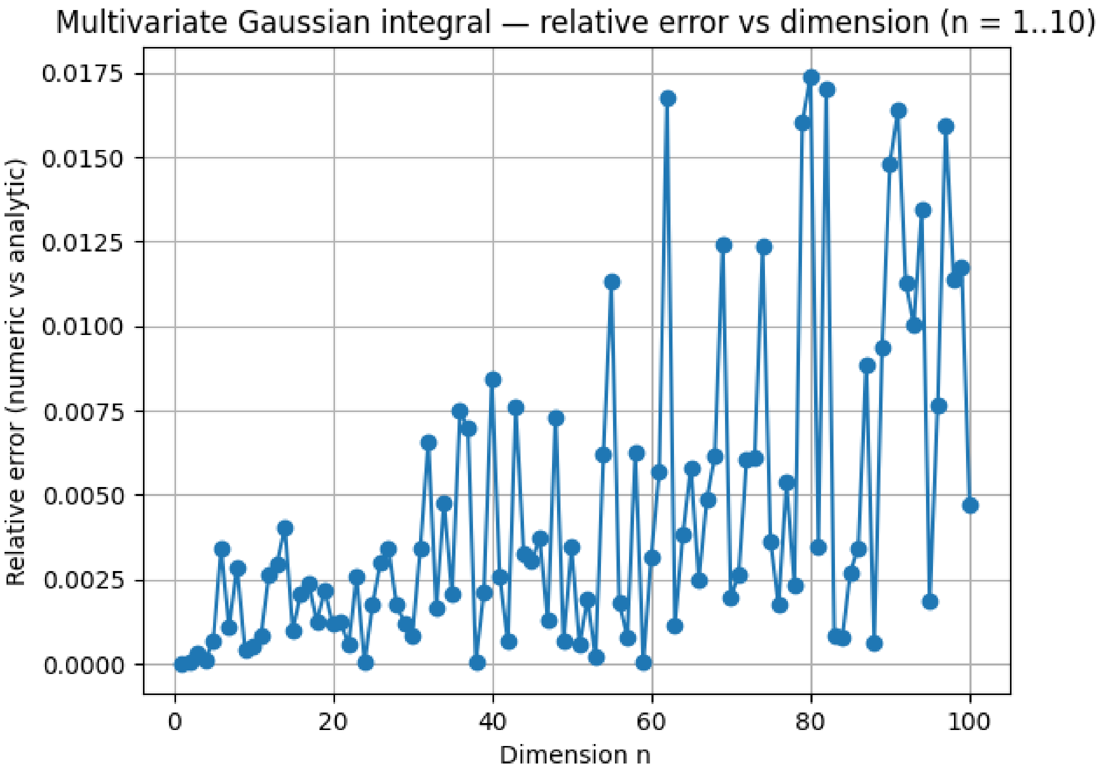
[3]:

```
import numpy as np
import matplotlib.pyplot as plt

def make_tridiagonal_toeplitz(n, diag_value=3.0, offdiag_value=0.4):
    A = np.zeros((n, n), dtype=float)
    np.fill_diagonal(A, diag_value)
    if n > 1:
        A[np.arange(n-1), np.arange(1, n)] = offdiag_value
        A[np.arange(1, n), np.arange(n-1)] = offdiag_value
    return A

def gaussian_integral_analytic(A, b):
    n = A.shape[0]
    mu = np.linalg.solve(A, b)
    quad = float(b @ mu)
    sign, logdetA = np.linalg.slogdet(A)
    if sign <= 0:
        raise ValueError("Matrix A must have positive determinant.")
    logI = 0.5 * n * np.log(2 * np.pi) - 0.5 * logdetA + 0.5 * quad
    I_value = float(np.exp(logI))
    return I_value, logI, mu

def importance_sampling_isotropic(A, b, mu, scale, N_samples, rng):
    n = A.shape[0]
    s = float(scale)
    s2 = s * s
    batch = min(20000, N_samples)
    total = 0
    sum_w = 0.0
    sum_w2 = 0.0
    while total < N_samples:
        k = min(batch, N_samples - total)
        Z = rng.standard_normal((k, n))
        X = mu + s * Z
        AX = X @ A.T
        logf = -0.5 * np.sum(X * AX, axis=1) + (X @ b)
        diff = X - mu
        quad_q = 0.5 * (1.0 / s2) * np.sum(diff * diff, axis=1)
        logq = -0.5 * n * np.log(2 * np.pi) - 0.5 * n * np.log(s2) - quad_q
        logw = logf - logq
        m = np.max(logw)
        w = np.exp(logw - m)
        sum_w  += w.sum() * np.exp(m)
        sum_w2 += np.sum(w * w) * np.exp(2 * m)
        total += k
    I_hat = sum_w / N_samples
    var_hat = max(sum_w2 / N_samples - I_hat * I_hat, 0.0)
    se = np.sqrt(var_hat / N_samples)
    return I_hat, se

def main():
    diag_value = 3.0
    offdiag_value = 0.4
    base_seed = 2025
    rng_global = np.random.default_rng(base_seed)
    proposal_scale = 1.0 / np.sqrt(diag_value)
    N_samples = 60000
    ns = list(range(1, 101))
    I_true_list = []
    I_num_list = []
    SE_list = []
    rel_err_list = []
    for n in ns:
        A = make_tridiagonal_toeplitz(n, diag_value=diag_value, offdiag_value=offdiag_value)
        b = rng_global.normal(size=n)
        I_true, logI, mu = gaussian_integral_analytic(A, b)
        rng_is = np.random.default_rng(base_seed + n)
        I_hat, se = importance_sampling_isotropic(
            A=A,
            b=b,
            mu=mu,
            scale=proposal_scale,
            N_samples=N_samples,
            rng=rng_is
        )
        rel_err = abs(I_hat - I_true) / I_true
        I_true_list.append(I_true)
        I_num_list.append(I_hat)
        SE_list.append(se)
        rel_err_list.append(rel_err)

    print(" n |      I_analytic         I_numeric(IS)        SE(IS)       ␣
 ↪RelErr")
    for n, Ia, In, se, re in zip(ns, I_true_list, I_num_list, SE_list,␣
 ↪rel_err_list):
        print(f"{n:2d} | {Ia:14.6e}   {In:14.6e}   {se:9.2e}   {re:8.3e}")

    plt.figure()
    plt.plot(ns, rel_err_list, marker='o')
    plt.xlabel("Dimension n")
    plt.ylabel("Relative error (numeric vs analytic)")
    plt.title("Multivariate Gaussian integral - relative error vs dimension (n␣
 ↪= 1..100)")
    plt.grid(True)
    plt.show()

main()
```

```
 n |      I_analytic         I_numeric(IS)        SE(IS)       RelErr
 1 |   3.293532e+00     3.293532e+00     0.00e+00   2.697e-16
 2 |   2.221701e+00     2.221857e+00     1.24e-03   7.020e-05
 3 |   1.132531e+01     1.132175e+01     9.33e-03   3.142e-04
 4 |   5.496558e+00     5.496008e+00     5.72e-03   9.994e-05
 5 |   1.289030e+01     1.289922e+01     1.58e-02   6.921e-04
 6 |   1.528187e+02     1.533375e+02     2.15e-01   3.395e-03
 7 |   5.362022e+01     5.356167e+01     7.98e-02   1.092e-03
 8 |   2.522477e+01     2.529694e+01     4.22e-02   2.861e-03
 9 |   1.759672e+02     1.758965e+02     3.08e-01   4.016e-04
10 |   3.328045e+03     3.326363e+03     6.33e+00   5.052e-04
11 |   3.227120e+02     3.224488e+02     6.45e-01   8.156e-04
12 |   5.082845e+02     5.096214e+02     1.07e+00   2.630e-03
13 |   2.227162e+03     2.233761e+03     4.96e+00   2.963e-03
14 |   2.403444e+04     2.413115e+04     5.79e+01   4.024e-03
15 |   4.934698e+03     4.929799e+03     1.17e+01   9.929e-04
16 |   9.512448e+03     9.532346e+03     2.49e+01   2.092e-03
17 |   4.340738e+04     4.351096e+04     1.13e+02   2.386e-03
18 |   1.593466e+04     1.595442e+04     4.30e+01   1.240e-03
19 |   4.177025e+04     4.167958e+04     1.17e+02   2.171e-03
20 |   3.438880e+05     3.442915e+05     1.01e+03   1.173e-03
21 |   1.504419e+05     1.506268e+05     4.42e+02   1.229e-03
22 |   1.866835e+05     1.865809e+05     5.75e+02   5.495e-04
23 |   5.917952e+05     5.902660e+05     1.84e+03   2.584e-03
24 |   2.218064e+05     2.218009e+05     7.08e+02   2.476e-05
```

```
25 |   8.585698e+04     8.570794e+04    2.97e+02   1.736e-03
26 |   1.162908e+06     1.166407e+06    4.57e+03   3.008e-03
27 |   2.660043e+06     2.669138e+06    9.70e+03   3.419e-03
28 |   2.409291e+07     2.405072e+07    8.66e+04   1.751e-03
29 |   3.283429e+06     3.279472e+06    1.20e+04   1.205e-03
30 |   2.812848e+06     2.810516e+06    1.08e+04   8.291e-04
31 |   1.121551e+07     1.125404e+07    4.31e+04   3.435e-03
32 |   2.952011e+08     2.971417e+08    1.18e+06   6.574e-03
33 |   1.251703e+08     1.253791e+08    5.50e+05   1.668e-03
34 |   7.659040e+07     7.622551e+07    3.32e+05   4.764e-03
35 |   1.534217e+08     1.531034e+08    6.34e+05   2.074e-03
36 |   3.181361e+08     3.157562e+08    1.34e+06   7.481e-03
37 |   2.215166e+08     2.199724e+08    9.31e+05   6.971e-03
38 |   4.045977e+08     4.046273e+08    1.98e+06   7.310e-05
39 |   3.287243e+08     3.280319e+08    1.49e+06   2.107e-03
40 |   1.227328e+09     1.237686e+09    6.08e+06   8.440e-03
41 |   4.546773e+08     4.558403e+08    2.21e+06   2.558e-03
42 |   9.552569e+10     9.558925e+10    4.64e+08   6.654e-04
43 |   6.876634e+10     6.824370e+10    3.15e+08   7.600e-03
44 |   2.537347e+10     2.529055e+10    1.24e+08   3.268e-03
45 |   7.648426e+09     7.625288e+09    3.91e+07   3.025e-03
46 |   1.618550e+12     1.612512e+12    8.70e+09   3.731e-03
47 |   1.092450e+11     1.091056e+11    5.87e+08   1.277e-03
48 |   1.406911e+11     1.417207e+11    7.74e+08   7.318e-03
49 |   7.995554e+10     7.990225e+10    4.33e+08   6.665e-04
50 |   2.893612e+11     2.903569e+11    1.59e+09   3.441e-03
51 |   1.413259e+12     1.414023e+12    8.20e+09   5.411e-04
52 |   1.916243e+12     1.912535e+12    1.19e+10   1.935e-03
53 |   6.694349e+11     6.693081e+11    3.87e+09   1.893e-04
54 |   2.888176e+12     2.870242e+12    1.67e+10   6.210e-03
55 |   6.467820e+12     6.540971e+12    4.35e+10   1.131e-02
56 |   2.462296e+14     2.457885e+14    1.55e+12   1.791e-03
57 |   3.320469e+14     3.317829e+14    2.19e+12   7.951e-04
58 |   8.644614e+14     8.590514e+14    5.21e+12   6.258e-03
59 |   3.139316e+13     3.139180e+13    1.98e+11   4.328e-05
60 |   3.704676e+13     3.716419e+13    2.38e+11   3.170e-03
61 |   7.980005e+15     7.934670e+15    5.05e+13   5.681e-03
62 |   1.927169e+15     1.894894e+15    1.19e+13   1.675e-02
63 |   3.818057e+15     3.813720e+15    2.72e+13   1.136e-03
64 |   8.562324e+15     8.529402e+15    5.64e+13   3.845e-03
65 |   5.011401e+14     5.040481e+14    3.51e+12   5.803e-03
66 |   4.668067e+15     4.679648e+15    3.54e+13   2.481e-03
67 |   4.721915e+15     4.698978e+15    3.26e+13   4.858e-03
68 |   1.299731e+16     1.291734e+16    8.92e+13   6.152e-03
69 |   1.271445e+18     1.255676e+18    8.78e+15   1.240e-02
70 |   4.529980e+17     4.538845e+17    3.17e+15   1.957e-03
71 |   9.224145e+17     9.199861e+17    7.02e+15   2.633e-03
72 |   5.722506e+17     5.757161e+17    4.47e+15   6.056e-03
```

```
73 |   5.067122e+17     5.036112e+17    3.68e+15   6.120e-03
74 |   2.547230e+18     2.578668e+18    2.05e+16   1.234e-02
75 |   4.452358e+17     4.436285e+17    3.38e+15   3.610e-03
76 |   1.830800e+18     1.827620e+18    1.43e+16   1.737e-03
77 |   2.528663e+18     2.542209e+18    1.96e+16   5.357e-03
78 |   4.916899e+17     4.928271e+17    4.13e+15   2.313e-03
79 |   4.598401e+18     4.672194e+18    4.10e+16   1.605e-02
80 |   1.149307e+18     1.129323e+18    8.54e+15   1.739e-02
81 |   1.643041e+20     1.648743e+20    1.43e+18   3.471e-03
82 |   1.500203e+19     1.474679e+19    1.17e+17   1.701e-02
83 |   3.605695e+17     3.608619e+17    3.79e+15   8.110e-04
84 |   1.234446e+19     1.233520e+19    1.24e+17   7.503e-04
85 |   3.265917e+20     3.274765e+20    2.87e+18   2.709e-03
86 |   1.257650e+20     1.253338e+20    1.19e+18   3.429e-03
87 |   9.734068e+19     9.820293e+19    9.11e+17   8.858e-03
88 |   4.986568e+20     4.989637e+20    4.70e+18   6.155e-04
89 |   5.604125e+20     5.656556e+20    5.07e+18   9.356e-03
90 |   1.376800e+22     1.397188e+22    1.51e+20   1.481e-02
91 |   5.755367e+21     5.849657e+21    5.74e+19   1.638e-02
92 |   1.501075e+22     1.484111e+22    1.37e+20   1.130e-02
93 |   8.523204e+22     8.608627e+22    8.44e+20   1.002e-02
94 |   1.051436e+22     1.037280e+22    9.52e+19   1.346e-02
95 |   3.761416e+22     3.754465e+22    3.64e+20   1.848e-03
96 |   9.439982e+22     9.367638e+22    9.03e+20   7.664e-03
97 |   1.562788e+21     1.537862e+21    1.41e+19   1.595e-02
98 |   4.940941e+25     4.884686e+25    5.22e+23   1.139e-02
99 |   1.061939e+25     1.074427e+25    1.21e+23   1.176e-02
100 |   8.262764e+22     8.301455e+22    8.35e+20   4.683e-03
```



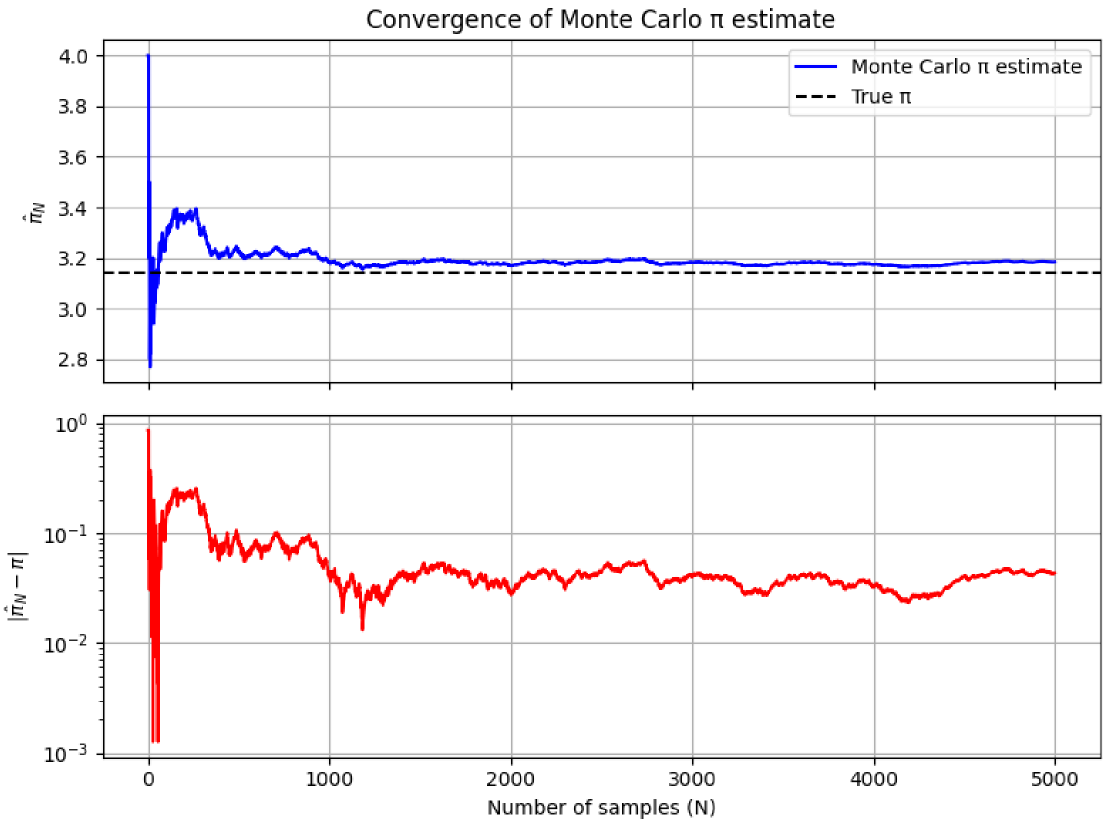
[8]:

```
import numpy as np
from math import sqrt

def pi(N, seed=12345):
    rng = np.random.default_rng(seed)

    # Generate N pairs (x, y) uniformly in [-1, 1] x [-1, 1]
    xy = rng.uniform(-1.0, 1.0, size=(N, 2))

    # Count how many fall inside the unit circle
    in_circle = np.count_nonzero(np.sum(xy**2, axis=1) <= 1.0)

    # Estimate  using   4 * (points in circle / total points)
    p_hat = in_circle / N
    pi_hat = 4.0 * p_hat

    # Standard error: SE = 4 * sqrt(p(1-p)/N)
    se = 4.0 * sqrt(p_hat * (1.0 - p_hat) / N)

    # 95% confidence interval
    ci_lower = pi_hat - 1.96 * se
    ci_upper = pi_hat + 1.96 * se
    return {
        "N": N,
        "In circle": in_circle,
        "  estimate": pi_hat,
        "Standard error (±1 )": se,
        "95% CI": (ci_lower, ci_upper)
    }

if __name__ == "__main__":
    trials = [10_000, 100_000, 1_000_000, 2_000_000]
    for N in trials:
        result = pi(N)
        print(f"N = {result['N']:,}")
        print(f"  In circle: {result['In circle']}")
        print(f"     {result['  estimate']:.6f} ± {result['Standard error␣
 ↪(±1 )']:.6f}")
        print(f"  95% CI: ({result['95% CI'][0]:.6f}, {result['95% CI'][1]:.
 ↪6f})\n")
```

```
N = 10,000
  In circle: 7883
     3.153200 ± 0.016341
  95% CI: (3.121173, 3.185227)

N = 100,000
  In circle: 78435
     3.137400 ± 0.005202
  95% CI: (3.127204, 3.147596)

N = 1,000,000
  In circle: 784595
     3.138380 ± 0.001644
  95% CI: (3.135157, 3.141603)

N = 2,000,000
  In circle: 1569055
     3.138110 ± 0.001163
  95% CI: (3.135831, 3.140389)
```

[11]:

```
import numpy as np
import matplotlib.pyplot as plt

def pi_convergence(Nmax=5000, seed=123):
    rng = np.random.default_rng(seed)
    x = rng.uniform(-1.0, 1.0, size=Nmax)
    y = rng.uniform(-1.0, 1.0, size=Nmax)
    r2 = x**2 + y**2
    inside = r2 <= 1.0

    pi_estimates = np.cumsum(inside) / np.arange(1, Nmax + 1) * 4
    true_pi = np.pi
    error = np.abs(pi_estimates - true_pi)

    fig, ax = plt.subplots(2, 1, figsize=(8, 6), sharex=True)

    ax[0].plot(pi_estimates, label='  estimate', color='blue')
    ax[0].axhline(true_pi, color='black', linestyle='--', label='True  ')
    ax[0].set_ylabel(r'$\hat{\pi}_N$')
    ax[0].set_title("Convergence of   estimate")
    ax[0].legend()
    ax[0].grid(True)

    ax[1].plot(error, label='Absolute error', color='red')
    ax[1].set_ylabel(r'$|\hat{\pi}_N - \pi|$')
    ax[1].set_xlabel("Number of samples (N)")
    ax[1].set_yscale("log")
    ax[1].grid(True)

    plt.tight_layout()
    plt.show()

plot_mc_pi_convergence()
```



1.6.1 Estimation of Using Truncated Normal Sampling

In this method, we estimate the value of using **Monte Carlo sampling**, but instead of drawing uniformly from the square $[-1,1]^2$, we sample points from a **normal distribution**:

$$x, y \sim \mathcal{N}(0, \sigma^2), \quad \text{with } \sigma = 0.5$$

We then **accept only those points** that fall within the square $[-1,1]^2$, i.e., we apply **rejection sampling**.

Unlike uniform sampling, where all regions of the square are equally likely, the normal distribution clusters points **closer to the origin** $(0,0)$, which lies **inside the unit circle**. This creates a non-uniform sampling density that affects:

- **More points fall inside the circle early on**, improving initial convergence.
- However, **fewer points sample the edges** of the square, making the estimator **more biased** if not enough points are used.
- Overall convergence to is still guaranteed by the **law of large numbers**, but it may **converge more slowly** than uniform sampling due to this skewed distribution.
- This method still estimates by:

$$\hat{\pi}_N = 4 \cdot \frac{\#\text{points inside circle}}{\#\text{total accepted points}}$$

- But the accuracy and convergence depend on how the sampling distribution populates the square.
- Normal distributions **require more samples** to compensate for non-uniformity at the boundaries of the square.

[ ]:

```
```