Problem Set 1: Gaussians and Visualization

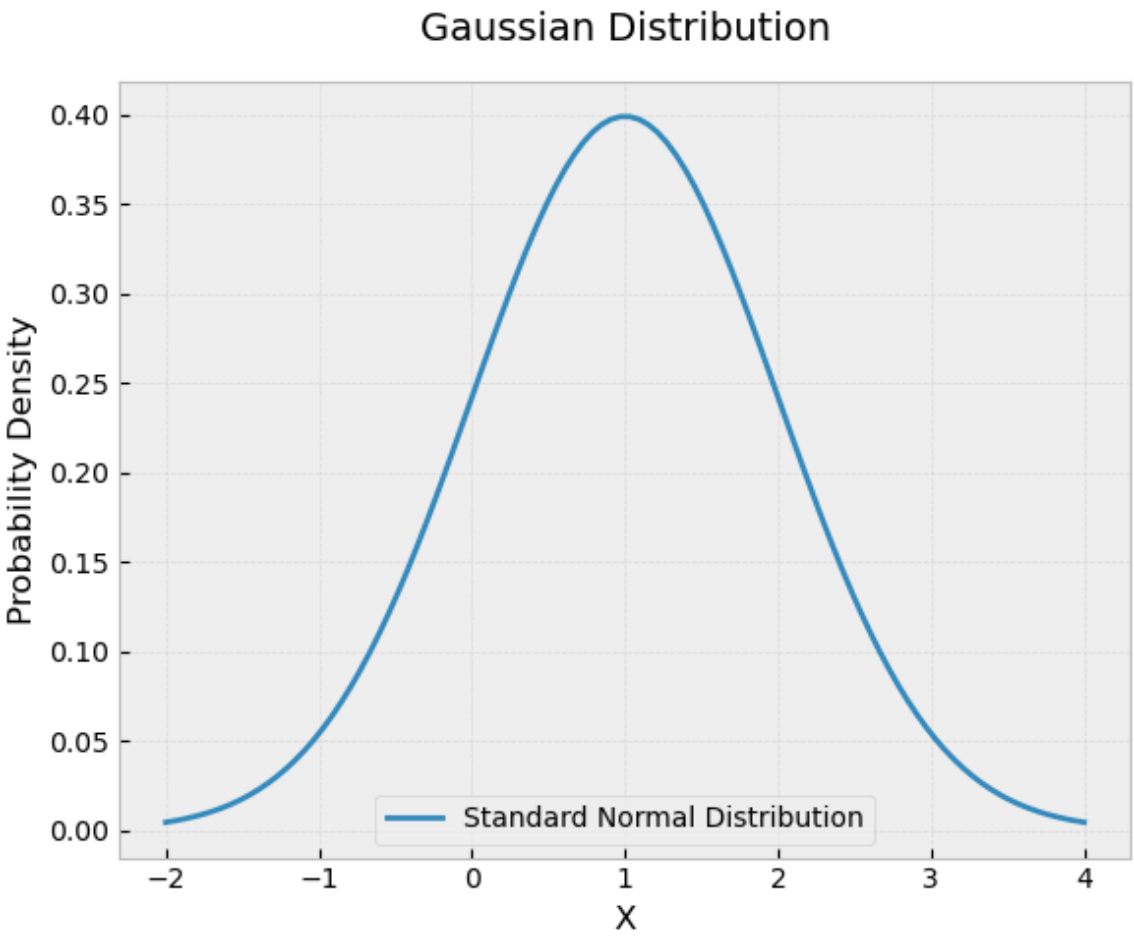
Made by Denis Fatykhoph

Task 1: Probability

A. Plot the probability density function $p(x)$ of a one dimensional Gaussian distribution $\mathcal{N}(x;1,1)$

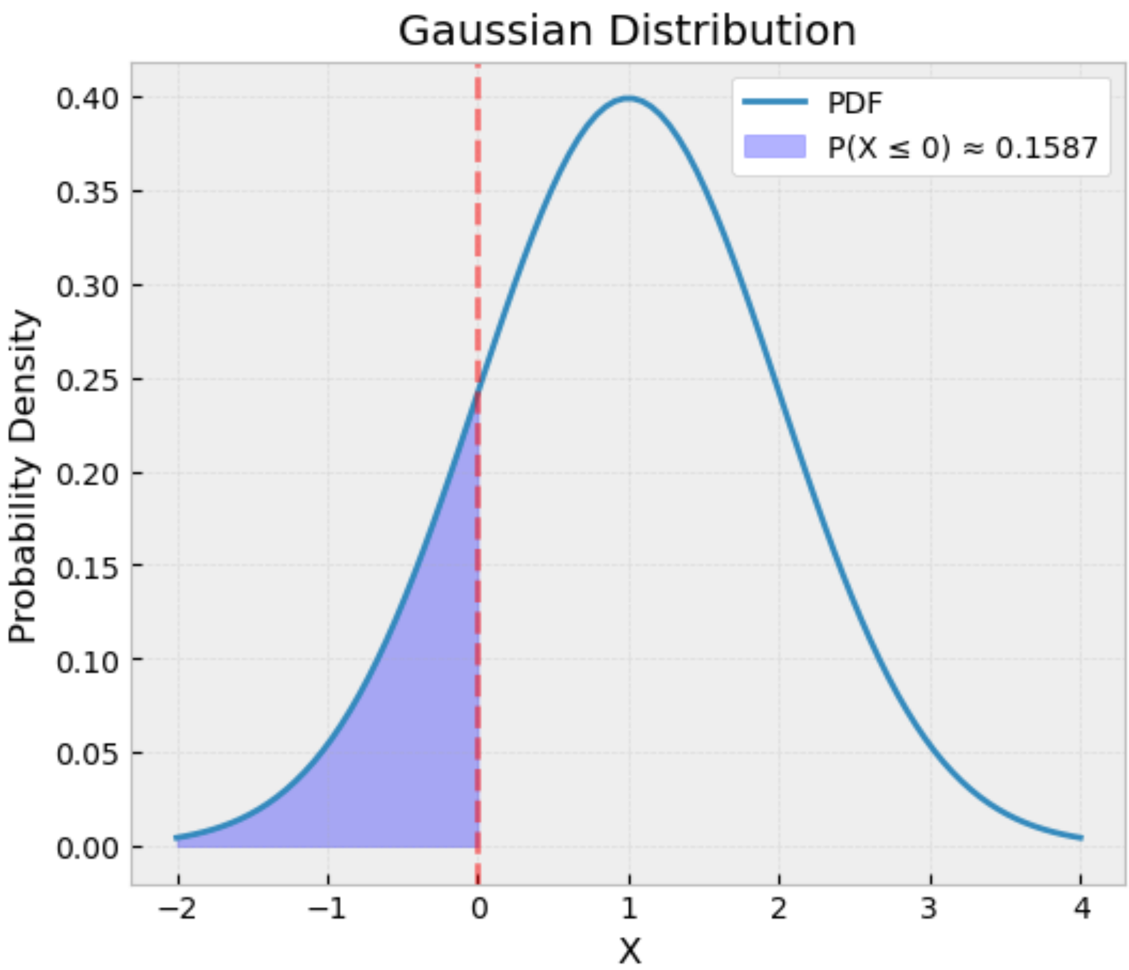
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats
import seaborn as sns
```

In [2]:
```
plt.style.use('bmh')  # or 'ggplot', 'bmh', 'classic'

x = np.linspace(-2, 4, 100)
dist_pdf = stats.norm.pdf(x , 1, 1)

plt.figure(figsize=(6, 5))
plt.plot(x, dist_pdf, linewidth=2, label='Standard Normal Distribut

plt.title('Gaussian Distribution', fontsize=14, pad=15)
plt.xlabel('X', fontsize=12)
plt.ylabel('Probability Density', fontsize=12)
plt.grid(True, alpha=0.3)
plt.legend(fontsize=10)

plt.tight_layout()
plt.show()
```



B. Calculate the probability mass that the random variable X is less than 0, that is, $Pr\{X \leq 0\} = \int_{-\infty}^{0} p(x)dx$

In [3]:
```
probability = stats.norm.cdf(0, loc=1, scale=1)

plt.figure(figsize=(6, 5))
plt.plot(x, dist_pdf, label='PDF')

# Shade the area for X ≤ 0
mask = x <= 0
plt.fill_between(x[mask], dist_pdf[mask], color='blue', alpha=0.3,
                 label=f'P(X ≤ 0) ≈ {probability:.4f}')

# Vertical line at x=0
plt.axvline(x=0, color='r', linestyle='--', alpha=0.5)

plt.title('Gaussian Distribution')
plt.xlabel('X')
plt.ylabel('Probability Density')
plt.grid(True, alpha=0.3)
plt.legend(facecolor='white', framealpha=1)
plt.show()
```



C. Consider the new observation variable z, it gives information about the variable x by the likelihood function $p(z|x) = \mathcal{N}\left(z; x, \sigma^2\right)$, with variance $\sigma^2 = 0.2$. Apply the Bayes' theorem to derive the posterior distribution, $p(x|z)$, given an observation $z = 0.75$ and plot it. For a better comparison, plot the prior distribution, $p(x)$, too.

According to Baies' theorem:

$$p(x|z) = \frac{p(z|x)p(x)}{p(z)}$$

We know $p(x)$ from the previous task and we can obtain $p(z|x)$ from the normal distribution with parameters as in task. But we need to find $p(z)$. There is several options how to obtain it:

1. We can integrate $p(z|x)$ over all x (so called marginalization):

$$p(z) = \int_{-\infty}^{\infty} p(z|x)p(x)dx$$

2. We can calculate mean and variance parameters of the normal distribution for $p(z)$:

$$\mu_z = \sigma_z \times \left(\frac{\mu_x}{\sigma_x} + \frac{z}{\sigma_z}\right)$$

$$\sigma_z = \frac{1}{\frac{1}{\sigma_x} + \frac{1}{\sigma_z}}$$

Thus, we will obtain distribution $p(x|z) \sim \mathcal{N}(z, \mu_z, \sigma_z)$

Let's try both options and compare results.

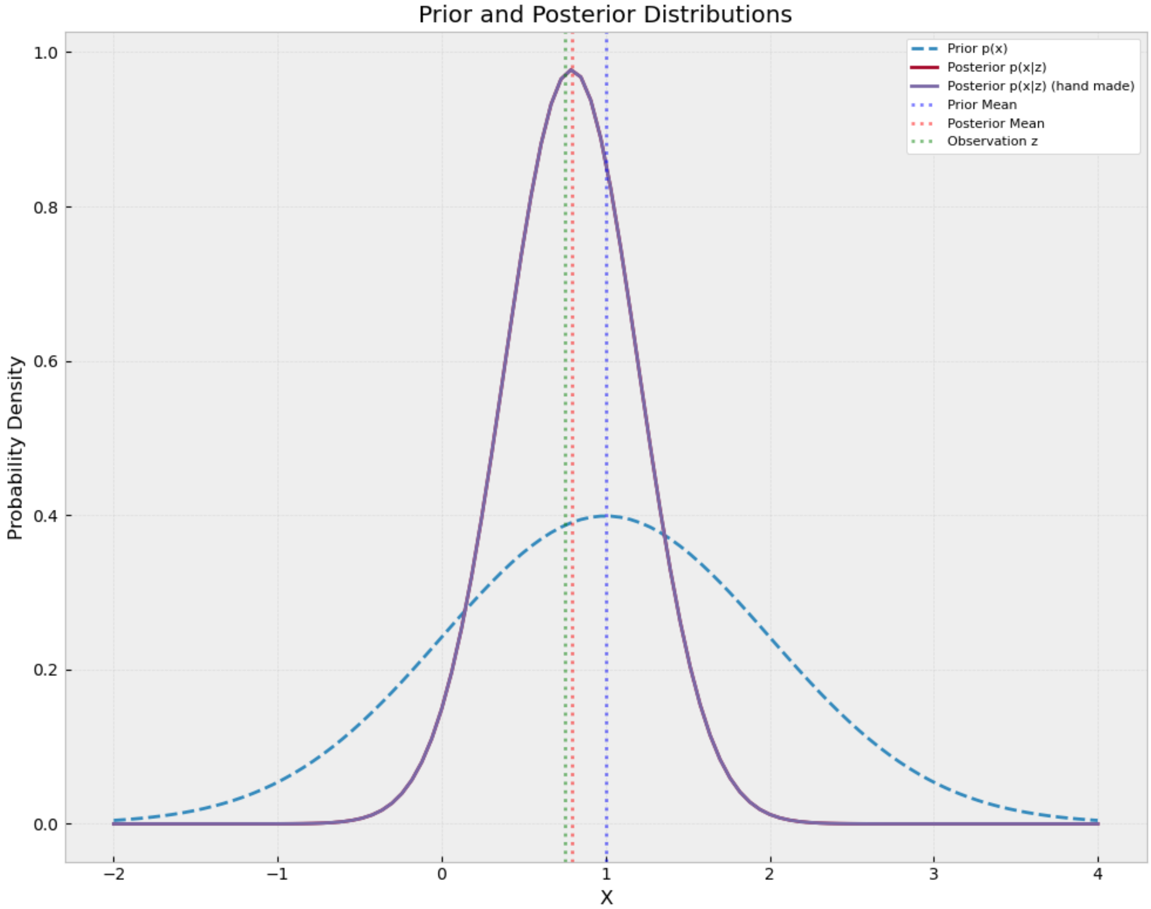
In [4]:
```
z = 0.75
sigma_z = 0.2
prior_mean = 1
prior_var = 1

def integrand(x):
    likelihood = stats.norm.pdf(z, x, np.sqrt(sigma_z))  # p(z|x)
    prior = stats.norm.pdf(x, prior_mean, np.sqrt(prior_var))  # p(
    return likelihood * prior

# Rectangle method
dx = x[1] - x[0]  # Width of each rectangle
p_z = np.sum(integrand(x) * dx)  # Area = sum of (height * width)

print(f"p(z) = {p_z:.4f}")
```

```
p(z) = 0.3548
```

In [5]:
```
posterior_hand_made = (stats.norm.pdf(x, prior_mean, np.sqrt(prior_
                       * stats.norm.pdf(z, x, np.sqrt(sigma_z)))/(p
```

In [6]:
```
# Parameters
z = 0.75
sigma_z = 0.2
prior_mean = 1
prior_var = 1

# Bayes theorem
posterior_var = 1 / (1/prior_var + 1/sigma_z)
posterior_mean = posterior_var * (prior_mean/prior_var + z/sigma_z)

# Calculate PDFs
prior_pdf = stats.norm.pdf(x, prior_mean, np.sqrt(prior_var))
posterior_pdf = stats.norm.pdf(x, posterior_mean, np.sqrt(posterior

plt.figure(figsize=(10, 8))
plt.plot(x, prior_pdf, label='Prior p(x)', linestyle='--')
plt.plot(x, posterior_pdf, label='Posterior p(x|z)')
plt.plot(x, posterior_hand_made, label='Posterior p(x|z) (hand made

plt.axvline(x=prior_mean, color='blue', linestyle=':', alpha=0.5, l
plt.axvline(x=posterior_mean, color='red', linestyle=':', alpha=0.5
plt.axvline(x=z, color='green', linestyle=':', alpha=0.5, label='Ob
```

```python
plt.title('Prior and Posterior Distributions')
plt.xlabel('X')
plt.ylabel('Probability Density')
plt.grid(True, alpha=0.3)
plt.legend(facecolor='white', framealpha=1, fontsize=8)
plt.tight_layout()
plt.show()
```



Task 2. Multivariate Gaussian

A. Write the function `plot2dcov` which plots the 2d contour given three core parameters: mean, covariance, and the iso-contour value k. You may add any other parameter such as color, number of points, etc.

Then, use `plot2dcov` to draw the iso-contours corresponding to $1, 2, 3-\sigma$ of the following Gaussian distributions: $\mathcal{N}\left(\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1 & 0\\0 & 2\end{bmatrix}\right)$, $\mathcal{N}\left(\begin{bmatrix}5\\0\end{bmatrix}, \begin{bmatrix}3 & -0.4\\-0.4 & 2\end{bmatrix}\right)$ and $\mathcal{N}\left(\begin{bmatrix}2\\2\end{bmatrix}, \begin{bmatrix}9.1 & 6\\6 & 4\end{bmatrix}\right)$. Use the `set_aspect('equal')` command and comment on them.

In [7]:
```python
def plot2dcov(mean, cov, k, n_points=100, color='blue', alpha=0.5,
    """
    Plot 2D Gaussian contours

    Parameters:
    mean: 2D mean vector
```

```
    cov: 2x2 covariance matrix
    k: sigma multiplier for iso-contour
    color: contour color
    n_points: number of points for contour
    alpha: transparency
    label: legend label
    """
    theta = np.linspace(0, 2*np.pi, n_points)
    circle = np.vstack([np.cos(theta), np.sin(theta)])

    A = np.linalg.cholesky(cov)

    scaled = k * circle
    transformed = A @ scaled + mean[:, np.newaxis]

    return plt.plot(transformed[0,:], transformed[1,:], color=color
```

In [8]:

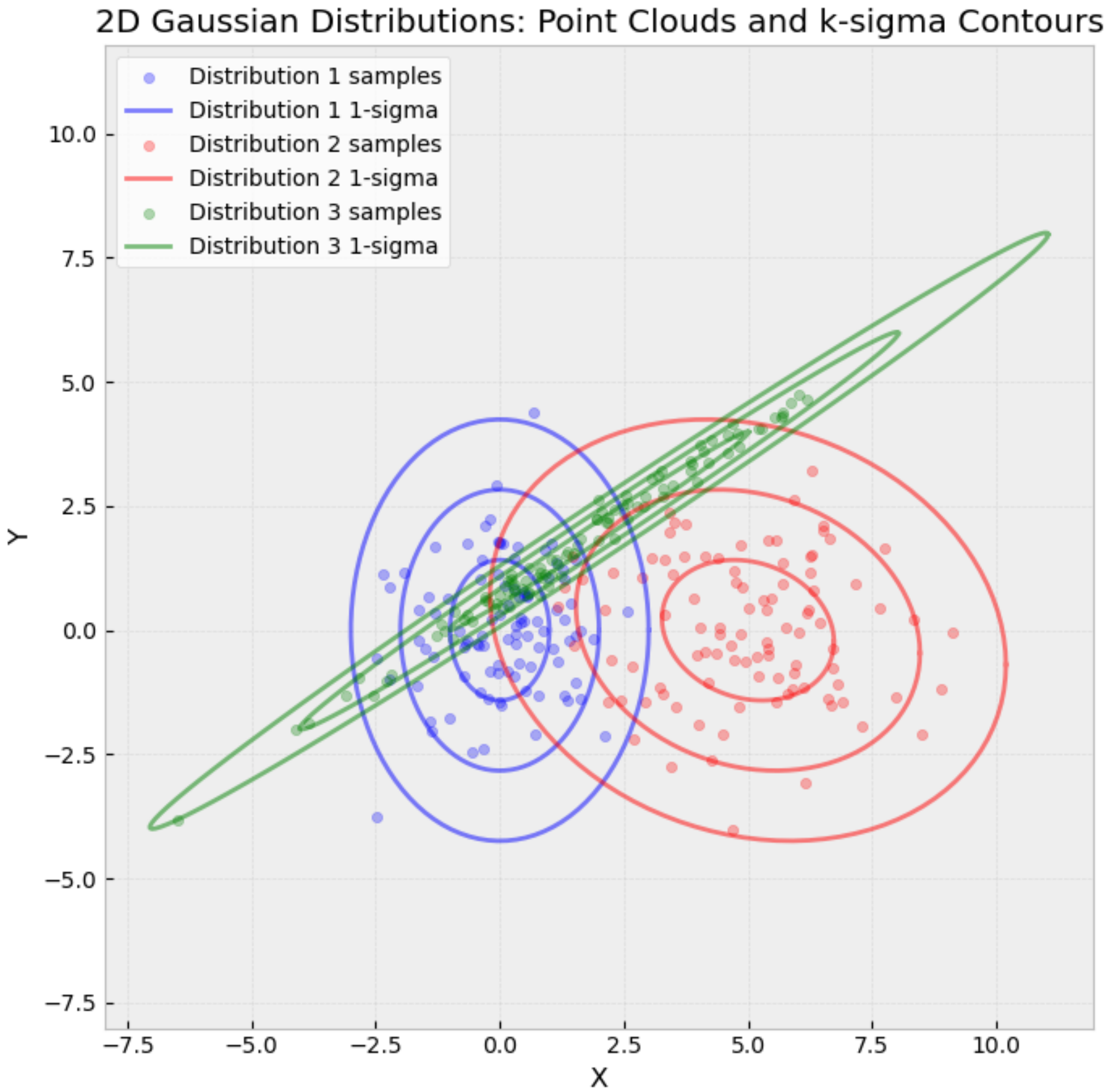
```
mean = np.array([
    [0, 5, 2],
    [0, 0, 2]
])

cov = np.array([
    [1, 0, 3, -0.4, 9.1, 6],
    [0, 2, -0.4, 2, 6, 4]
])

for i in range(len(mean.T)):
    print(f'mean\n {mean[:,i]}')
    print(f'cov\n {cov[:,2*i:2*(i+1)]}')
```

```
mean
 [0 0]
cov
 [[1. 0.]
 [0. 2.]]
mean
 [5 0]
cov
 [[ 3.  -0.4]
 [-0.4  2. ]]
mean
 [2 2]
cov
 [[9.1 6. ]
 [6.  4. ]]
```

In [9]:

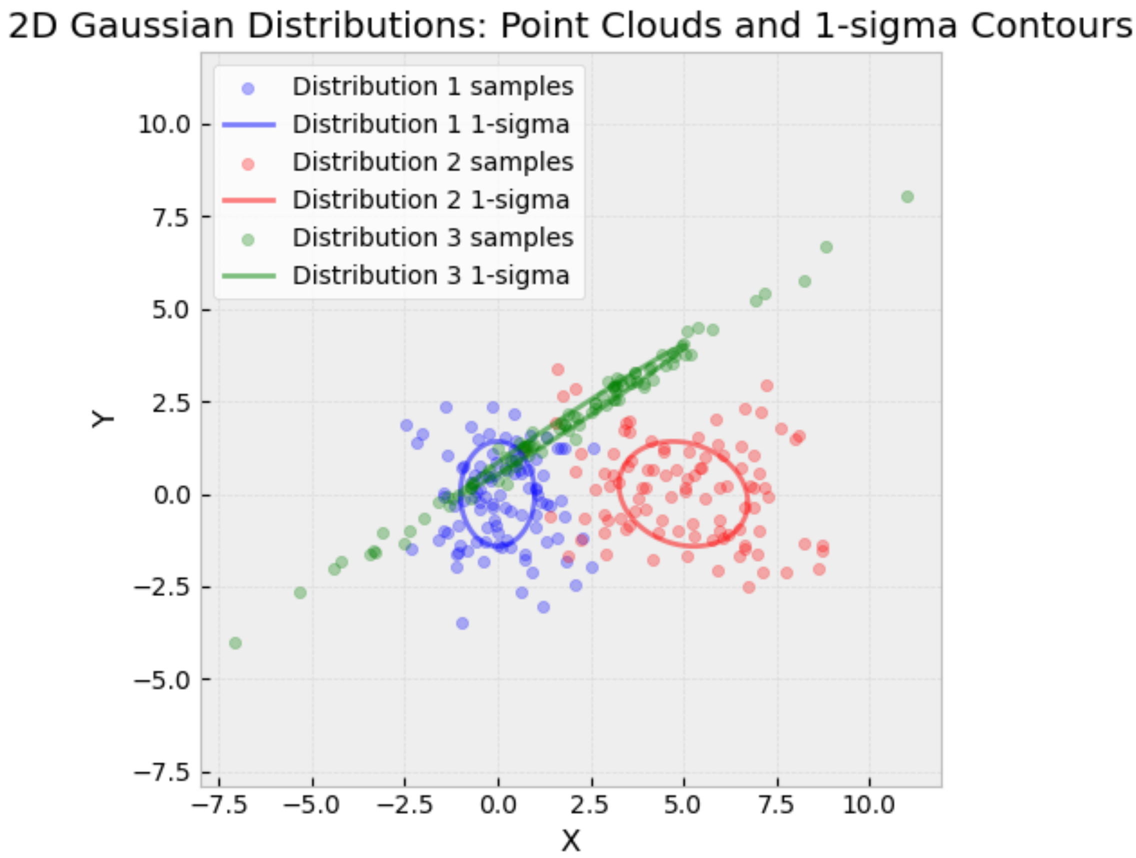
```
# Create figure
plt.figure(figsize=(7,7))

# Plot contours for each distribution
sigmas = [1, 2, 3]
distributions = [
    (mean[:,0], cov[:,0:2], 'Distribution 1'),
    (mean[:,1], cov[:,2:4], 'Distribution 2'),
    (mean[:,2], cov[:,4:6], 'Distribution 3')
```

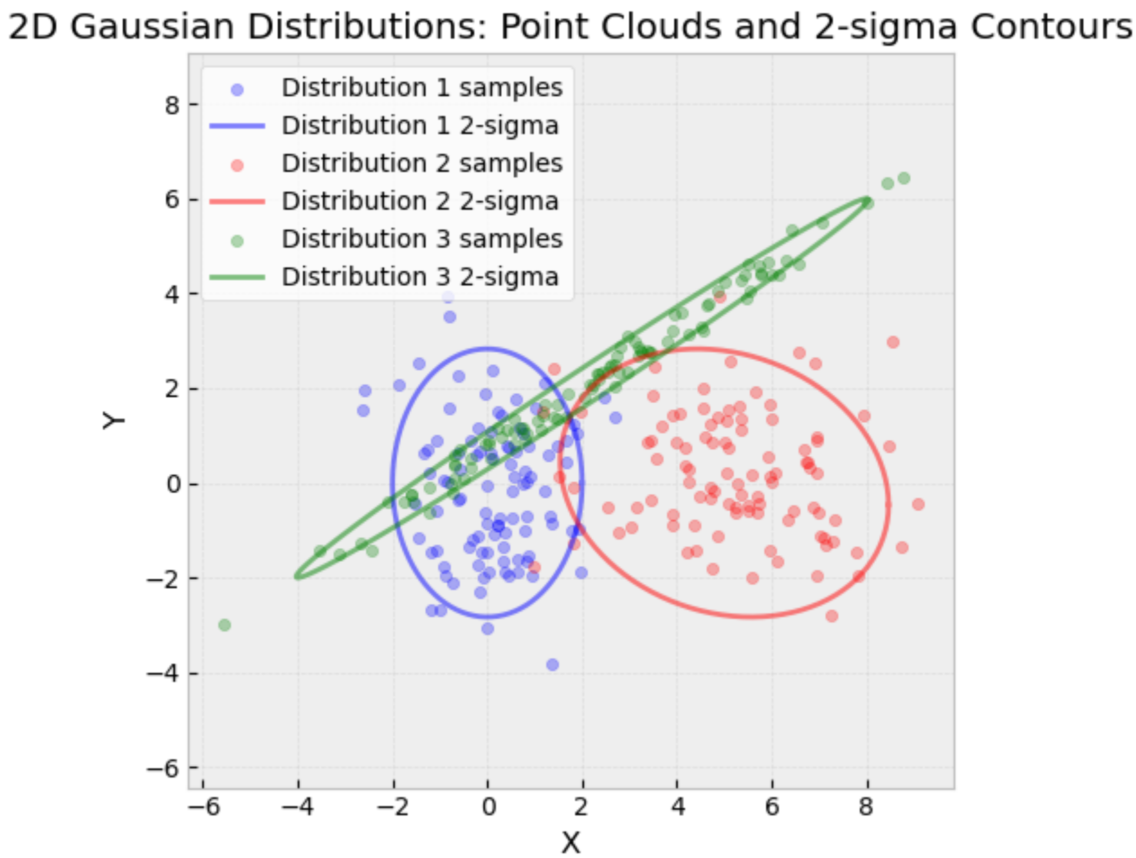
```
]
colors = ['blue', 'red', 'green']

# Generate and plot point clouds
n_samples = 100
for (mu, sigma, name), color in zip(distributions, colors):
    # Generate random samples
    original_cloud = np.random.randn(n_samples, 2)
    A = np.linalg.cholesky(sigma)
    cloud = (A @ original_cloud.T + mu.reshape(-1,1)).T

    # Plot point cloud
    plt.scatter(cloud[:,0], cloud[:,1], c=color, alpha=0.3, s=20, l

    # Plot contours
    for k in sigmas:
        plot2dcov(mu, sigma, k, color=color, alpha=0.5,
                  label=f'{name} {k}-sigma' if k==1 else None)

plt.grid(True, alpha=0.3)
plt.xlabel('X')
plt.ylabel('Y')
plt.title('2D Gaussian Distributions: Point Clouds and k-sigma Cont
plt.legend(facecolor='white', loc='upper left')
plt.axis('equal')  # set_aspect('equal')
plt.tight_layout()
plt.show()
```

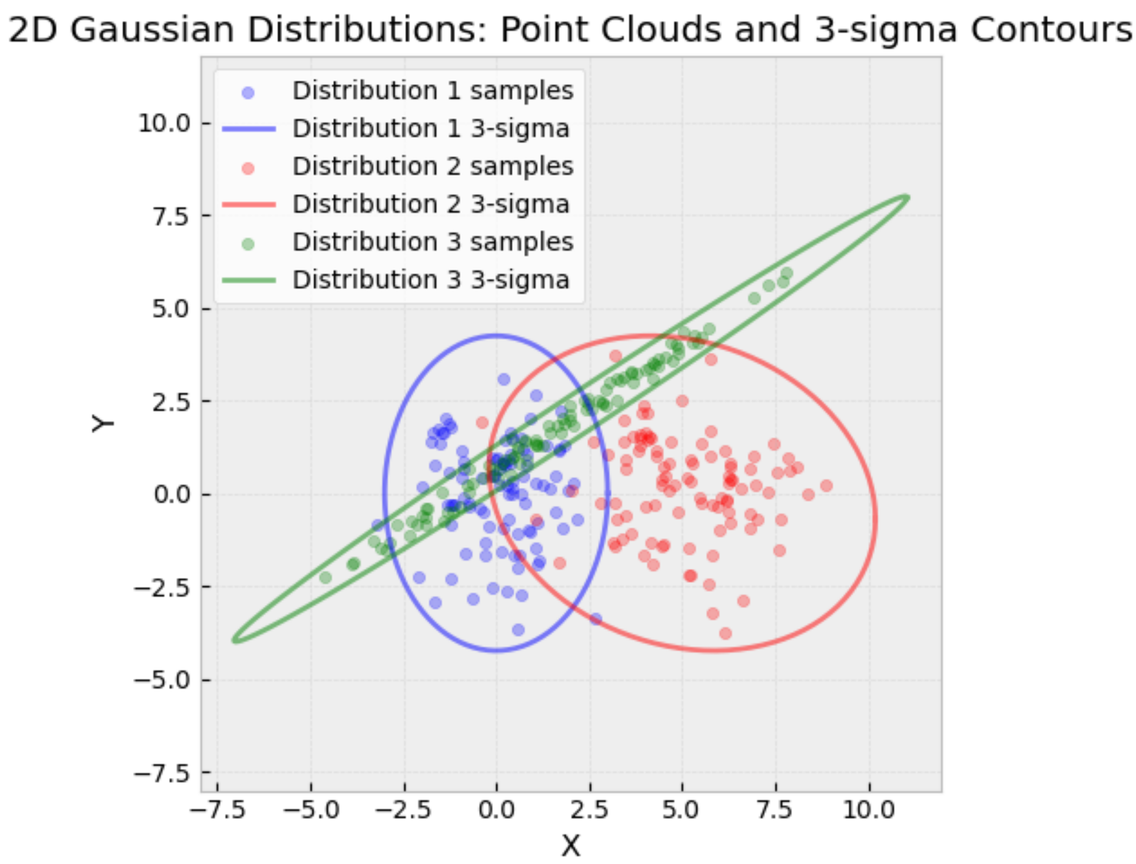


In [10]:
```
# Create 3 figures, one for each sigma value
for sigma_val in sigmas:
    plt.figure(figsize=(5,5))

    # Plot point clouds and contours for each distribution
    for (mu, sigma, name), color in zip(distributions, colors):
        # Generate random samples
        original_cloud = np.random.randn(n_samples, 2)
        A = np.linalg.cholesky(sigma)
        cloud = (A @ original_cloud.T + mu.reshape(-1,1)).T

        # Plot point cloud
        plt.scatter(cloud[:,0], cloud[:,1], c=color, alpha=0.3, s=2

        # Plot single contour with current sigma value
        plot2dcov(mu, sigma, sigma_val, color=color, alpha=0.5,
                  label=f'{name} {sigma_val}-sigma')

    plt.grid(True, alpha=0.3)
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title(f'2D Gaussian Distributions: Point Clouds and {sigma_
    plt.legend(facecolor='white', loc='upper left')
```

```
    plt.axis('equal')
    plt.tight_layout()
    plt.show()
```







B. Write the equation of sample mean and sample covariance of a set of points $\{x_i\}$, in vector form as was shown during the lecture. You can provide your solution by using Markdown, latex, by hand, etc.

From first lecture, the sample mean is expectation of the set:

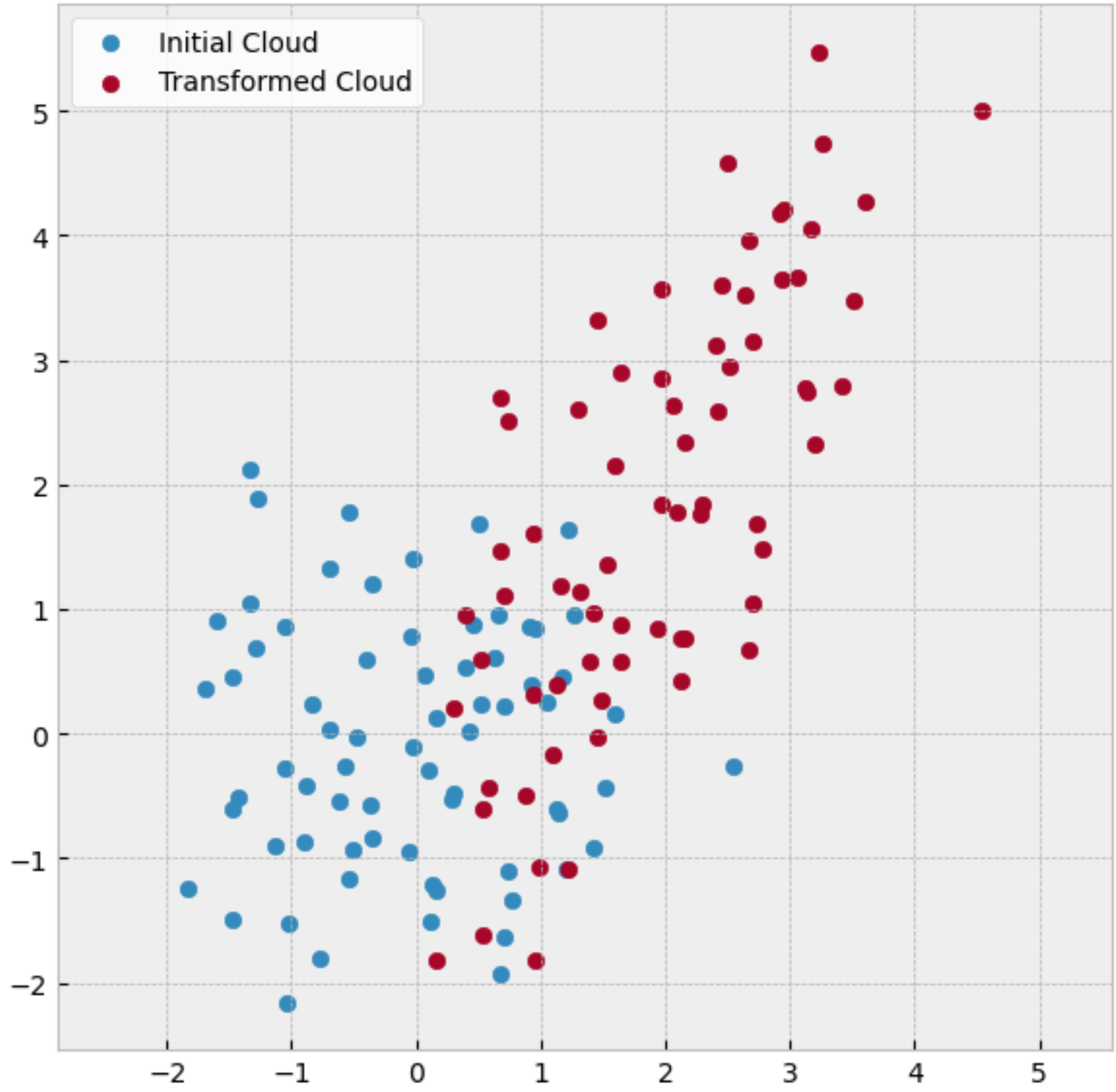
$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

The sample covariance matrix is defined as:

$$\bar{\Sigma} = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})(x_i - \bar{x})^T$$

C. Draw random samples from a multivariate normal distribution. You can use the python function that draws samples from the univariate normal distribution $\mathcal{N}(0,1)$. In particular, draw and plot 200 samples from $\mathcal{N}\left(\begin{bmatrix}2\\2\end{bmatrix}, \begin{bmatrix}1 & 1.3\\1.3 & 3\end{bmatrix}\right)$; also plot their corresponding 1-sigma iso-contour. Then calculate the sample mean and covariance in vector form and plot again the 1-sigma iso-contour for the estimated Gaussian parameters. Run the experiment multiple times and try a different number of samples (e.g. 50, 400). Comment briefly on the results

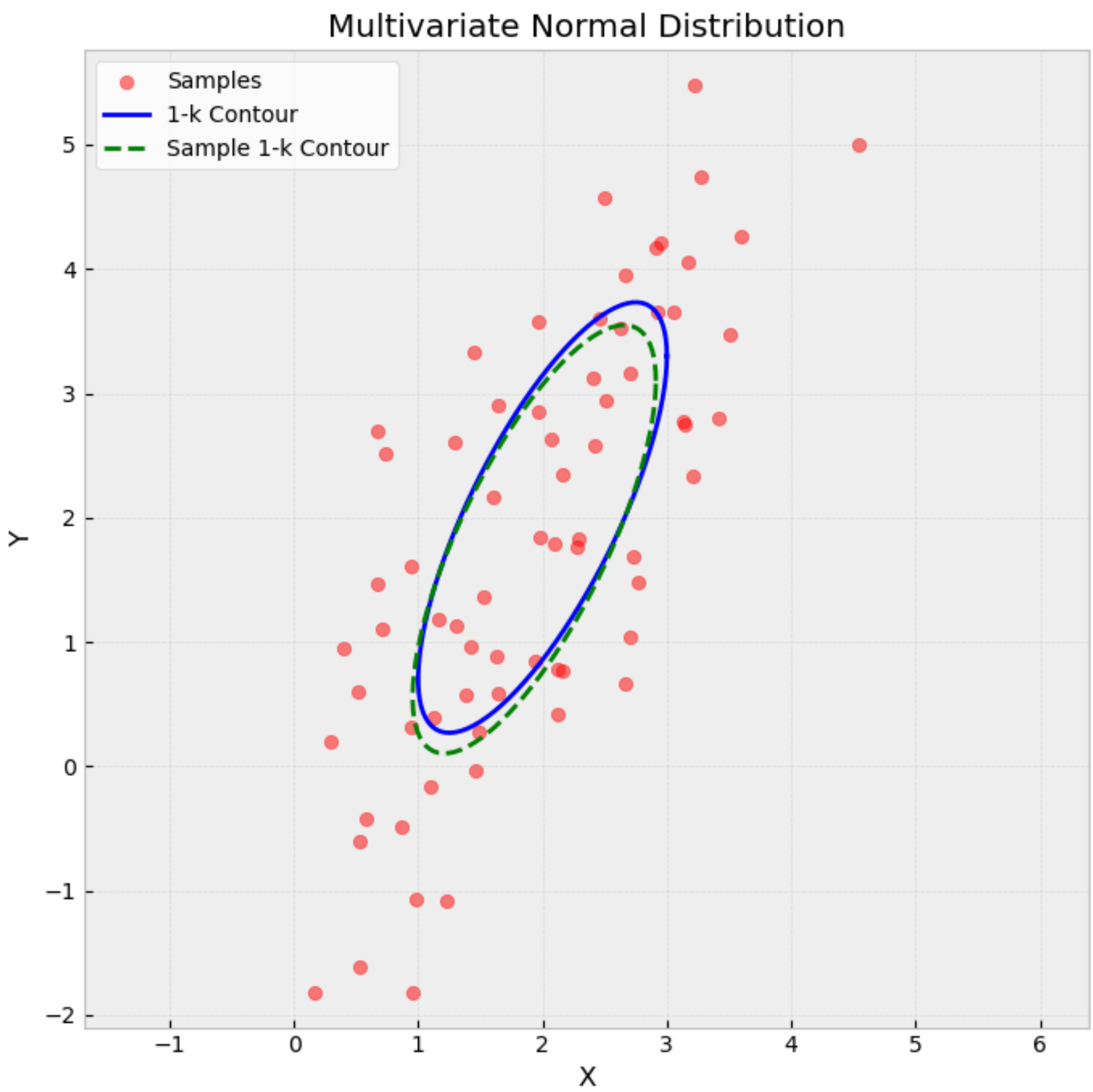
In [11]:
```
n_samples = 70
original_cloud = np.random.randn(n_samples, 2)

mean = np.array([
    [2],
    [2]
])

cov = np.array([
    [1, 1.3],
    [1.3, 3]
])

A = np.linalg.cholesky(cov)
cloud = (A @ original_cloud.T + mean).T

plt.figure(figsize=(7,7))
plt.scatter(original_cloud[:,0], original_cloud[:,1],label='Initial
plt.scatter(cloud[:,0], cloud[:,1],label='Transformed Cloud')
plt.axis('equal')
plt.legend(facecolor='white')
plt.show()
```



In [12]:
```
plt.figure(figsize=(7,7))
```

```
plt.scatter(cloud[:,0], cloud[:,1], alpha=0.5, color='red', label='

theta = np.linspace(0, 2*np.pi, 100)
circle = np.vstack([np.cos(theta), np.sin(theta)])

A = np.linalg.cholesky(cov)
ellipse = A @ circle + mean
plt.plot(ellipse[0,:], ellipse[1,:], 'r-', color='blue', label='1-k

sample_mean = np.mean(cloud, axis=0).reshape(-1,1)
sample_cov = np.cov(cloud.T)
A_sample = np.linalg.cholesky(sample_cov)

sample_ellipse = A_sample @ circle + sample_mean
plt.plot(sample_ellipse[0,:], sample_ellipse[1,:], 'g--', label='Sa

plt.grid(True, alpha=0.3)
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Multivariate Normal Distribution')
plt.legend(facecolor='white', loc='upper left')
plt.axis('equal')
plt.tight_layout()
plt.show()
```

```
/var/folders/7k/4vb9j3_s13l8qv3m4vdw9m5w0000gn/T/ipykernel_90625/170
4225953.py:9: UserWarning: color is redundantly defined by the 'colo
r' keyword argument and the fmt string "r-" (-> color='r'). The keyw
ord argument will take precedence.
  plt.plot(ellipse[0,:], ellipse[1,:], 'r-', color='blue', label='1-
k Contour')
```



Observations on Sample Size Effects:

- With a larger number of samples (70+), the sample covariance ellipse (green) closely matches the true covariance ellipse (blue)
- With fewer samples (e.g., <50), there is more discrepancy between sample and true covariance due to increased sampling variability
- The sample mean (center of green ellipse) also becomes more accurate with increased sample size

Task 3. Covariance Propagation

Discrete-time propagation model:

$$\begin{bmatrix} x \\ y \end{bmatrix}_t = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}_{t-1} + \begin{bmatrix} \Delta t & 0 \\ 0 & \Delta t \end{bmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix}_t + \begin{bmatrix} \eta_x \\ \eta_y \end{bmatrix}_t$$

Control:

$$u = \left[v_x, v_y\right]^T$$

Uncertainty on command execution:

$$\begin{bmatrix} \eta_x \\ \eta_y \end{bmatrix}_t \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0.1 & 0 \\ 0 & 0.1 \end{bmatrix}\right)$$

Time step is $\Delta t = 0.5$

A. Write the equations corresponding to the mean and covariance after a single propagation of the holonomic platform.

We can rewrite Discrete-time propagation model as:

$$\begin{cases} x_t = x_{t-1} + \Delta t v_{xt} + \eta_{xt} \\ y_t = y_{t-1} + \Delta t v_{yt} + \eta_{yt} \end{cases}$$

Then propagate mean:

$$\mu_x = E\{x_t\} = E\{x_{t-1} + \Delta t v_{xt} + \eta_{xt}\}$$
$$\mu_x = E\{x_{t-1}\} + E\{\Delta t v_{xt}\} + E\{\eta_{xt}\}$$
$$\mu_x = \mu_{x-1} + \Delta t v_{xt} + 0$$

Similar equation we can write for y_t:

$$\mu_y = \mu_{y-1} + \Delta t v_{yt} + 0$$

And write this in matrix form:

$$\mu_t = \begin{bmatrix} \mu_x \\ \mu_y \end{bmatrix}_t = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \mu_{t-1} + \begin{bmatrix} \Delta t & 0 \\ 0 & \Delta t \end{bmatrix} u_t, where\ \mu_{t-1} = [\mu_x, \mu_y]^T_{t-1}\ and$$

And for covariance:

$$\Sigma_x = E\{(x_t - \mu_x)(x_t - \mu_x)^T\} =$$

$$= E\{(x_{t-1} + \text{\bcancel}\Delta t v_{xt} + \eta_{xt} - \mu_{x-1} - \text{\bcancel}\Delta t v_{xt})(x_{t-1} + \eta_{xt} - \mu$$

$$= E\{\underbrace{(x_{t-1} - \mu_{x-1})(x_{t-1} - \mu_{x-1})^T}_{cov(x_{t-1}) \text{ by def.}} + (x_{t-1} - \mu_{x-1})\eta_{xt}^T + \underbrace{\eta_{xt}\eta_{xt}^T}_{var(\eta_x) \text{ by def.}}\}$$

$$= \Sigma_{x_{t-1}} + E\{x_{t-1}\}\text{\cancel}E\{\eta_{xt}^T\} - \mu_{x-1}\ \text{\cancel}\mu_{xt}^T + \Sigma_{\eta_x}$$

Finally:

$$\Sigma_x = \Sigma_{x_{t-1}} + \Sigma_{\eta_x}$$

And similar for y_t:

$$\Sigma_y = \Sigma_{y_{t-1}} + \Sigma_{\eta_x}$$

In matrix form:

$$\Sigma_t = \begin{bmatrix} \Sigma_x \\ \Sigma_y \end{bmatrix}_t = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \Sigma_{t-1} + \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \Sigma_\eta$$

Thus we obtain:

$$x_t \sim \mathcal{N}(\mu_t, \Sigma_t)$$

B. Show how to use this result iteratively for multiple propagations.

For iterative propagation over multiple steps, we have the basic update equations:

$$\mu_t = \begin{bmatrix} \mu_x \\ \mu_y \end{bmatrix}_t = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \mu_{t-1} + \begin{bmatrix} \Delta t & 0 \\ 0 & \Delta t \end{bmatrix} u_t \text{ and}$$

$$\Sigma_t = \begin{bmatrix} \Sigma_x \\ \Sigma_y \end{bmatrix}_t = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \Sigma_{t-1} + \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \Sigma_\eta$$

For n-step propagation, the position evolves as:

$$\begin{bmatrix} x \\ y \end{bmatrix}_n = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}_0 + \begin{bmatrix} \Delta t & 0 \\ 0 & \Delta t \end{bmatrix} \sum_{i=0}^{n} \begin{bmatrix} v_x \\ v_y \end{bmatrix}_i + \sum_{i=0}^{n} \begin{bmatrix} \eta_x \\ \eta_y \end{bmatrix}_i$$

The mean after **n** steps becomes with constant control command u:

$$\mu_n = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \mu_0 + n \begin{bmatrix} \Delta t & 0 \\ 0 & \Delta t \end{bmatrix} u_0$$

The covariance accumulates linearly:

$$\Sigma_t = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \Sigma_0 + n \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \Sigma_\eta$$

Therefore, the final n-step distribution is:

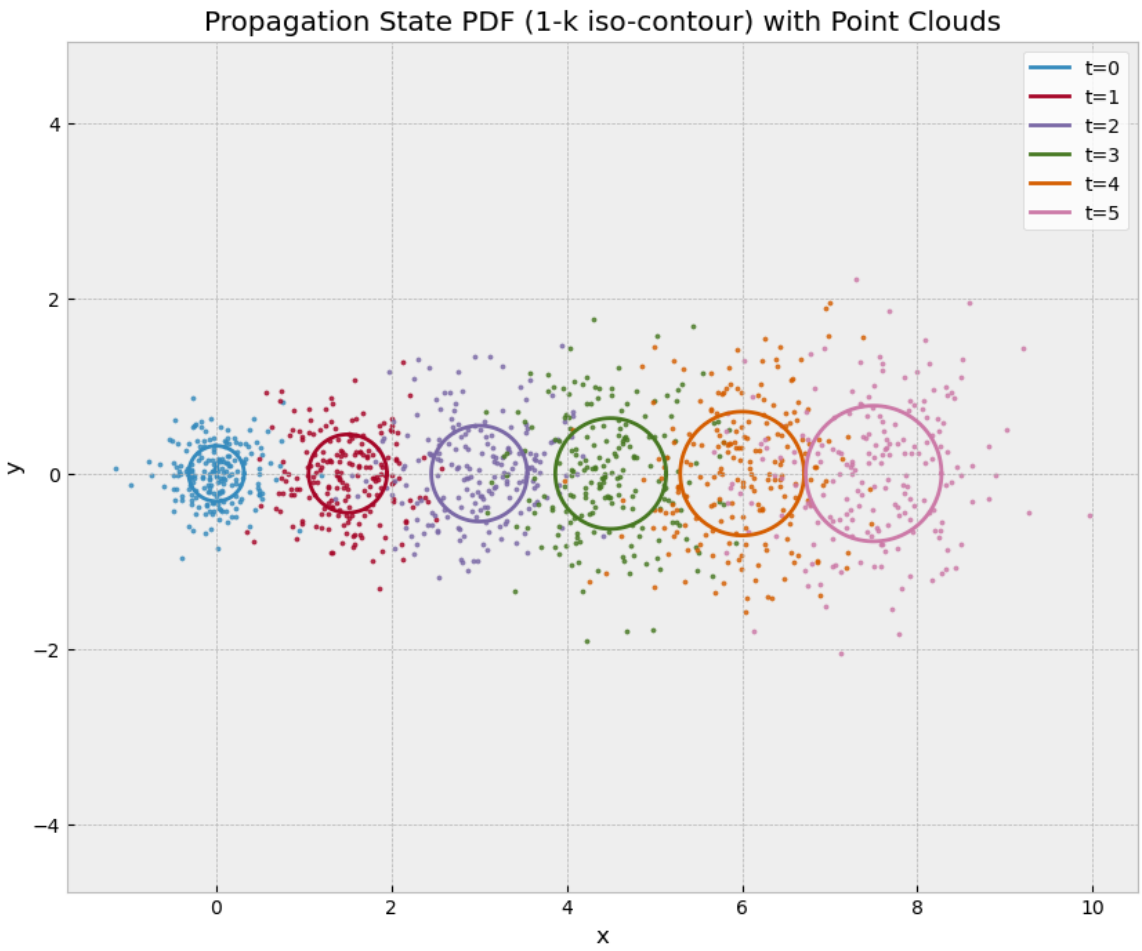
$$x_n \sim \mathcal{N}(\mu_0 + n\Delta t u, \Sigma_0 + n\Sigma_\eta)$$

C. Draw the propagation state PDF ($1 - \sigma$ iso-contour) for times indexes $t = 0, \ldots, 5$ and the control sequence $u = [3, 0]^T$ for all times t. The PDF for the initial state is $\mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0.1 & 0 \\ 0 & 0.1 \end{bmatrix}\right)$.

In [13]:
```
mu_initial = np.array([0, 0])
u_control = np.array([3, 0])
cov_initial = np.array([[0.1, 0], [0, 0.1]])
sigma_eta = np.array([[0.1, 0], [0, 0.1]])
```

```
dt = 0.5

plt.figure(figsize=(10, 8))

# For each time step
for n in range(6):
    # Calculate mean at time t
    mu_t = mu_initial + n * dt * u_control

    # Calculate covariance at time t
    sigma_t = cov_initial + n * sigma_eta

    L = np.linalg.cholesky(sigma_t)

    # Generate points for 1-sigma ellipse
    theta = np.linspace(0, 2*np.pi, 100)
    ellipse = np.zeros((2, theta.size))

    for i in range(theta.size):
        circle_pt = np.array([np.cos(theta[i]), np.sin(theta[i])])
        ellipse_pt = mu_t + np.dot(L, circle_pt)
        ellipse[:,i] = ellipse_pt

    plt.plot(ellipse[0,:], ellipse[1,:], label=f't={n}')

    num_points = 200
    samples = np.random.randn(2, num_points)
    points = mu_t.reshape(-1,1) + np.dot(L, samples)

    plt.scatter(points[0,:], points[1,:], alpha=0.8, s=5)

plt.grid(True)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Propagation State PDF (1-k iso-contour) with Point Cloud
plt.legend(facecolor='white')
plt.axis('equal')
```

Out[13]:
```
(-1.704289026848616,
 10.513129490253842,
 -2.2750412282121455,
 2.438562214388717)
```



D. Discrete-time propagation model changed:

$$\begin{bmatrix} x \\ y \end{bmatrix}_t = \begin{bmatrix} 1 & 0 \\ 0.1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}_{t-1} + \begin{bmatrix} \Delta t & 0 \\ 0 & \Delta t \end{bmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix}_t + \begin{bmatrix} \eta_x \\ \eta_y \end{bmatrix}_t$$

Other parameters are the same as previous. Draw the propagation state PDF ($1 - \sigma$ iso-contour and 500 particles) for times indexes $t = 0, \ldots, 5$ in the same figure.

Recall previous formulas and according to new information:

The mean after **n** steps becomes with constant control command u:

μ_x is same as before, but μ_y is changed:

$$\mu_{yt} = E\{y_t\} = E\{0.1x_{t-1}\} + E\{y_{t-1}\} + E\{\Delta t v_{yt}\} + \textcolor{red}{\backslash\text{cancel}}E\{\eta_{yt}\}$$

Simplify and obtain:

$$\mu_{yt} = 0.1\mu_{xt-1} + \mu_{yt-1} + \Delta t v_{yt}$$

In matrix form:

$$\mu_n = \underbrace{\begin{bmatrix} 1 & 0 \\ 0.1 & 1 \end{bmatrix}}_{A} \mu_0 + n \underbrace{\begin{bmatrix} \Delta t & 0 \\ 0 & \Delta t \end{bmatrix}}_{B} u_0$$

The situation with covariance is more complex. For Σ_x we have same, but for Σ_y we have:

$$\Sigma_y = E\{(0.1x_{t-1} + y_{t-1} + \backslash\text{cancel}\Delta t v_{yt} + \eta_{yt} - 0.1\mu_{xt-1} - \mu_{yt-1} - \backslash\text{cancel}$$

$$= E\{(0.1(x_{t-1} - \mu_{xt-1}) + (y_{t-1} - \mu_{yt-1}) + \eta_{yt})(0.1(x_{t-1} - \mu_{xt-1}) + (y_{t-1} -$$

Expand braces and obtain:

Here's the LaTeX version of the expanded equation:

$$= (0.1x_{t-1} - 0.1\mu_{xt-1})(0.1x_{t-1} - 0.1\mu_{xt-1})^T + (\text{we can regonize this as } \Sigma_{x_t}$$

$$+(0.1x_{t-1} - 0.1\mu_{xt-1})(y_{t-1} - \mu_{yt-1})^T + (y_{t-1} - \mu_{yt-1})(0.1x_{t-1} - 0.1\mu_{xt-1})^T$$

$$+(0.1x_{t-1} - 0.1\mu_{xt-1})\backslash\text{bcancel}\eta_{yt}^T +$$

$$+(y_{t-1} - \mu_{yt-1})(y_{t-1} - \mu_{yt-1})^T + (\text{we can regonize this as } \Sigma_{y_{t-1}})$$

$$+(y_{t-1} - \mu_{yt-1})\backslash\text{bcancel}\eta_{yt}^T +$$

$$+\backslash\text{bcancel}\eta_{yt}(0.1x_{t-1} - \ldots)^T + \backslash\text{bcancel}\eta_{yt}(y_{t-1} - \ldots)^T + \eta_{yt}\eta_{yt}^T + (\text{last is}$$

Which simplifies to:

$$\Sigma_{yt} = 0.01\Sigma_{xt-1} + \Sigma_{yt-1} + \Sigma_{\eta y} + 0.1(\Sigma_{xy_{t-1}} + \Sigma_{xy_{t-1}}^T)$$

And with Σ_{xt} in matrix form we have:

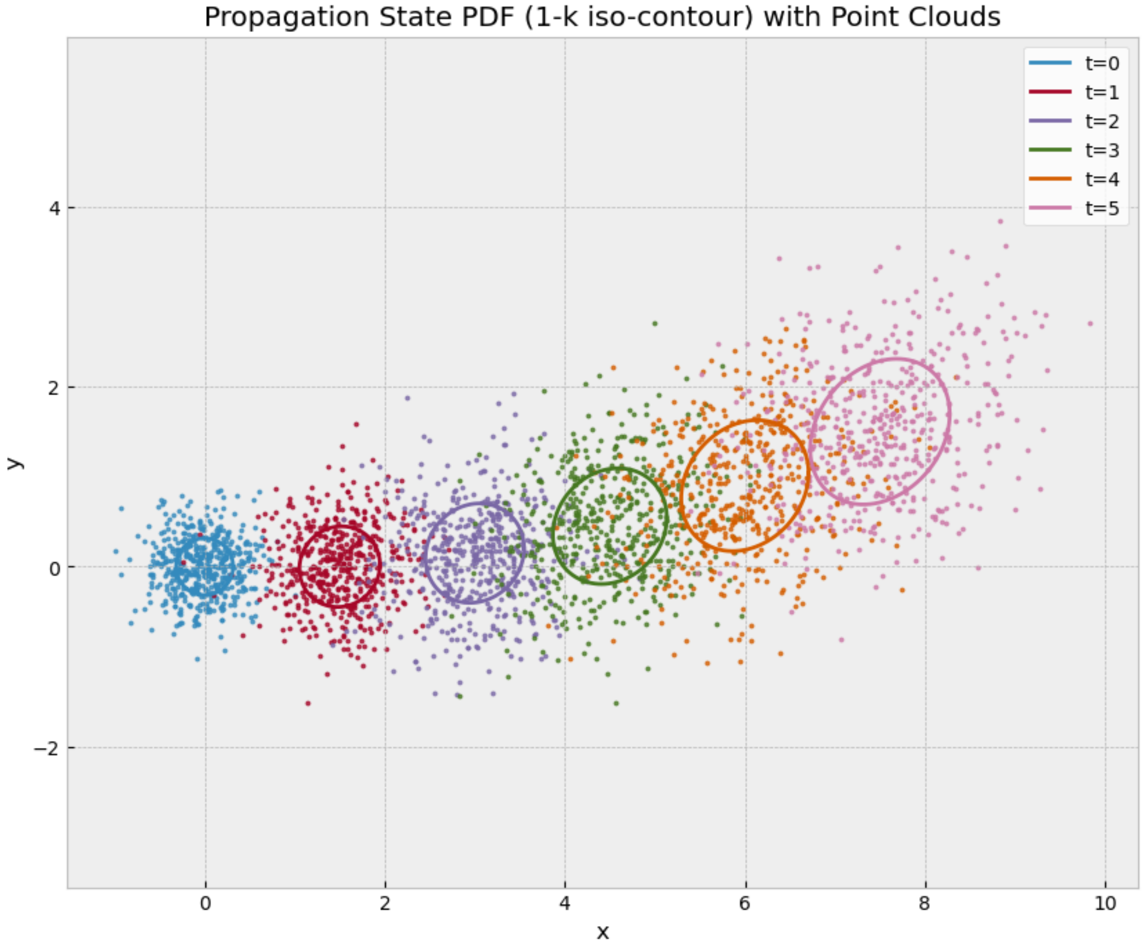
$$\Sigma_t = \begin{bmatrix} \Sigma_x & \Sigma_{xy} \\ \Sigma_{xy}^T & \Sigma_y \end{bmatrix}_t = \underbrace{\begin{bmatrix} 1 & 0 \\ 0.1 & 1 \end{bmatrix}}_{A} \begin{bmatrix} \Sigma_x & \Sigma_{xy} \\ \Sigma_{xy}^T & \Sigma_y \end{bmatrix}_{t-1} \underbrace{\begin{bmatrix} 1 & 0.1 \\ 0 & 1 \end{bmatrix}}_{A^T} + \underbrace{\begin{bmatrix} \Sigma_{\eta x} & 0 \\ 0 & \Sigma_{\eta y} \end{bmatrix}}_{\Sigma_\eta}$$

In [14]:
```
mu_initial = np.array([0, 0])
u_control = np.array([3, 0])
cov_initial = np.array([[0.1, 0], [0, 0.1]])
sigma_eta = np.array([[0.1, 0], [0, 0.1]])
dt = 0.5

A = np.array([[1, 0], [0.1, 1]])
B = np.array([[dt, 0], [0, dt]])

plt.figure(figsize=(10, 8))

for n in range(6):
    if n == 0:
```

```
        mu_t = mu_initial
        sigma_t = cov_initial
    else:
        # Calculate mean at time t using new state transition model
        mu_t = A@mu_prev + B@u_control

        # Calculate covariance at time t
        sigma_t = ((A@sigma_prev)@A.T) + sigma_eta

    mu_prev = mu_t
    sigma_prev = sigma_t

    L = np.linalg.cholesky(sigma_t)

    theta = np.linspace(0, 2*np.pi, 100)
    ellipse = np.zeros((2, theta.size))

    for i in range(theta.size):
        circle_pt = np.array([np.cos(theta[i]), np.sin(theta[i])])
        ellipse_pt = mu_t + L@circle_pt
        ellipse[:,i] = ellipse_pt

    plt.plot(ellipse[0,:], ellipse[1,:], label=f't={n}')

    num_points = 500
    samples = np.random.randn(2, num_points)
    points = mu_t.reshape(-1,1) + L@samples

    plt.scatter(points[0,:], points[1,:], alpha=0.8, s=5)

plt.grid(True)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Propagation State PDF (1-k iso-contour) with Point Cloud
plt.legend(facecolor='white')
plt.axis('equal')
```

Out[14]:

```
(-1.530658170942804, 10.36494617017292, -1.778408859112764, 4.1066
64453698389)
```



E. Now, suppose that the robotic platform is non-holonomic, and the corresponding propagation model is:

$$\begin{bmatrix} x \\ y \\ \theta \end{bmatrix}_t = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}_{t-1} + \begin{bmatrix} \cos\theta\Delta t & 0 \\ \sin\theta\Delta t & 0 \\ 0 & \Delta t \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix}_t + \begin{bmatrix} \eta_x \\ \eta_y \\ \eta_\theta \end{bmatrix}_t \sim \mathcal{N}\left(\left[\right.\right.$$

PDF for initial state:

$$\begin{bmatrix} x \\ y \\ \theta \end{bmatrix}_0 \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0.1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.1 \end{bmatrix}\right)$$

Propagate, as explained in class (linearize plus covariance propagation), for five time intervals, using the control $u = [3, 1.5]^T$ showing the propagated Gaussian by plotting the $1 - \sigma$ iso-contour. Angles are in radians.

Unicycle Model Formulas (from lecture)

Expectation

The state update equation shows how the robot's state (position x,y and orientation θ) changes over time:

$$\mu_t = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}_t = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}_{t-1} + \begin{bmatrix} \Delta t \cdot V_t \cdot \cos\theta \\ \Delta t \cdot V_t \cdot \sin\theta \\ \Delta t \cdot \omega_t \end{bmatrix}$$

Where:

- V_t is the linear velocity
- ω_t is the angular velocity
- Δt is the time step

Linearization

The Jacobian matrix G_t represents the linearized system:

$$G_t = \begin{bmatrix} 1 & 0 & -\sin(\theta) \cdot \Delta t \cdot V_t \\ 0 & 1 & \cos(\theta) \cdot \Delta t \cdot V_t \\ 0 & 0 & 1 \end{bmatrix}$$

Covariance Update

The covariance matrix is:

$$\Sigma_t = G_t \Sigma_{t-1} G_t^T + R$$

Where R is the process noise covariance matrix.

Noise Model

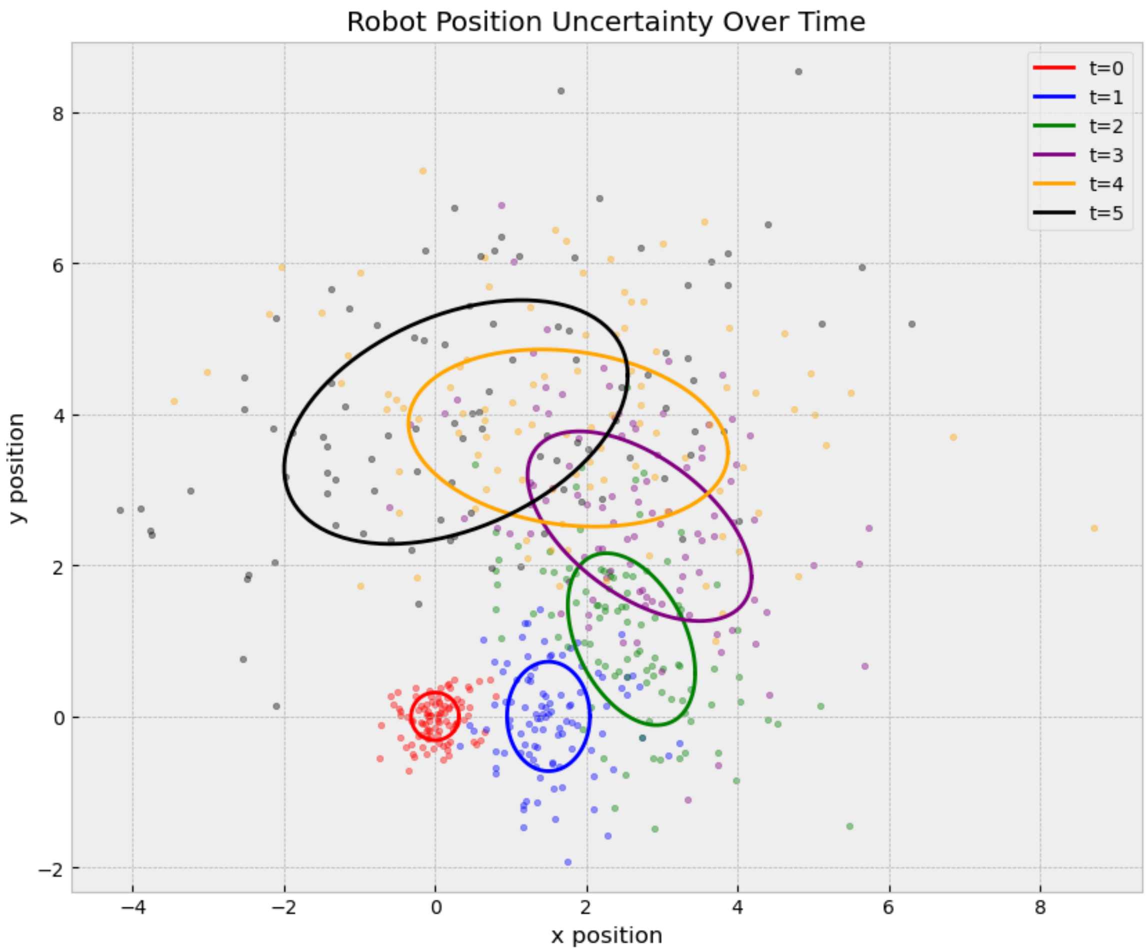
The noise is modeled as zero-mean Gaussian:

$$\eta_t \sim \mathcal{N}(0, R)$$

This model describes a wheeled robot that can move forward/backward and rotate, with uncertainty in its motion captured by the noise term.

In [21]:
```
dt = 0.5
control = np.array([3, 1.5])  # [velocity, angular velocity]
noise_cov = np.array([[0.2, 0, 0],
                      [0, 0.2, 0],
                      [0, 0, 0.1]])
initial_state = np.array([0, 0, 0])  # [x, y, theta]
initial_cov = np.array([[0.1, 0, 0],
                        [0, 0.1, 0],
                        [0, 0, 0.1]])

def plot_ellipse(mean, cov, color='red', label=None):
    """Plot uncertainty ellipse"""
    # Generate points on a circle
    angles = np.linspace(0, 2*np.pi, 100)
    circle = np.array([np.cos(angles), np.sin(angles)])

    # Transform circle into ellipse
    L = np.linalg.cholesky(cov[:2, :2])
    ellipse = (L @ circle + mean[:2, np.newaxis]).T
```

```
    plt.plot(ellipse[:, 0], ellipse[:, 1], color=color, label=label

def propagate_state(n_steps=5):
    """Propagate robot state and uncertainty"""
    states = []
    mean = initial_state
    cov = initial_cov
    v, w = control

    # Store initial state
    states.append((mean.copy(), cov.copy()))

    # Propagate for n steps
    for t in range(n_steps):
        theta = mean[2]

        # Jacobian matrix
        G = np.array([
            [1, 0, -v * np.sin(theta) * dt],
            [0, 1, v * np.cos(theta) * dt],
            [0, 0, 1]
        ])

        # Update mean
        mean = mean + np.array([
            dt * np.cos(theta) * v,
            dt * np.sin(theta) * v,
            dt * w
        ])

        # Update covariance
        cov = G @ cov @ G.T + noise_cov
        states.append((mean.copy(), cov.copy()))

    return states

plt.figure(figsize=(10, 8))
colors = ['red', 'blue', 'green', 'purple', 'orange', 'black']

states = propagate_state()
for t, (mean, cov) in enumerate(states):
    color = colors[t]
    plot_ellipse(mean, cov, color=color, label=f"t={t}")

    # Add some random samples to show uncertainty
    samples = np.random.multivariate_normal(mean[:2], cov[:2, :2],
    plt.scatter(samples[:, 0], samples[:, 1], color=color, alpha=0.

plt.grid(True)
plt.xlabel('x position')
plt.ylabel('y position')
plt.title('Robot Position Uncertainty Over Time')
plt.legend()
plt.axis('equal')
plt.show()
```



F. Repeat the same experiment as above, using the same control input ut and initial state estimate, now considering that noise is expressed in the action space instead of state space:

$$\begin{bmatrix} x \\ y \\ \theta \end{bmatrix}_t = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}_{t-1} + \begin{bmatrix} \cos\theta\Delta t & 0 \\ \sin\theta\Delta t & 0 \\ 0 & \Delta t \end{bmatrix} \begin{bmatrix} v + \eta_v \\ \omega + \eta_\omega \end{bmatrix}_t$$

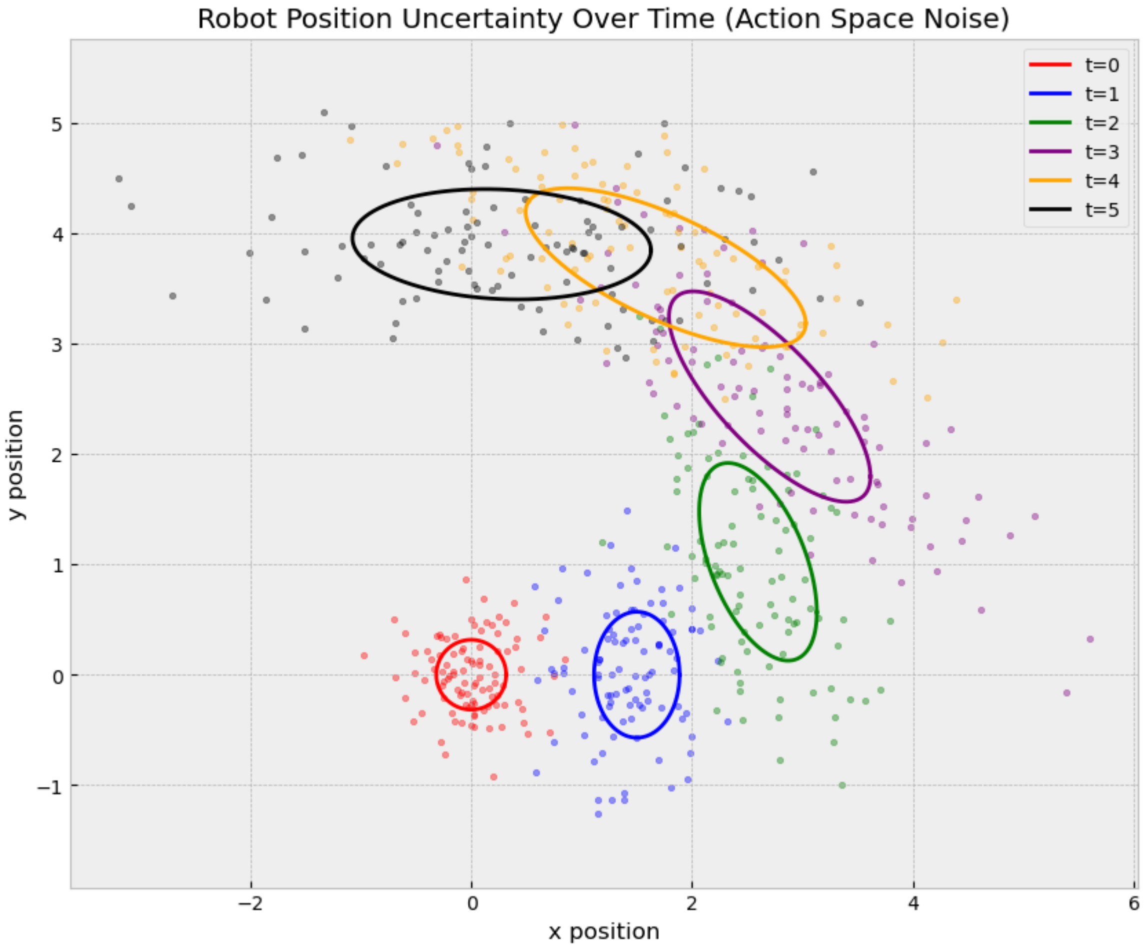
PDF for initial state:

$$\begin{bmatrix} \eta_v \\ \eta_\omega \end{bmatrix}_t \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0.2 & 0 \\ 0 & 0.01 \end{bmatrix} \right)$$

In [22]:
```
dt = 0.5
control = np.array([3, 1.5])  # [velocity, angular velocity]
noise_cov_action = np.array([[0.2, 0],  # Action space noise covari
                             [0, 0.01]])
initial_state = np.array([0, 0, 0])  # [x, y, theta]
initial_cov = np.array([[0.1, 0, 0],
                        [0, 0.1, 0],
                        [0, 0, 0.1]])

def plot_ellipse(mean, cov, color='red', label=None):
    """Plot uncertainty ellipse"""
    # Generate points on a circle
    angles = np.linspace(0, 2*np.pi, 100)
```

```
    circle = np.array([np.cos(angles), np.sin(angles)])

    # Transform circle into ellipse
    L = np.linalg.cholesky(cov[:2, :2])
    ellipse = (L @ circle + mean[:2, np.newaxis]).T

    plt.plot(ellipse[:, 0], ellipse[:, 1], color=color, label=label

def propagate_state(n_steps=5):
    """Propagate robot state and uncertainty"""
    states = []
    mean = initial_state
    cov = initial_cov
    v, w = control

    states.append((mean.copy(), cov.copy()))

    # Propagate for n steps
    for t in range(n_steps):
        theta = mean[2]

        # Control input matrix B
        B = np.array([
            [np.cos(theta) * dt, 0],
            [np.sin(theta) * dt, 0],
            [0, dt]
        ])

        # Jacobian of state transition w.r.t. state
        G = np.array([
            [1, 0, -v * np.sin(theta) * dt],
            [0, 1, v * np.cos(theta) * dt],
            [0, 0, 1]
        ])

        mean = mean + B @ control

        cov = G @ cov @ G.T + B @ noise_cov_action @ B.T
        states.append((mean.copy(), cov.copy()))

    return states

plt.figure(figsize=(10, 8))
colors = ['red', 'blue', 'green', 'purple', 'orange', 'black']

states = propagate_state()
for t, (mean, cov) in enumerate(states):
    color = colors[t]
    plot_ellipse(mean, cov, color=color, label=f"t={t}")

    # Add some random samples to show uncertainty
    samples = np.random.multivariate_normal(mean[:2], cov[:2, :2],
    plt.scatter(samples[:, 0], samples[:, 1], color=color, alpha=0.

plt.grid(True)
plt.xlabel('x position')
```

```
plt.ylabel('y position')
plt.title('Robot Position Uncertainty Over Time (Action Space Noise
plt.legend()
plt.axis('equal')
plt.show()
```



Linear Systems

State Noise (C)

- **Characteristics**: Direct uniform noise addition to state space at each timestep
- **Behavior**:
  - Uniform and consistent variance increase over time
  - Nearly circular (isotropic) isocontour growth pattern
  - Linear error accumulation without correlation effects
  - Predictable spread in all directions
- **Impact**: Represents the simplest case of noise propagation, serving as a baseline for comparison

Control Noise (D)

- **Characteristics**: Noise applied specifically to control inputs (velocity and angular velocity components)
- **Behavior**:

- Asymmetric covariance growth patterns
- Formation of distinctly elongated elliptical distributions
- Progressive error accumulation through control channels
- Directionally-dependent uncertainty propagation

- **Impact**: Demonstrates how control uncertainty can lead to structured, non-uniform prediction errors

Nonlinear Systems

State Noise (E)

- **Characteristics**: System influenced by nonlinear dynamic effects, particularly rotation-dependent phenomena
- **Behavior**:
  - Distinctly curved distribution patterns
  - Abnormally stretched and distorted isocontours
  - Highly asymmetric distribution development
  - Non-uniform error propagation
- **Impact**: Prediction accuracy significantly deteriorates due to complex interaction between noise and nonlinear dynamics

Control Noise (F)

- **Characteristics**: Most complex case combining control noise effects with nonlinear system transitions
- **Behavior**:
  - Severely deflected and distorted elliptical patterns
  - Chaotic point density distributions
  - Maximum uncertainty growth among all cases
  - Complex error propagation pathways
- **Impact**:
  - Highest sensitivity to error accumulation
  - Most unpredictable long-term behavior
  - Requires sophisticated estimation techniques
- **Implications**:
  - Traditional control strategies may be insufficient
  - Need for robust control methods that account for both nonlinearity and noise
  - Important considerations for practical system design

Overall Conclusions

1. Linear systems show predictable error growth patterns, whether from state or control noise

2. Nonlinear systems exhibit fundamentally more complex uncertainty propagation
3. Control noise tends to create more structured uncertainty patterns than state noise
4. The combination of nonlinearity and control noise presents the greatest challenge for prediction and control
5. Understanding these patterns is crucial for designing appropriate estimation and control strategies

Note: In some portions of this document (not exceeding 15% of the entire text) Artificial Intelligence assistant, particularly Generative AI, has been used to rephrase, shorten, or summarize the content. The technologies used include Claude 3.5-Sonnet and Perplexity.