In [5]:
```python
import numpy as np
from scipy.integrate import nquad
import numpy as np
from scipy.linalg import cholesky, solve
from scipy.stats import multivariate_normal
```

In [15]:
```python
# Prob 1(a)
# This Method, after scolding GPT, I found it to be way better than the nquad, with grea
def compute_numeric_integral(A, w, num_samples=10000):
    """
    Computes the integral using importance sampling with Monte Carlo estimation.

    Parameters:
    - A: NxN positive definite matrix
    - w: N-dimensional vector
    - num_samples: Number of Monte Carlo samples to use

    Returns:
    - Approximated integral value
    """
    A = np.array(A, dtype=np.float64)
    w = np.array(w, dtype=np.float64).flatten()
    N = len(w)

    # Compute A⁻¹ using Cholesky decomposition (O(N²))
    L = cholesky(A, lower=True)   # A = LL^T
    A_inv = solve(A, np.eye(N))   # More efficient than np.linalg.inv(A)

    # Generate samples from multivariate Gaussian centered at A⁻¹w
    mean = A_inv @ w
    cov = A_inv
    samples = np.random.multivariate_normal(mean, cov, size=num_samples)

    # Compute function values at sampled points
    quad_term = -0.5 * np.einsum('ij,ji->i', samples @ A, samples.T)
    linear_term = np.dot(samples, w)
    integrand_values = np.exp(quad_term + linear_term)

    # Compute probability density of samples under the proposal distribution
    proposal_pdf = multivariate_normal.pdf(samples, mean=mean, cov=cov)

    # Compute the Monte Carlo estimate of the integral
    integral_estimate = np.mean(integrand_values / proposal_pdf)
    return integral_estimate

def compute_closed_form(A, w):
    """
    Computes the closed-form solution of the Gaussian integral.

    Parameters:
    - A: NxN positive definite matrix
    - w: N-dimensional vector

    Returns:
    - Exact closed-form result
    """
    A = np.array(A, dtype=np.float64)
    w = np.array(w, dtype=np.float64).flatten()
    N = len(w)

    # Solve A⁻¹w efficiently using Cholesky decomposition
    L = cholesky(A, lower=True)
    A_inv_w = solve(A, w)
```

```python
    # Compute determinant efficiently
    det_A = np.prod(np.diag(L))**2  # Since det(A) = (det(L))^2

    # Compute normalization factor
    normalization = np.sqrt(((2 * np.pi) ** N) / det_A)

    # Compute quadratic form
    quadratic_form = 0.5 * np.dot(w, A_inv_w)

    return normalization * np.exp(quadratic_form)
```

In [16]:

```python
# Prob 1(b)
A = [[4,2, 1],
     [2,5, 3],
     [1,3,6]]
w = [1, 2,3]

print("For A matrix: ")
result_numeric = compute_numeric_integral(A, w, num_samples=10000)
print(f"Numerical 3D Integral (Monte Carlo): {result_numeric:.6f}")

result_closed = compute_closed_form(A, w)
print(f"Closed-form result: {result_closed:.6f}")

error = abs(result_numeric - result_closed)
print(f"Error between Monte Carlo and closed-form: {error:.6e}")


A_prime = [[4,2, 1],
     [2,1, 3],
     [1,3,6]]
w = [1, 2,3]

print("For A prime matrix: ")
result_numeric = compute_numeric_integral(A_prime, w, num_samples=10000)
print(f"Numerical 3D Integral (Monte Carlo): {result_numeric:.6f}")

result_closed = compute_closed_form(A_prime, w)
print(f"Closed-form result: {result_closed:.6f}")

error = abs(result_numeric - result_closed)
print(f"Error between Monte Carlo and closed-form: {error:.6e}")
```

```
For A matrix:
Numerical 3D Integral (Monte Carlo): 4.275824
Closed-form result: 4.275824
Error between Monte Carlo and closed-form: 8.881784e-16
For A prime matrix:
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_24240\666776887.py in <cell line: 0>()
     22
     23 print("For A prime matrix: ")
---> 24 result_numeric = compute_numeric_integral(A_prime, w, num_samples=10000)
     25 print(f"Numerical 3D Integral (Monte Carlo): {result_numeric:.6f}")
     26

~\AppData\Local\Temp\ipykernel_24240\767848007.py in compute_numeric_integral(A, w, num_
samples)
     18
     19     # Compute A⁻¹ using Cholesky decomposition (O(N²))
---> 20     L = cholesky(A, lower=True)  # A = LL^T
     21     A_inv = solve(A, np.eye(N))  # More efficient than np.linalg.inv(A)
     22
```

```
c:\Users\Eric\AppData\Local\Programs\Python\Python313\Lib\site-packages\scipy\linalg\_de
comp_cholesky.py in cholesky(a, lower, overwrite_a, check_finite)
     99
    100     """
--> 101     c, lower = _cholesky(a, lower=lower, overwrite_a=overwrite_a, clean=True,
    102                          check_finite=check_finite)
    103     return c

c:\Users\Eric\AppData\Local\Programs\Python\Python313\Lib\site-packages\scipy\linalg\_de
comp_cholesky.py in _cholesky(a, lower, overwrite_a, clean, check_finite)
     36     c, info = potrf(a1, lower=lower, overwrite_a=overwrite_a, clean=clean)
     37     if info > 0:
---> 38         raise LinAlgError("%d-th leading minor of the array is not positive "
     39                           "definite" % info)
     40     if info < 0:

LinAlgError: 2-th leading minor of the array is not positive definite
```

As we can see, for A prime matrix, the integral is not positive definite

In [24]:

```python
import numpy as np
from scipy.linalg import cholesky, solve_triangular, inv
from scipy.integrate import nquad

# Define matrix A and vector w
A = np.array([[4, 2, 1],
              [2, 5, 3],
              [1, 3, 6]])

w = np.array([1, 2, 3])
Sigma = inv(A)
mu = Sigma @ w





# <v1> <v2> <v3>
# According to the Wick Theorem, closed form is <vi> = mu_i, where mu is a vector of A^-
def monte_carlo_vi(A, w, num_samples=5000000):
    A_inv = solve(A, np.eye(A.shape[0]))   # Efficient way to get A⁻¹
    mean = A_inv @ w
    cov = A_inv
    samples = np.random.multivariate_normal(mean, cov, size=num_samples)
    vi_estimates = np.mean(samples, axis=0)   # Expectation values
    return vi_estimates
mc_vi = monte_carlo_vi(A, w)
print("\nComparing Numerical and Closed-form Solutions:")
for i in range(3):
    print(f"<v{i+1}>: Numerical = {mc_vi[i]:.6f}, Closed-form = {mu[i]:.6f}")


# <v1v2> <v2v3> <v1v3>
# According to the Wick Theorem,
# closed form is <vi vj> = A^-1_{ij} + mu_i*mu_j

def compute_closed_form_vivj(Sigma, mu):
    vivj = np.zeros((3, 3))   # 3x3 matrix for all <vi vj>
    for i in range(3):
        for j in range(3):
            vivj[i, j] = Sigma[i, j] + mu[i] * mu[j]
    return vivj

closed_vivj = compute_closed_form_vivj(Sigma, mu)
def monte_carlo_vivj(A, w, num_samples=5000000):
```

```
    A_inv = solve(A, np.eye(A.shape[0]))  # Efficient A⁻¹
    mean = A_inv @ w
    cov = A_inv
    samples = np.random.multivariate_normal(mean, cov, size=num_samples)
    vivj_estimates = np.einsum('ij,ik->jk', samples, samples) / num_samples
    return vivj_estimates
mc_vivj = monte_carlo_vivj(A, w)
print("\nComparing Numerical and Closed-form Solutions for <vi vj>:")
for i, j in [(0, 1), (1, 2), (0, 2)]:
    print(f"<v{i+1} v{j+1}>: Numerical = {mc_vivj[i, j]:.6f}, Closed-form = {closed_vivj


# <v1^2v2>, <v3^2 v2>
# Closed form for <vi^2vj> = 2 mui A^-1ij + muj A^-1ii + mui^2 muj

def compute_closed_form_vi2vj(Sigma, mu):
    vi2vj = np.zeros((3, 3))
    for i in range(3):
        for j in range(3):
            vi2vj[i, j] = 2 * mu[i] * Sigma[i, j] + mu[j] * Sigma[i, i] + mu[i]**2 * mu[
    return vi2vj

closed_vi2vj = compute_closed_form_vi2vj(Sigma, mu)
def monte_carlo_vi2vj(A, w, num_samples=5000000):
    A_inv = Sigma # Efficient A⁻¹
    mean = A_inv @ w
    cov = A_inv
    samples = np.random.multivariate_normal(mean, cov, size=num_samples)
    vi2vj_estimates = np.zeros((3, 3))
    for i in range(3):
        for j in range(3):
            vi2vj_estimates[i, j] = np.mean(samples[:, i]**2 * samples[:, j])

    return vi2vj_estimates
mc_vi2vj = monte_carlo_vi2vj(A, w)
print("\nComparing Numerical and Closed-form Solutions for <vi^2 vj>:")
for i, j in [(0, 1), (2, 1)]:  # (v1^2 v2), (v3^2 v2)
    print(f"<v{i+1}^2 v{j+1}>: Numerical = {mc_vi2vj[i, j]:.6f}, Closed-form = {closed_v


# <v1^2 v2^2> <v2^2 v3^3>
#  Closed form for <vi^2vj^2> = (A^-1 ii + mui^2)(A^-1jj+muj^2)+2 A^-1ij^2

def compute_closed_form_vi2vj2(Sigma, mu):
    vi2vj2 = np.zeros((3, 3))  # 3x3 matrix for all <vi^2 vj^2>
    for i in range(3):
        for j in range(3):
            vi2vj2[i, j] = (Sigma[i, i] + mu[i]**2) * (Sigma[j, j] + mu[j]**2) + 2 * Sig
    return vi2vj2

closed_vi2vj2 = compute_closed_form_vi2vj2(Sigma, mu)

def monte_carlo_vi2vj2(A, w, num_samples=5000000):
    A_inv = solve(A, np.eye(A.shape[0]))  # Efficient A⁻¹
    mean = A_inv @ w
    cov = A_inv
    samples = np.random.multivariate_normal(mean, cov, size=num_samples)
    vi2vj2_estimates = np.zeros((3, 3))
    for i in range(3):
        for j in range(3):
            vi2vj2_estimates[i, j] = np.mean(samples[:, i]**2 * samples[:, j]**2)

    return vi2vj2_estimates

mc_vi2vj2 = monte_carlo_vi2vj2(A, w)
print("\nComparing Numerical and Closed-form Solutions for <vi^2 vj^2>:")
```

```python
for i, j in [(0, 1), (1, 2)]:  # (v1^2 v2^2), (v2^2 v3^2)
    print(f"<v{i+1}^2 v{j+1}^2>: Numerical = {mc_vi2vj2[i, j]:.6f}, Closed-form = {close
```

```
Comparing Numerical and Closed-form Solutions:
<v1>: Numerical = 0.089242, Closed-form = 0.089552
<v2>: Numerical = 0.104743, Closed-form = 0.104478
<v3>: Numerical = 0.432627, Closed-form = 0.432836

Comparing Numerical and Closed-form Solutions for <vi vj>:
<v1 v2>: Numerical = -0.124967, Closed-form = -0.124972
<v2 v3>: Numerical = -0.103951, Closed-form = -0.104032
<v1 v3>: Numerical = 0.053431, Closed-form = 0.053687

Comparing Numerical and Closed-form Solutions for <vi^2 vj>:
<v1^2 v2>: Numerical = 0.009465, Closed-form = 0.009526
<v3^2 v2>: Numerical = -0.084691, Closed-form = -0.084681

Comparing Numerical and Closed-form Solutions for <vi^2 vj^2>:
<v1^2 v2^2>: Numerical = 0.145074, Closed-form = 0.149946
<v2^2 v3^2>: Numerical = 0.168467, Closed-form = 0.195496
```

In [ ]: