Question 1 – Transformation Method and Rejection Method

(a) Normalization Constant A

We are given the probability density function:

$$P(x) = A\cos x, \quad \text{for } x \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right].$$

To normalize this PDF, we require:

$$\int_{-\pi/2}^{\pi/2} A\cos x\,dx = 1.$$

Evaluating the integral:

$$A\int_{-\pi/2}^{\pi/2} \cos x\,dx = A[\sin x]_{-\pi/2}^{\pi/2} = A(1-(-1)) = 2A.$$

So,

$$2A = 1 \quad \Rightarrow \quad A = \frac{1}{2}.$$

(b) Sampling with the Transformation Method

Cumulative Distribution Function (CDF)

The cumulative distribution function is:

$$F(x) = \int_{-\pi/2}^{x} \frac{1}{2}\cos t\,dt = \frac{1}{2}(\sin x + 1).$$

Inverse Transform Sampling

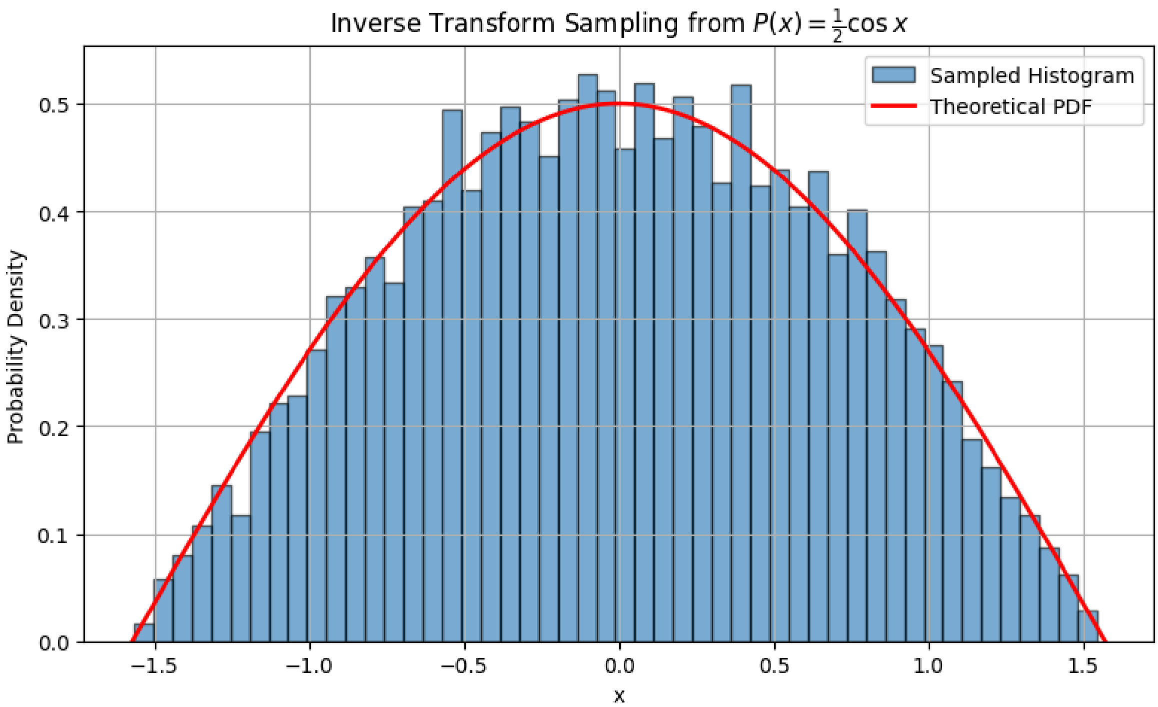
Let $u \sim \mathcal{U}(0,1)$. Setting $F(x) = u$ yields:

$$\frac{1}{2}(\sin x + 1) = u \quad \Rightarrow \quad \sin x = 2u - 1 \quad \Rightarrow \quad x = \arcsin(2u-1).$$

Python Code to Generate Samples and Plot Histogram

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

# Define normalization factor for the PDF P(x) = A * cos(x)
normalization_factor = 0.5

# Set number of random samples to draw
```

```
num_samples = 10000

# Draw uniform random numbers from U(0, 1)
uniform_samples = np.random.rand(num_samples)

# Apply inverse CDF transformation: x = arcsin(2u − 1)
transformed_samples = np.arcsin(2 * uniform_samples - 1)

# Create histogram of the sampled data
plt.figure(figsize=(8, 5))
plt.hist(transformed_samples,
         bins=50,
         density=True,
         alpha=0.6,
         edgecolor='black',
         label='Sampled Histogram')

# Define the theoretical PDF for comparison
x_vals = np.linspace(-np.pi / 2, np.pi / 2, 300)
pdf_vals = normalization_factor * np.cos(x_vals)

# Plot the theoretical PDF curve
plt.plot(x_vals, pdf_vals,
         color='red',
         linewidth=2,
         label='Theoretical PDF')

# Annotate the plot
plt.xlabel('x')
plt.ylabel('Probability Density')
plt.title('Inverse Transform Sampling from $P(x) = \\frac{1}{2}\\cos x$')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



Plot Description

- The histogram shows the distribution of 10,000 samples drawn using the inverse transform method.
- The red curve overlays the theoretical PDF $P(x) = \frac{1}{2}\cos x$.
- The match between the histogram and the red curve validates the sampling method.

(c) Why Transformation Method Does Not Work

Consider the PDF:

$$P(x) = \frac{3}{5}\left(1 - \frac{x^2}{2}\right), \quad x \in [-1, 1].$$

The cumulative distribution function is:

$$F(x) = \int_{-1}^{x} \frac{3}{5}\left(1 - \frac{t^2}{2}\right) dt = \frac{3}{5}\left(x - \frac{x^3}{6} + \frac{5}{6}\right).$$

To perform inverse transform sampling, we must solve:

$$F(x) = u \quad \Rightarrow \quad x - \frac{x^3}{6} + \frac{5}{6} = \frac{5}{3}u,$$

which is a cubic equation in x. Without access to a general formula for solving cubic equations, we cannot invert $F(x)$ analytically. Therefore, the transformation method is not applicable.

(d) Rejection Method Implementation

We are given the inequality:

$$\cos x \geq 1 - \frac{x^2}{2}, \quad \text{for } x \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right].$$

This suggests that the function $f(x) = \frac{3}{5}\cos x$ can serve as an envelope that satisfies $f(x) \geq P(x)$ on the interval $[-1, 1]$.

Proposal Distribution $g(x)$

The normalizing constant for $f(x)$ is:

$$C = \int_{-1}^{1} \frac{3}{5}\cos x \, dx = \frac{6}{5}\sin 1.$$

Hence, the normalized proposal PDF is:

$$g(x) = \frac{f(x)}{C} = \frac{\cos x}{2\sin 1}, \quad x \in [-1, 1].$$

Acceptance Probability

A sample $x \sim g(x)$ is accepted with probability:

$$\frac{P(x)}{f(x)} = \frac{1 - \frac{x^2}{2}}{\cos x}.$$

Theoretical Acceptance Rate

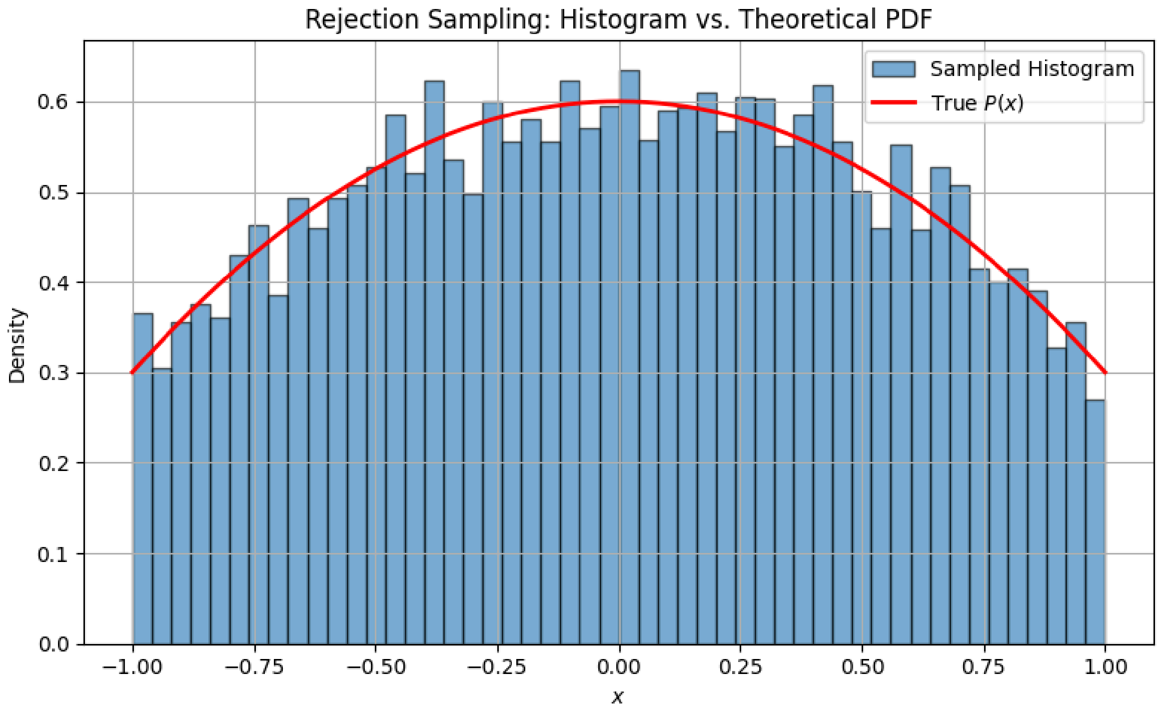
The expected acceptance rate is the ratio of the areas under the two curves:

$$\text{Acceptance Rate} = \frac{\int_{-1}^{1} P(x)dx}{\int_{-1}^{1} f(x)dx} = \frac{1}{C} = \frac{5}{6\sin 1} \approx 0.9903.$$

In [2]:

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

# Define the target PDF and the envelope function
def target_pdf(x):
    return (3 / 5) * (1 - x**2 / 2)

def envelope_function(x):
    return (3 / 5) * np.cos(x)

# Desired number of accepted samples
num_accepted = 10000

# Normalization constant of envelope: ∫_{-1}^{1} f(x) dx = (6/5) * sin(1)
sin_1 = np.sin(1.0)
normalization_constant = (6 / 5) * sin_1
theoretical_acceptance_rate = 1 / normalization_constant

# Rejection sampling loop
accepted_values = []
total_trials = 0

while len(accepted_values) < num_accepted:
    batch_size = num_accepted - len(accepted_values)

    # Step 1: Sample from proposal distribution g(x) ∝ cos(x)
    u_sample = np.random.rand(batch_size)
    x_trial = np.arcsin(sin_1 * (2 * u_sample - 1))

    # Step 2: Accept with probability P(x)/f(x)
    u_accept = np.random.rand(batch_size)
    is_accepted = u_accept <= (target_pdf(x_trial) / envelope_function(x_

    accepted_values.extend(x_trial[is_accepted].tolist())
    total_trials += batch_size

# Finalize sample array and compute empirical acceptance rate
samples = np.array(accepted_values[:num_accepted])
empirical_acceptance_rate = num_accepted / total_trials

# Plot: histogram of accepted samples vs. true PDF
plt.figure(figsize=(8, 5))
plt.hist(samples, bins=50, density=True, alpha=0.6, edgecolor='black',
         label='Sampled Histogram')
```

```python
x_plot = np.linspace(-1, 1, 300)
plt.plot(x_plot, target_pdf(x_plot), 'r-', linewidth=2, label='True $P(x)
plt.xlabel('$x$')
plt.ylabel('Density')
plt.title('Rejection Sampling: Histogram vs. Theoretical PDF')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Print summary statistics
print(f"Total proposals generated:     {total_trials}")
print(f"Number of samples accepted:    {num_accepted}")
print(f"Observed acceptance rate:      {empirical_acceptance_rate:.4f}")
print(f"Theoretical acceptance rate:   {theoretical_acceptance_rate:.4f}"
```




```
Total proposals generated:     10094
Number of samples accepted:    10000
Observed acceptance rate:      0.9907
Theoretical acceptance rate:   0.9903
```

Conclusion

The observed acceptance fraction is very close to the theoretical acceptance rate:

$$\frac{5}{6\sin 1} \approx 0.9903.$$

This confirms the correctness and efficiency of the rejection sampling method using the cosine envelope.

Question 2 – Integration via Deterministic and Monte Carlo Methods

---

(a) Change of Variable to Finite Interval

We consider the substitution:

$$x = \tan z, \quad \text{with } z \in \left(-\frac{\pi}{2}, \frac{\pi}{2}\right),$$

so that:

$$dx = \sec^2 z \, dz.$$

Under this transformation, the original integral becomes:

$$I = \int_{-\infty}^{\infty} e^{-x^4} dx = \int_{-\pi/2}^{\pi/2} e^{-\tan^4 z} \cdot \sec^2 z \, dz.$$

We define the new integrand:

$$g(z) = e^{-\tan^4 z} \cdot \sec^2 z, \quad \text{with limits } a = -\frac{\pi}{2},\ b = \frac{\pi}{2}.$$

Endpoint behavior and analytic limits

As $z \to \frac{\pi}{2}^-$, we observe that:

$$\tan z \to +\infty, \quad \sec^2 z \to +\infty, \quad \Rightarrow \quad g(z) = e^{-\tan^4 z} \cdot \sec^2 z = \infty \cdot 0,$$

which is an indeterminate form. Therefore, we compute the limit analytically.

Let:

$$t = \frac{\pi}{2} - z \quad \Rightarrow \quad z = \frac{\pi}{2} - t, \quad \text{with } t \to 0^+.$$

Then we have the asymptotic expansions:

$$\tan z = \cot t = \frac{1}{t} - \frac{t}{3} + \mathcal{O}(t^3), \quad \Rightarrow \quad \tan^4 z = \left(\frac{1}{t} - \frac{t}{3} + \cdots\right)^4 = \frac{1}{t^4} + \mathcal{O}\left(\frac{1}{t}\right.$$

Also:

$$\sec^2 z = 1 + \tan^2 z = 1 + \left(\frac{1}{t} - \frac{t}{3} + \cdots\right)^2 = \frac{1}{t^2} + \mathcal{O}(1).$$

Putting these together:

$$g(z) = e^{-\tan^4 z} \cdot \sec^2 z = \left[e^{-1/t^4 + \mathcal{O}(1/t^2)}\right] \cdot \left[\frac{1}{t^2} + \mathcal{O}(1)\right].$$

We now use the classical result that:

$$\lim_{t \to 0^+} \frac{e^{-1/t^4}}{t^n} = 0 \quad \text{for any } n > 0,$$

which implies that:

$$\lim_{t \to 0^+} g(z) = 0.$$

A symmetric argument applies as $z \to -\frac{\pi}{2}^+$,
by letting $t = \frac{\pi}{2} + z \to 0^+$, or by symmetry of the integrand.

Final conclusion

The substitution transforms the original improper integral to a definite integral:

$$I = \int_a^b g(z)\, dz, \quad \text{with } g(z) = e^{-\tan^4 z} \cdot \sec^2 z, \quad a = -\frac{\pi}{2},\ b = \frac{\pi}{2},$$

where the integrand satisfies:

$$\boxed{\lim_{z \to a^+} g(z) = \lim_{z \to b^-} g(z) = 0.}$$

Thus, $g(z)$ is continuous on the open interval and vanishes at both endpoints, and the transformed integral is proper and well-defined.

(b)(c)(d)(d)

In [3]:

```
import numpy as np
from numpy.polynomial.legendre import leggauss

# Target integrand after change of variable: g(z) = e^{-tan^4 z} * sec^2
def g(z):
    return np.exp(-np.tan(z)**4) / np.cos(z)**2

# Composite midpoint rule
def midpoint_rule(f, a, b, N):
    h = (b - a) / N
    midpoints = a + (np.arange(N) + 0.5) * h
    return h * np.sum(f(midpoints))

# Adaptive midpoint integration with tripling
def integrate_midpoint_tripling(f, a, b, tol=1e-6, max_iter=10):
    N = 1
    estimate_prev = midpoint_rule(f, a, b, N)
    for _ in range(max_iter):
        N *= 3
        estimate_curr = midpoint_rule(f, a, b, N)
        rel_err = abs(estimate_curr - estimate_prev) / abs(estimate_curr)
        if rel_err <= tol:
            return estimate_curr, N, rel_err
        estimate_prev = estimate_curr
    return estimate_curr, N, rel_err

# Romberg integration using midpoint rule and tripling
def romberg_midpoint_tripling(f, a, b, tol=1e-6, max_levels=10):
    R = []
    for level in range(max_levels):
        N = 3**level
        row = [midpoint_rule(f, a, b, N)]
```

```
        for k in range(1, level + 1):
            factor = 3**(2 * k)
            extrapolated = (factor * row[k - 1] - R[level - 1][k - 1]) /
            row.append(extrapolated)
        R.append(row)
        if level > 0:
            rel_err = abs(R[level][level] - R[level - 1][level - 1]) / ab
            if rel_err <= tol:
                return R[level][level], level + 1, rel_err
    return R[-1][-1], max_levels, rel_err

# Composite trapezoidal rule
def trapezoidal_rule(f, a, b, N):
    h = (b - a) / N
    x = np.linspace(a, b, N + 1)
    y = f(x)
    return h * (0.5 * y[0] + y[1:-1].sum() + 0.5 * y[-1])

# Adaptive trapezoidal rule with doubling
def integrate_trapezoidal_doubling(f, a, b, tol=1e-6, max_iter=20):
    N = 1
    estimate_prev = trapezoidal_rule(f, a, b, N)
    for _ in range(max_iter):
        N *= 2
        estimate_curr = trapezoidal_rule(f, a, b, N)
        rel_err = abs(estimate_curr - estimate_prev) / abs(estimate_curr)
        if rel_err <= tol:
            return estimate_curr, N, rel_err
        estimate_prev = estimate_curr
    return estimate_curr, N, rel_err

# Gaussian quadrature over [a, b] using n-point Legendre rule
def gaussian_quadrature(f, a, b, n):
    nodes, weights = leggauss(n)
    x_mapped = 0.5 * (b + a) + 0.5 * (b - a) * nodes
    return 0.5 * (b - a) * np.dot(weights, f(x_mapped))

# Adaptive Gaussian quadrature with error control
def integrate_gauss_adaptive(f, a, b, tol=1e-6, max_n=1024):
    n = 4
    estimate_prev = gaussian_quadrature(f, a, b, n)
    while n <= max_n:
        n *= 2
        estimate_curr = gaussian_quadrature(f, a, b, n)
        rel_err = abs(estimate_curr - estimate_prev) / abs(estimate_curr)
        if rel_err <= tol:
            return estimate_curr, n, rel_err
        estimate_prev = estimate_curr
    return estimate_curr, n, rel_err

# Integration bounds after substitution: z ∈ (−π/2, π/2)
a, b = -0.5 * np.pi, 0.5 * np.pi

# Midpoint Rule
mid_result, mid_panels, mid_error = integrate_midpoint_tripling(g, a, b)
print(f"[Midpoint Rule]       I ≈ {mid_result:.8f} | Panels: {mid_panels:

# Romberg Extrapolation (Midpoint + Tripling)
rom_result, rom_levels, rom_error = romberg_midpoint_tripling(g, a, b)
print(f"[Romberg Midpoint]    I ≈ {rom_result:.8f} | Levels: {rom_levels:
```

```
# Trapezoidal Rule
trap_result, trap_panels, trap_error = integrate_trapezoidal_doubling(g,
print(f"[Trapezoidal Rule]    I ≈ {trap_result:.8f} | Panels: {trap_panel

# Gaussian Quadrature
gauss_result, gauss_nodes, gauss_error = integrate_gauss_adaptive(g, a, b
print(f"[Gaussian Quadrature] I ≈ {gauss_result:.8f} | Nodes:  {gauss_nod
```

```
[Midpoint Rule]       I ≈ 1.81280495 | Panels: 243 | Rel. Error: 3.65e-12
[Romberg Midpoint]    I ≈ 1.81280495 | Levels:   7 | Rel. Error: 6.90e-08
[Trapezoidal Rule]    I ≈ 1.81280495 | Panels:  128 | Rel. Error: 3.00e-10
[Gaussian Quadrature] I ≈ 1.81280495 | Nodes:   128 | Rel. Error: 2.54e-09
```

(f) Monte Carlo Integration in 4D

We estimate the 4-dimensional integral:

$$I = \int_{\mathbb{R}^4} \frac{e^{-(x_1^4+x_2^4+x_3^4+x_4^4)}}{1+x_1^2+x_2^2+x_3^2+x_4^2}\, dx_1 dx_2 dx_3 dx_4$$

We use importance sampling with the unnormalized density:

$$w(x) \propto e^{-\sum x_i^4}$$

which gives:

$$I = Z \cdot \mathbb{E}_{x\sim w}\left[\frac{1}{1+\sum x_i^2}\right] \quad \text{where} \quad Z = \left(\int_{-\infty}^{\infty} e^{-x^4}dx\right)^4 = \left(\frac{1}{2}\Gamma\left(\frac{1}{4}\right)\right)^4$$

We sample 10^6 points from each 1D marginal of $w(x)$ using rejection sampling with standard normal proposals.

In [14]:
```
import numpy as np
from math import gamma, sqrt, pi

# --------------------------------
# 1. Normalization Constant Z
# --------------------------------
Z_1d = 0.5 * gamma(0.25)
Z_4d = Z_1d ** 4  # Because w(x) factorizes over 4D

# --------------------------------
# 2. Rejection Sampling for 1D ∝ exp(−x⁴)
# --------------------------------

def sample_exp_neg_x4(n):
    """
    Generate samples from the 1D density ∝ exp(-x^4)
    using rejection sampling with N(0,1) proposals.
    """
    accepted = []
    while len(accepted) < n:
        batch = int((n - len(accepted)) * 1.2)  # Slight oversampling
        proposals = np.random.randn(batch)
        # Compute log of acceptance ratio
        log_accept = -proposals**4 + 0.5 * proposals**2 - 1/16
```

```python
        accept = np.random.rand(batch) < np.exp(log_accept)
        accepted.extend(proposals[accept])
    return np.array(accepted[:n])

# ------------------------------
# 3. Monte Carlo Integration in 4D
# ------------------------------

N = 10**6  # number of Monte Carlo samples

# Generate independent samples from w(x) ∝ exp(-x^4)
X = [sample_exp_neg_x4(N) for _ in range(4)]

# Compute h(x) = 1 / (1 + x1^2 + x2^2 + x3^2 + x4^2)
r_squared = sum(x**2 for x in X)
h_vals = 1 / (1 + r_squared)

# Monte Carlo estimate and standard error
h_mean = h_vals.mean()
h_std  = h_vals.std(ddof=1)

I_est = Z_4d * h_mean
I_se  = Z_4d * h_std / np.sqrt(N)

# ------------------------------
# 4. Output results
# ------------------------------
print("[Monte Carlo Integration (4D)]")
print(f"Estimated Integral     I ≈ {I_est:.6f}")
print(f"Standard Error         ≈ {I_se:.6f} (1σ)")
print(f"Relative Error         ≈ {I_se / I_est:.2e}")
```

```
[Monte Carlo Integration (4D)]
Estimated Integral     I ≈ 5.048876
Standard Error         ≈ 0.001578 (1σ)
Relative Error         ≈ 3.13e-04
```