Homework 4 - Question 1 - Luke Arend

In [1]:
```
import numpy as np
import seaborn as sns
from matplotlib import pyplot as plt
sns.set_style('darkgrid')
```

Middleville is a town of families, each with exactly two children. Each child can have either blue eyes or green eyes, and a family can have any combination of blue-eyed or green-eyed children. In this problem, you'll use Matlab to simulate this situation and compute approximate solutions.

a)

Create a function `Bernoulli(alpha,M,N)` that returns an MxN matrix of independently and randomly selected 0s and 1s, where the probability of a 1 is alpha (i.e., the function should generate MxN samples from the Bernoulli distribution with parameter alpha formatted into a MxN matrix).

In [2]:
```
def bernoulli(alpha, M, N):
    X = np.random.rand(M, N) < alpha
    return X.astype('int')
```

In [3]:
```
bernoulli(0.25, 3, 8)
```

Out[3]:
```
array([[0, 0, 0, 1, 0, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 1, 0],
       [0, 0, 0, 0, 1, 0, 0, 0]])
```

b)

Use your function to generate an example of 10 Middleville families (a 10x2 matrix), assuming `alpha=0.5`.

In [4]:
```
families = bernoulli(0.5, 10, 2)
families.T
```

Out[4]:
```
array([[0, 0, 1, 1, 1, 0, 0, 1, 1, 0],
       [1, 0, 0, 0, 0, 0, 1, 0, 1, 0]])
```

Compute a vector containing the indices of the families that have at least one blue-eyed child.

In [5]:
```
blue_families = np.where(np.sum(families, axis=1) >= 1)[0]
blue_families
```

Out[5]:
```
array([0, 2, 3, 4, 6, 7, 8])
```

How many of these are there (as a fraction of the total number of families)?

In [6]:
```
frac_blue = len(blue_families) / len(families)
frac_blue
```

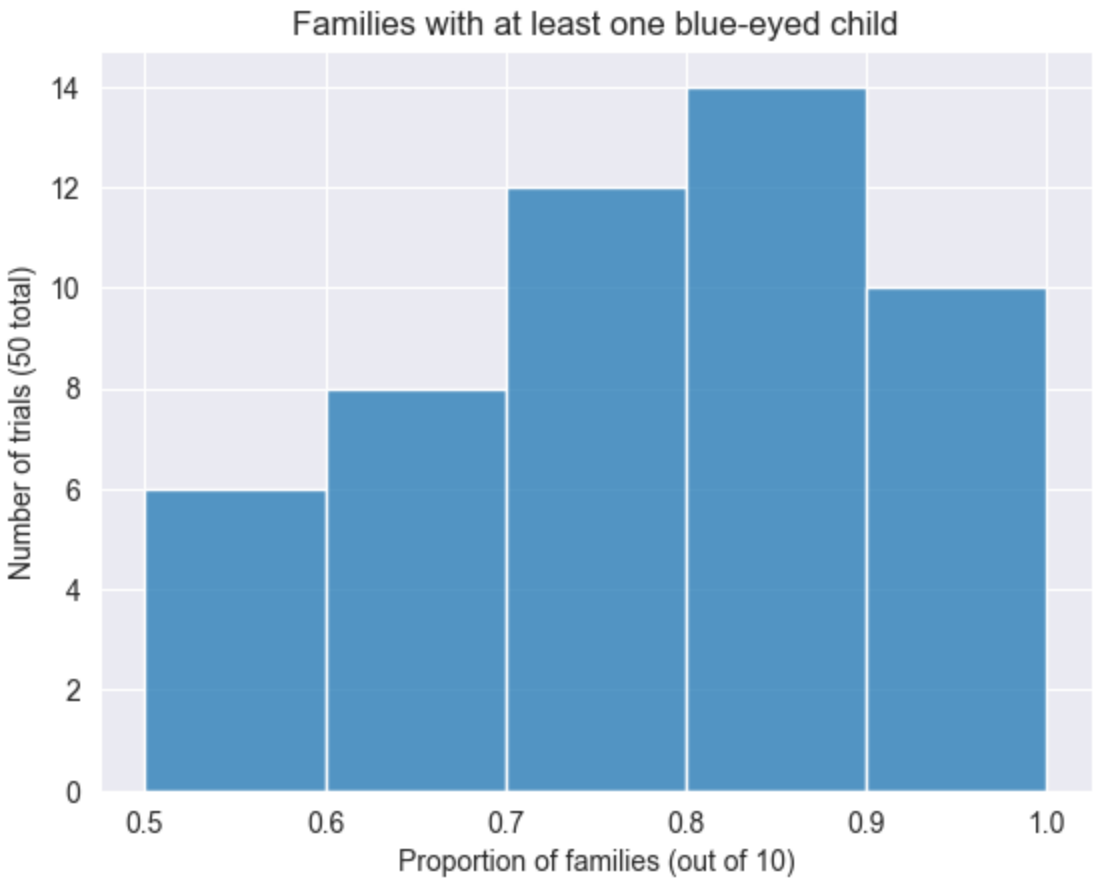
Out[6]:
```
0.7
```

Do this 50 times, computing the proportion containing at least one blue-eyed child for each. Plot a histogram of these 50 values. What is the average value? The standard deviation?

In [7]:
```
def sample_middleville(nfamilies=10, binsize=0.1):
    fracs = []
    for i in range(50):
        families = bernoulli(0.5, nfamilies, 2)
        blue_families = np.where(np.sum(families, axis=1) >= 1)[0]
        frac_blue = len(blue_families) / len(families)
        fracs.append(frac_blue)

    print(f"Average value: {np.mean(fracs)}")
    print(f"Standard deviation: {np.std(fracs)}")
    sns.histplot(fracs, binwidth=binsize)
    return np.std(fracs)
```

In [8]:
```
std_10 = sample_middleville(nfamilies=10)
plt.title("Families with at least one blue-eyed child")
plt.ylabel("Number of trials (50 total)")
plt.xlabel(f"Proportion of families (out of 10)")
plt.show()
```

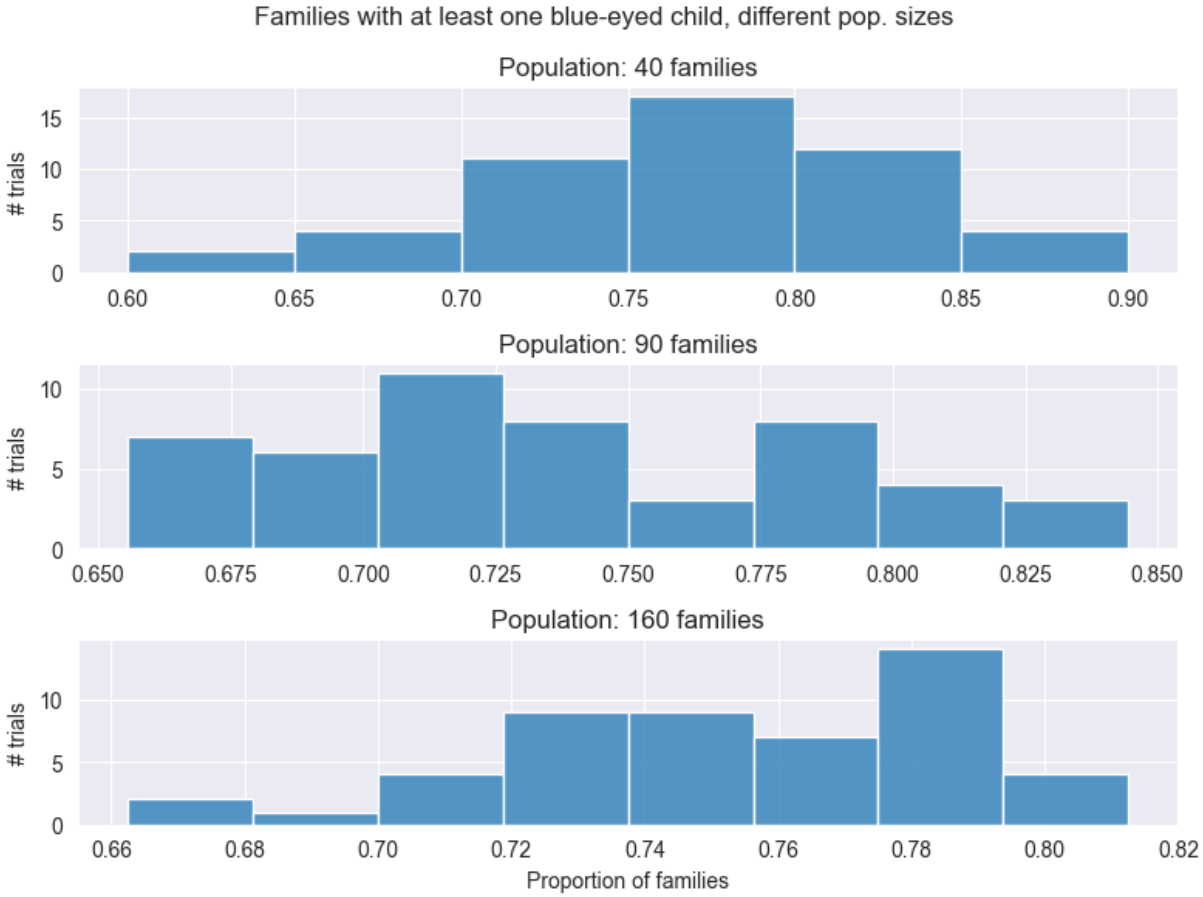
```
Average value: 0.7339999999999999
Standard deviation: 0.13800000000000004
```



Now do this all again, but for populations of 40 families, 90 families, and 160 families. What average and standard deviation do you measure for each of these population sizes?

In [9]:
```
fig, axs = plt.subplots(3, 1, figsize=(8, 6))
plt.suptitle('Families with at least one blue-eyed child, different pop. siz
print(f"Population of 40 families:")
plt.sca(axs[0])
std_40 = sample_middleville(40, binsize=0.05)
plt.title('Population: 40 families')
plt.ylabel('# trials')
print(f"\nPopulation of 90 families:")
plt.sca(axs[1])
std_90 = sample_middleville(90, binsize=0.025)
plt.title('Population: 90 families')
plt.ylabel('# trials')
print(f"\nPopulation of 160 families:")
plt.sca(axs[2])
std_160 = sample_middleville(160, binsize=0.02)
plt.title('Population: 160 families')
plt.ylabel('# trials')
plt.xlabel('Proportion of families')
plt.tight_layout()
```

```
Population of 40 families:
Average value: 0.759
Standard deviation: 0.06378871373526825

Population of 90 families:
Average value: 0.7388888888888888
Standard deviation: 0.048495895206211545

Population of 160 families:
Average value: 0.7506249999999999
Standard deviation: 0.03460152633916602
```



In general, what happens to the average and standard deviation as the number of families in the population grows?

As the number of families in the population grows, the **average stays the same** but the **standard deviation drops**.

In [10]:
```
std = [std_10, std_40, std_90, std_160]
inv_sqrt = 1 / np.sqrt([10, 40, 90, 160])
inv_sqrt / std
```

Out[10]:
```
array([2.29150555, 2.47871251, 2.17357067, 2.28478191])
```

In particular, the standard deviation is inversely proportional to the square root of the sample size.

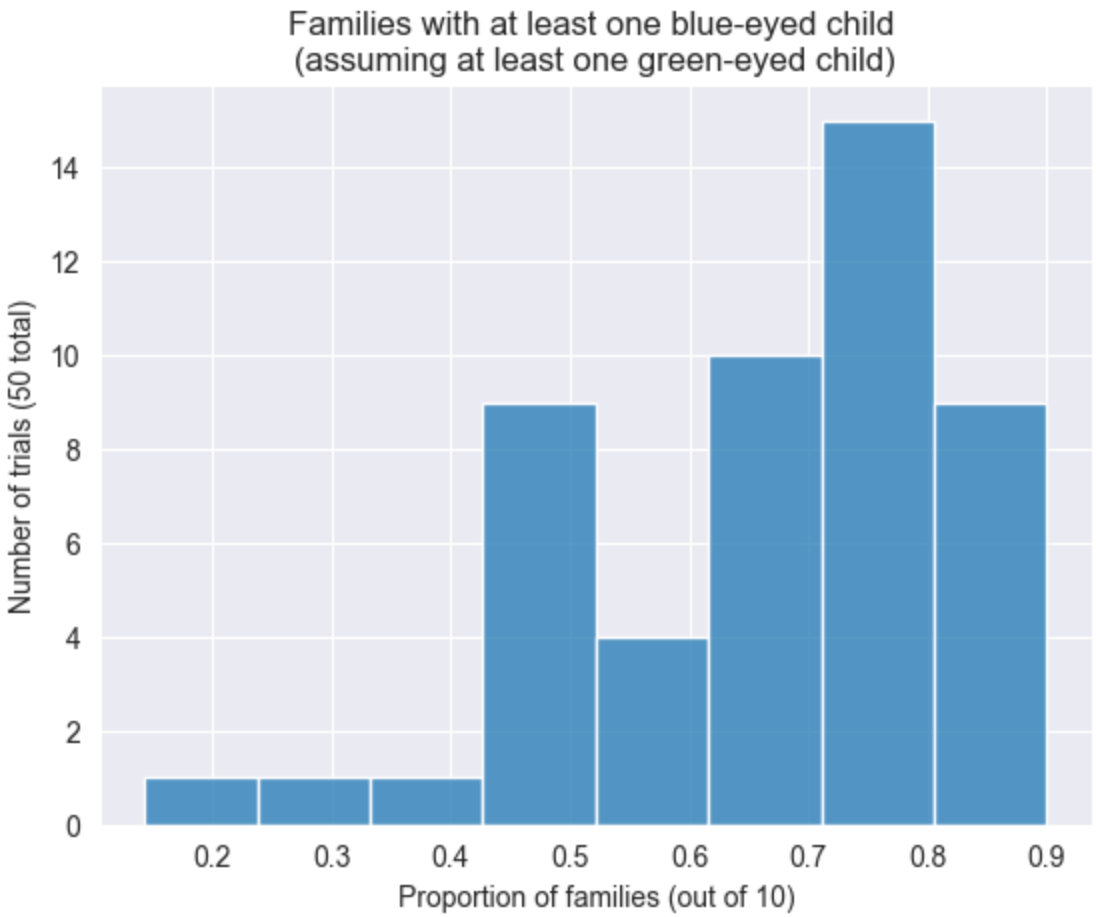
c)

Now consider conditional probability $P[A|B]$ where the event A is "the family has one or more green-eyed child" and the event B is "the family has one or more blue-eyed child". What is the value of this (again, assuming `alpha=0.5`). Now estimate this from a simulated population (as in previous part), in two different ways. First, find the indices of all families satisfying B, make a new matrix containing these, and then compute the proportion of these that satisfy A.

In [11]:
```
def estimate_method_1(alpha, nfamilies):
    families = bernoulli(alpha, nfamilies, 2)
    B_inds = np.where(np.sum(families, axis=1) >= 1)[0]
    B_families = families[B_inds, :]
    AB_inds = np.where(np.sum(B_families, axis=1) <= 1)[0]
    frac = len(AB_inds) / len(B_families)
    return frac
```

In [12]:
```
fracs = []
for i in range(50):
    frac = estimate_method_1(alpha=0.5, nfamilies=10)
    fracs.append(frac)

print(f"Average value: {np.mean(fracs)}")
print(f"Standard deviation: {np.std(fracs)}")

sns.histplot(fracs, binwidth=0.1)
plt.title("Families with at least one blue-eyed child \n(assuming at least c
plt.ylabel("Number of trials (50 total)")
plt.xlabel(f"Proportion of families (out of 10)")
plt.show()
```

```
Average value: 0.6616904761904762
Standard deviation: 0.15997689368460546
```



In [13]:
```
estimate_method_1(alpha=0.5, nfamilies=100)
```

Out[13]:
```
0.6756756756756757
```

In [14]:
```
estimate_method_1(alpha=0.5, nfamilies=1000)
```

Out[14]:
```
0.6640419947506562
```

In [15]:
```
estimate_method_1(alpha=0.5, nfamilies=10000)
```

Out[15]:
```
0.6705616783959634
```

By running on various size populations we see that **method 1 estimates** $P[A|B] \approx 0.67$.

Second, use the definition of conditional probability: count the number of families satisfying both A and B, and then dividing by the number satisfying B. Convince yourself that these compute the same value by running them both on some large populations. As in 1B, run one of these methods on 50 populations of 10 families, and plot a histogram of the estimated values. Re-compute for a population of 10,000 families.

In [16]:
```
def estimate_method_2(alpha, nfamilies):
    families = bernoulli(alpha, nfamilies, 2)
```

```
    both_inds = np.where(np.sum(families, axis=1) == 1)[0]
    B_inds = np.where(np.sum(families, axis=1) >= 1)[0]
    # both_inds = set(A_inds) & set(B_inds)
    frac = len(both_inds) / len(B_inds)
    return frac
```

In [17]:
```
estimate_method_2(alpha=0.5, nfamilies=100)
```

Out[17]:
```
0.6
```

In [18]:
```
estimate_method_2(alpha=0.5, nfamilies=1000)
```

Out[18]:
```
0.6431535269709544
```

In [19]:
```
estimate_method_2(alpha=0.5, nfamilies=10000)
```

Out[19]:
```
0.6667111051859752
```

By running on various size populations we see that **method 2 also estimates** $P[A|B] \approx 0.67$.

In [ ]:
```

```

Homework 4 - Question 2 - Luke Arend

In [1]:
```
import math
import numpy as np
import seaborn as sns
from matplotlib import pyplot as plt
sns.set_style('darkgrid')
```

The Poisson distribution is commonly used to model neural spike counts:

$$p(k) = \frac{\mu^k e^{-\mu}}{k!},$$

where k is the spike count (over some specified time interval), and μ is the expected number of spikes over that interval.

(a)

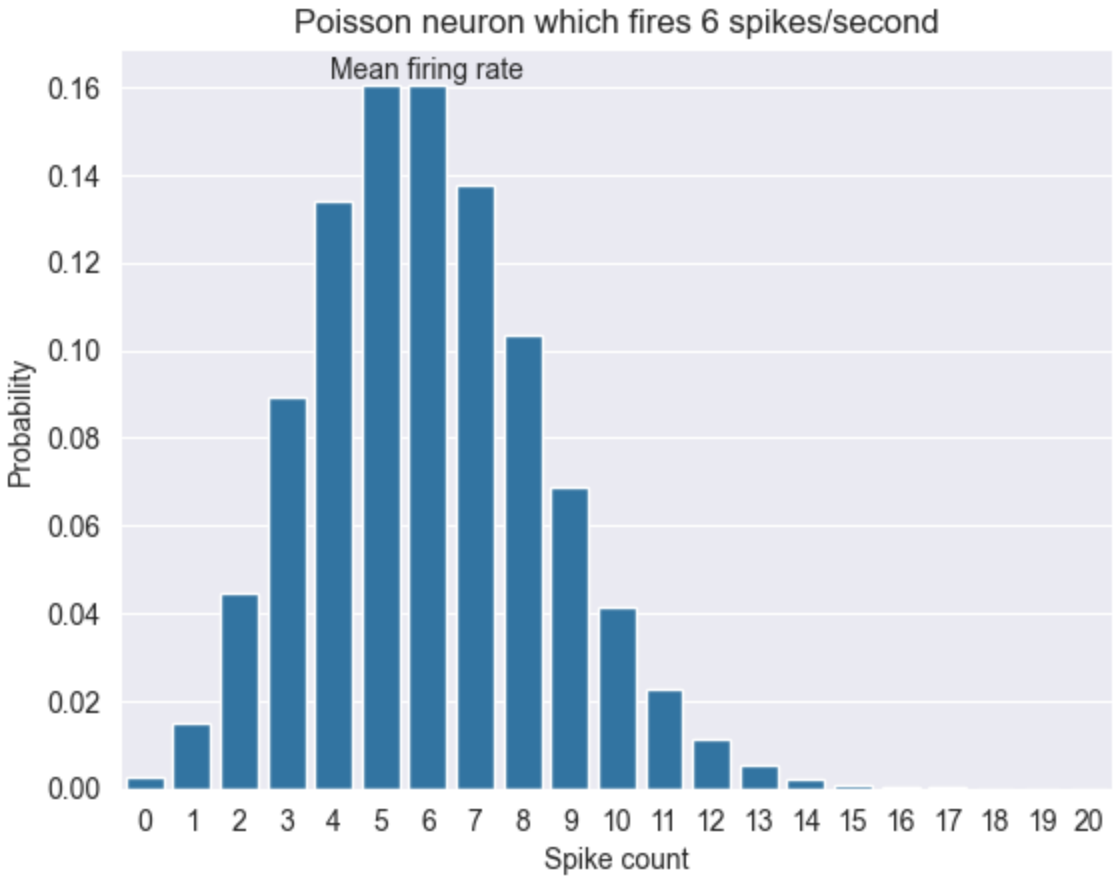
We would like to know what the Poisson distribution looks like. Set the expected number of spikes to μ = 6 spikes/interval then create a vector p of length 21, whose elements contain the probabilities of Poisson spike counts for $k = [0...20]$. Since we're clipping the range at a maximum value of 20, you'll need to normalize the vector so it sums to one (the distribution given above is normalized over the range from 0 to infinity) to make the vector p represent a valid probability distribution.

In [2]:
```
mu = 6 # spikes/interval
p = np.zeros(21)
for k in range(21):
    p[k] = mu ** k * np.exp(-mu) / math.factorial(k)
p /= np.sum(p) # normalize
np.sum(p)
```

Out[2]:
```
1.0000000000000002
```

Plot p in a bar plot and mark the mean firing rate. Is it equal to μ? Why or why not?

In [3]:
```
mean_p = p @ np.arange(21)
sns.barplot(x=np.arange(len(p)), y=p)
plt.text(mean_p, max(p), "Mean firing rate", ha='center', va='bottom')
plt.title('Poisson neuron which fires 6 spikes/second')
plt.ylabel('Probability')
plt.xlabel('Spike count')
plt.show()
```



In [4]:
```
np.allclose(mu, mean_p)
mean_p
```

Out[4]: 5.999977649595796

μ and the mean firing rate measured from the PDF are both equal to 6. We defined μ as the expected number of spikes per unit interval. In the limit of many unit intervals, on average we observe μ spikes each time. μ is equal to the mean firing rate by definition.

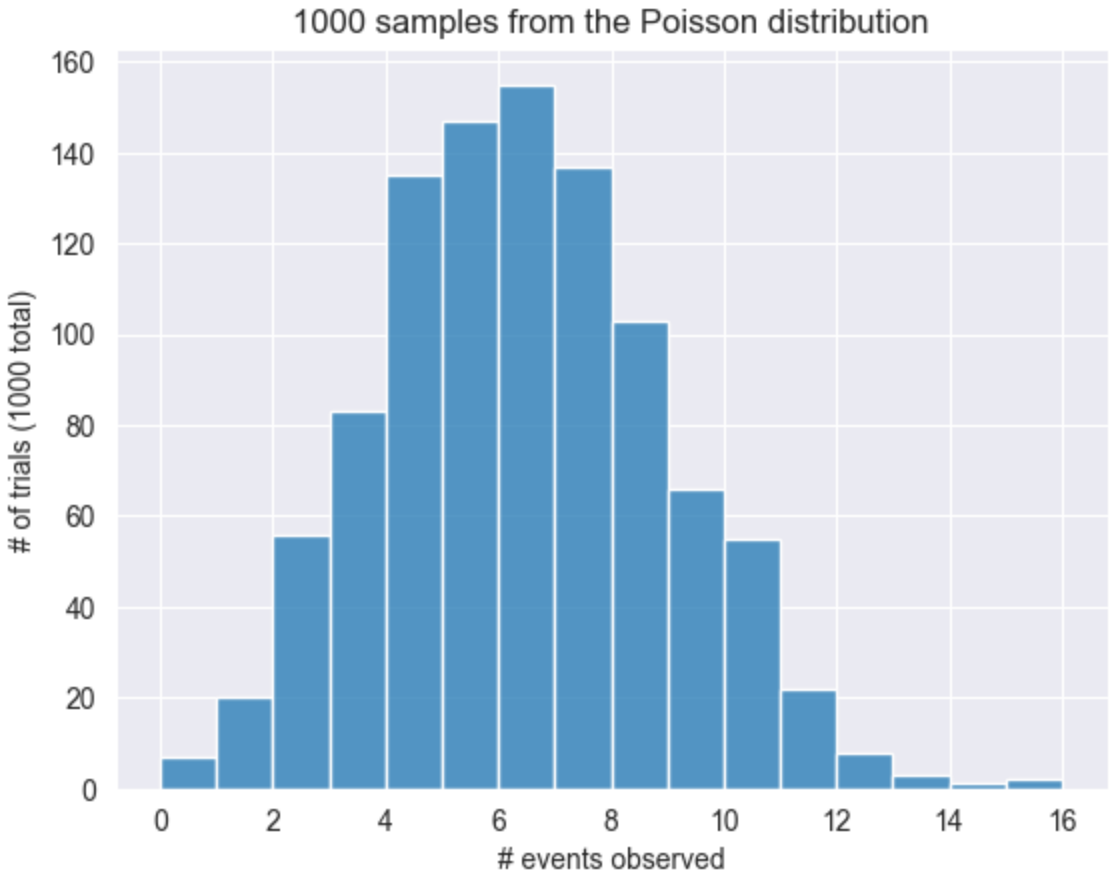
(b)

Generate samples from the Poisson distribution where each sample represents the number of spikes and ranges from 0 to 20. To simplify the problem, use a clipped Poisson vector p to write a function `samples = randp(p, num)` that generates num samples from the probability distribution function (PDF) specified by p. [Hint: use the `rand` function, which generates real values over the interval $[0...1]$, and partition this interval into portions proportional in size to the probabilities in p].

In [5]:
```
def randp(p, num):
    cdf = np.cumsum(p)
    randnums = np.random.rand(num)
    samples = [np.sum(cdf < x) for x in randnums]
    return np.array(samples)
```

Test your function by drawing 1,000 samples from the Poisson distribution in (a), plotting a histogram of how many times each value is sampled, and comparing this to the frequencies predicted by p.

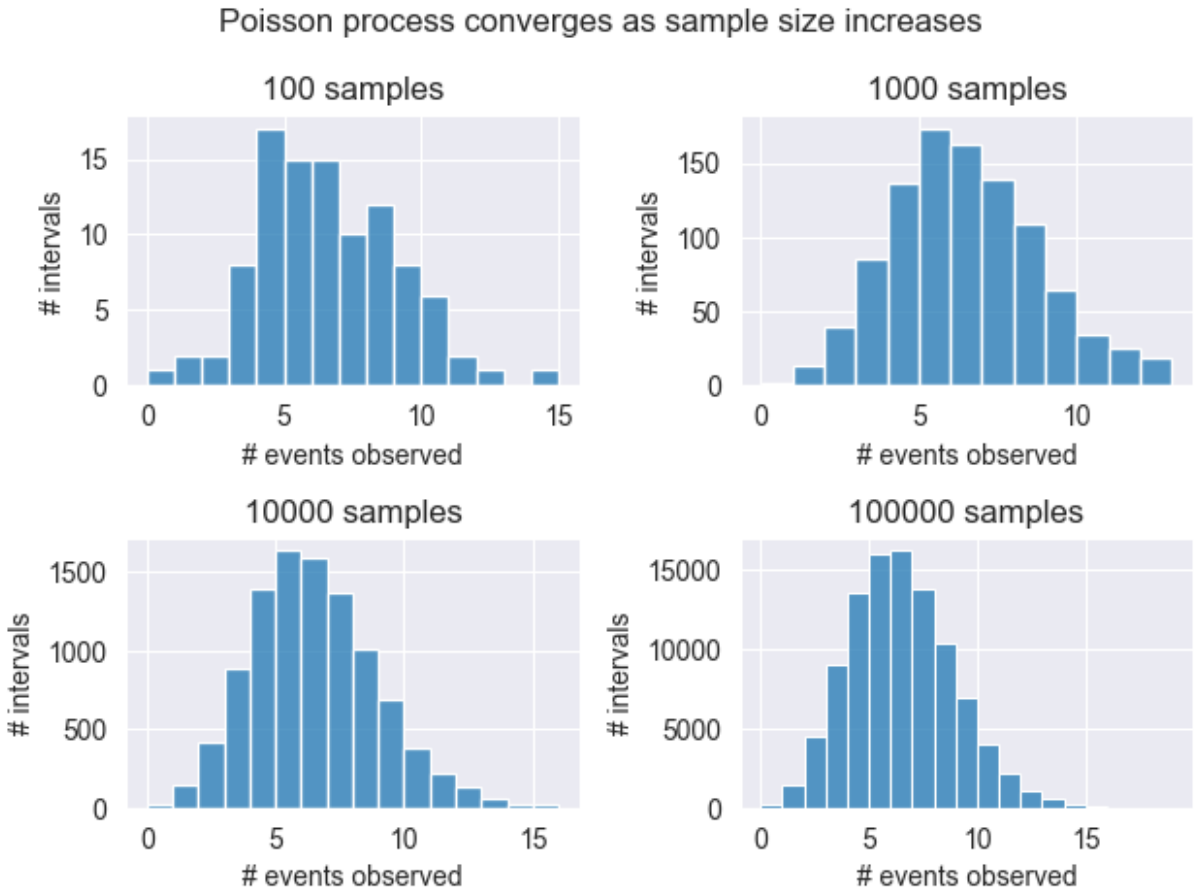
In [6]:
```
samples = randp(p, 1000)
sns.histplot(samples, binwidth=1)
plt.title('1000 samples from the Poisson distribution')
plt.ylabel('# of trials (1000 total)')
plt.xlabel('# events observed')
plt.show()
```



Verify qualitatively that the answer gets closer (converges) as you increase the number of samples (try 10 raised to powers $[2, 3, 4, 5]$).

In [7]:
```
fig, axs = plt.subplots(2, 2)
plt.suptitle('Poisson process converges as sample size increases')
for i, pow in enumerate([2, 3, 4, 5]):
    nsamples = 10 ** pow
    samples = randp(p, nsamples)
    plt.sca(axs.flat[i])
```

```
    sns.histplot(samples, binwidth=1)
    plt.title(f'{nsamples} samples')
    plt.ylabel(f'# intervals')
    plt.xlabel('# events observed')
plt.tight_layout()
```



(c)

Imagine you're recording with an electrode from two neurons simultaneously, whose spikes have very similar waveforms (and thus can't be distinguished by the spike sorting software). Create a probability vector, q, for the second neuron, assuming a mean rate of 4 spikes/interval.

In [8]:
```
def poisson_pdf(n, mu):
    p = np.zeros(n)
    for k in range(n):
        p[k] = mu ** k * np.exp(-mu) / math.factorial(k)
    return p / np.sum(p)
```

In [9]:
```
q = poisson_pdf(n=21, mu=4)
```

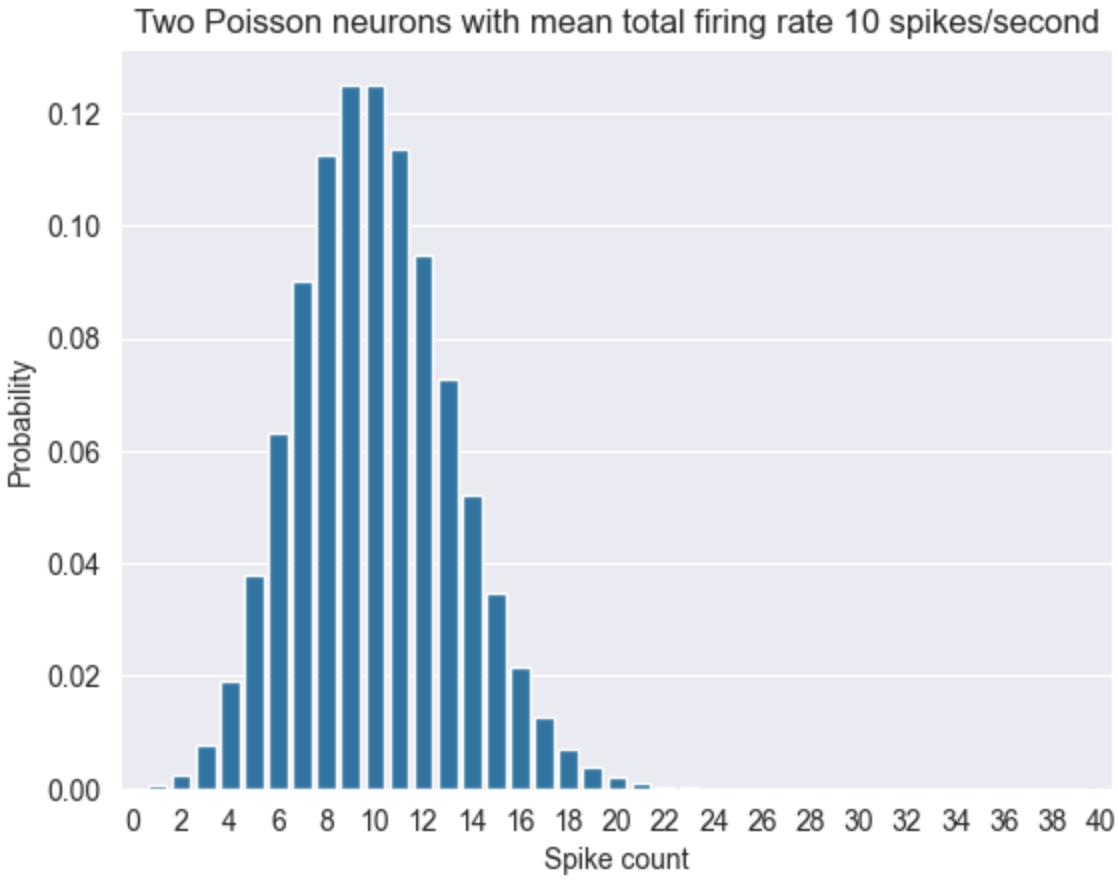
What is the probability distribution of the observed spike counts, which will be the sum of spike counts from the two neurons derived from p and q? [Hint: the output vector should have length $m + n - 1$ when m and n are the lengths of the two input PDFs.

This is because the maximum spike count will be bigger than the maximum of each respective individual neuron.]

A property of Poisson distributions is that a sum of two independent Poisson random variables is also Poisson. In particular, if X_A is Poisson-distributed with mean firing rate μ_A and X_B is also Poisson with mean firing rate μ_B, then $X_C = X_A + X_B$ is Poisson with mean firing rate $\mu_C = \mu_A + \mu_B$.

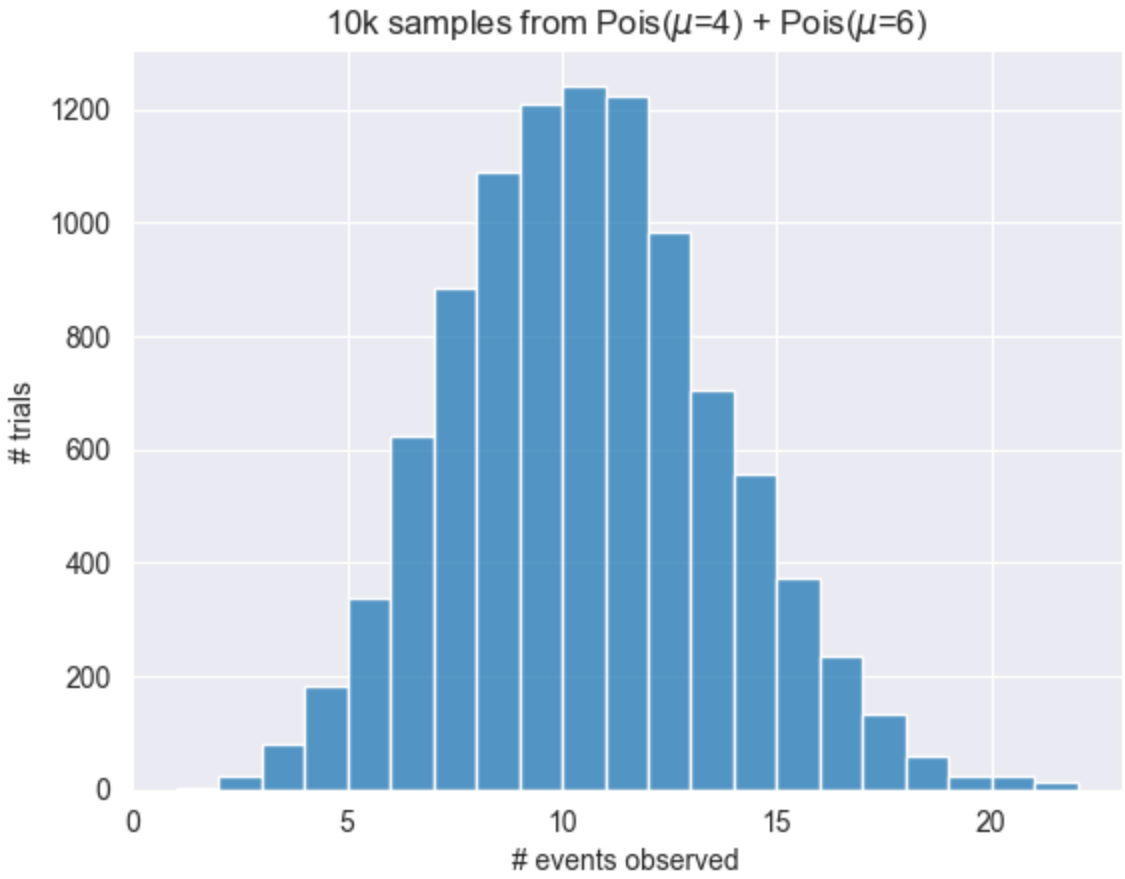
In [10]:
```
mu_a = 6
mu_b = 4
mu_c = mu_a + mu_b
n = len(p) + len(q) - 1
poisson_c = poisson_pdf(n, mu_c)
```

In [11]:
```
x = np.arange(len(poisson_c))
sns.barplot(x=x, y=poisson_c)
plt.title('Two Poisson neurons with mean total firing rate 10 spikes/second'
plt.xticks(np.arange(0, 41, 2))
plt.ylabel('Probability')
plt.xlabel('Spike count')
plt.show()
```



Verify your answer by comparing it to the histogram of 1,000 samples generated by summing two calls to `randp` (choose a big enough number of samples!).

In [12]:
```
X_a = randp(p, 10000)
X_b = randp(q, 10000)
X_c = X_a + X_b
sns.histplot(X_c, binwidth=1)
plt.title('10k samples from Pois($\mu$=4) + Pois($\mu$=6)')
plt.ylabel('# trials')
plt.xlabel('# events observed')
plt.show()
```

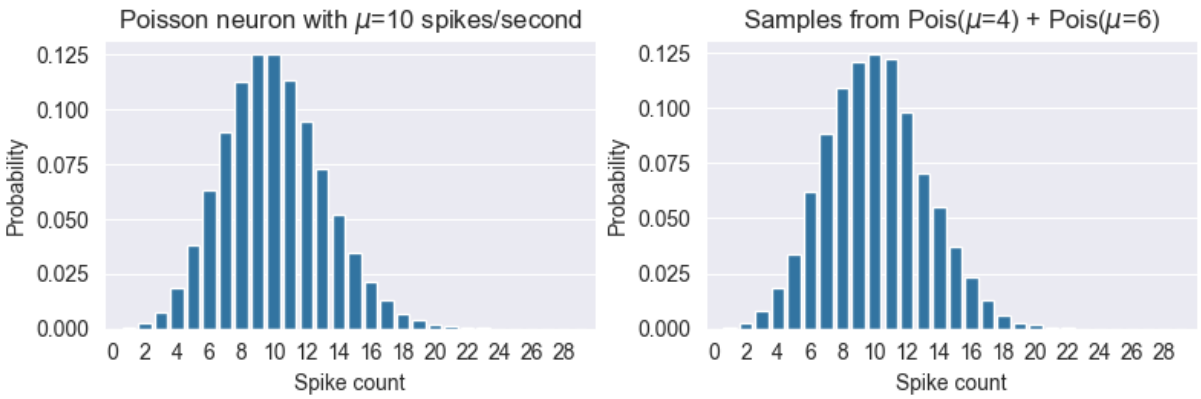


(d)

Now imagine you are recording from a neuron with mean rate 10 spikes/interval (the sum of the rates from the neurons above). Plot the distribution of spike counts for this neuron, in comparison with the distribution of the sum of the previous two neurons.

In [13]:
```
nbins = 30
poisson_c = poisson_pdf(n=nbins, mu=10)
fig, axs = plt.subplots(1, 2, figsize=(8, 3))
plt.suptitle('')
plt.sca(axs[0])
x = np.arange(len(poisson_c))
sns.barplot(x=x, y=poisson_c)
plt.xticks(np.arange(0, 31, 2))
plt.title('Poisson neuron with $\mu$=10 spikes/second')
plt.ylabel('Probability')
```

```
plt.xlabel('Spike count')
plt.sca(axs[1])
x = np.arange(nbins)
y = [np.sum(X_c == i) for i in range(nbins)]
y /= np.sum(y)
sns.barplot(x=x, y=y)
plt.xticks(np.arange(0, 31, 2))
plt.title('Samples from Pois($\mu$=4) + Pois($\mu$=6)')
plt.ylabel('Probability')
plt.xlabel('Spike count')
plt.tight_layout()
```



Based on the results of these two experiments, if we record a new spike train, can you tell whether the spikes you have recorded came from one or two neurons just by looking at their distribution of spike counts? Comment about the reason why based on the intuition behind the Poisson distribution.

We **cannot tell whether** we are recording from one or two neurons just by looking at their distribution of spike counts (to the extent the neurons themselves follow a Poisson model). This is because the sum of Poisson distributions is also Poisson with a rate parameter equal to the sum of the two input distributions.

In [ ]:

Homework 4 - Question 3 - Luke Arend

In [1]:
```
import numpy as np
import seaborn as sns
from matplotlib import pyplot as plt
from scipy import stats
sns.set_style('darkgrid')
```

The Central Limit theorem states that the distribution of the average of n independent samples drawn from any fixed distribution with finite mean and variance, gets closer and closer to a Normal (Gaussian) distribution as n increases. Specifically, if the mean and variance of the original distribution are μ and σ, the distribution of $\sqrt{n}(\bar{x} - \mu)/\sigma$ converges to $\mathcal{N}(0, 1)$ as n increases (where $\bar{x}$ is the average of n samples).

(a)

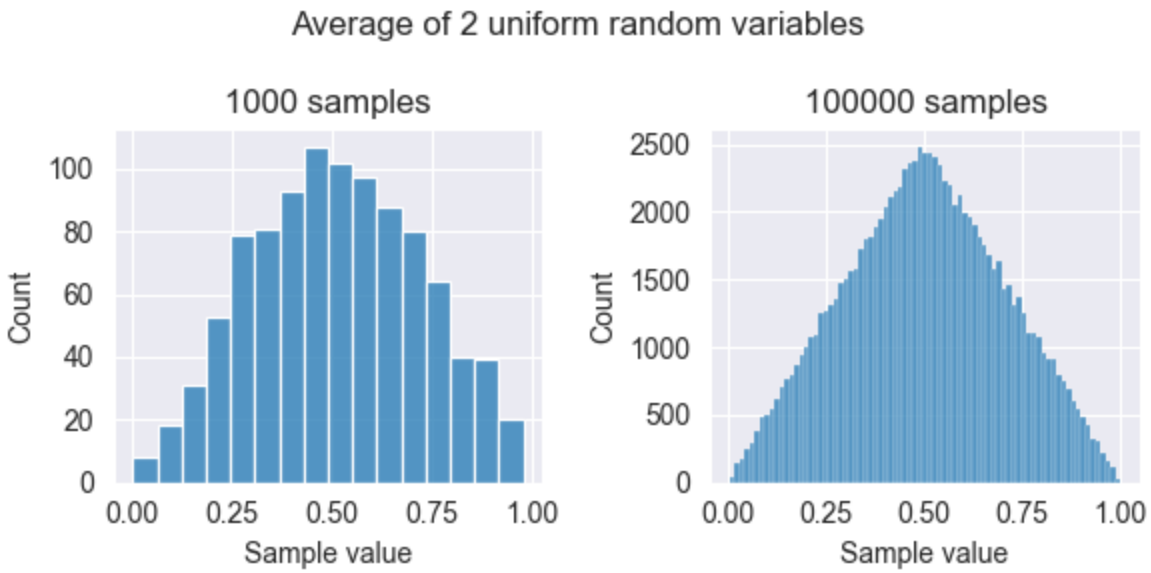
Generate 1,000 samples of two values each from a uniform distribution (use `rand`).

In [2]:
```
X = np.random.rand(1000, 2)
```

Compute the average of each sample (pair of values), and plot a histogram of these. What shape is it, approximately? What shape should it have in the limit, as you gather more and more samples (try with 100,000 samples)? Why?

In [3]:
```
def plot_random_uniforms(n):
    fig, axs = plt.subplots(1, 2, figsize=(6, 3))
    plt.suptitle(f'Average of {n} uniform random variables')
    X_avg = np.random.rand(1000, n).mean(axis=1)
    plt.sca(axs[0])
    sns.histplot(X_avg)
    plt.title('1000 samples')
    plt.xlabel('Sample value')
    plt.sca(axs[1])
    X_avg = np.random.rand(100000, n).mean(axis=1)
    sns.histplot(X_avg)
    plt.title('100000 samples')
    plt.xlabel('Sample value')
    plt.tight_layout()
```

In [4]:
```
plot_random_uniforms(2)
```

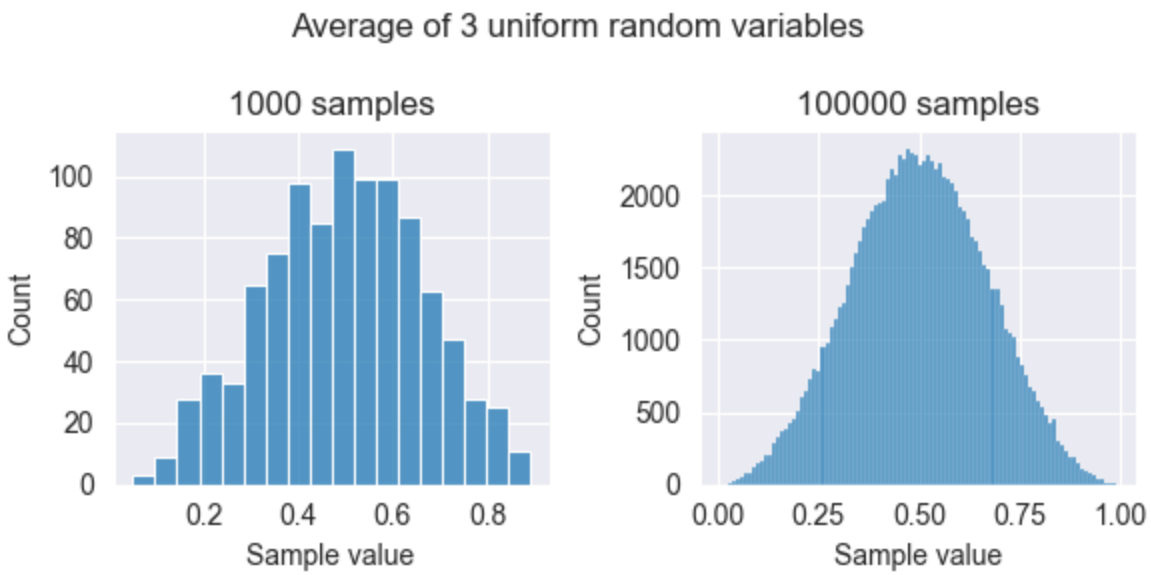


The histogram for 1000 samples has the approximate shape of a triangle. In the limit it should converge to a perfect triangle centered on 0.5. While the two samples' average can be anywhere between 0 and 1, it tends toward the center of the unit line since it must lie between both points. We can also see it by thinking combinatorically: there are many more pairings of samples that average to something around 0.5 than pairings that average to values close to 0 or 1.

(b)

Now try this again with samples containing 3 values. How has the histogram changed?

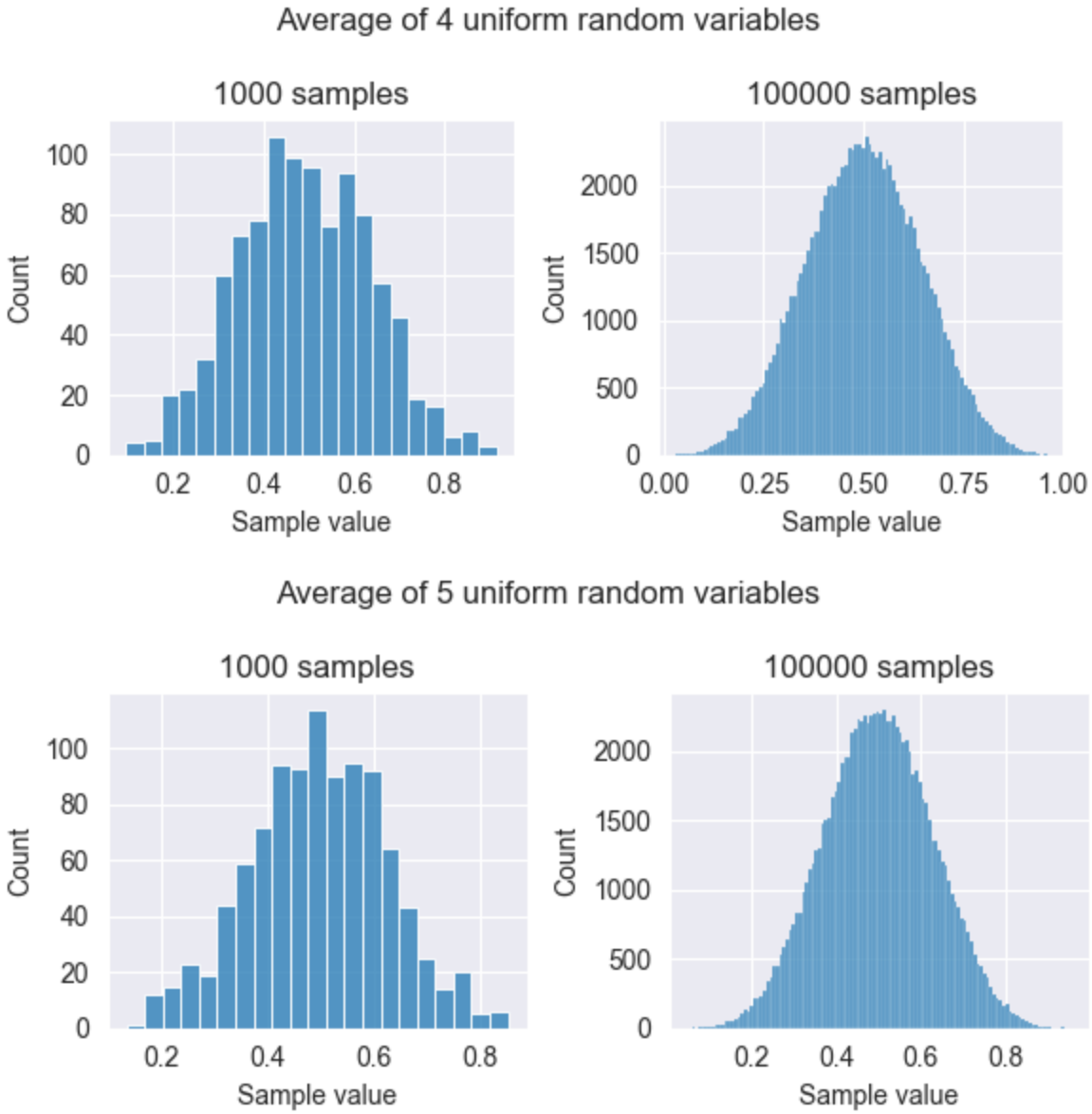
In [5]:
```
plot_random_uniforms(3)
```

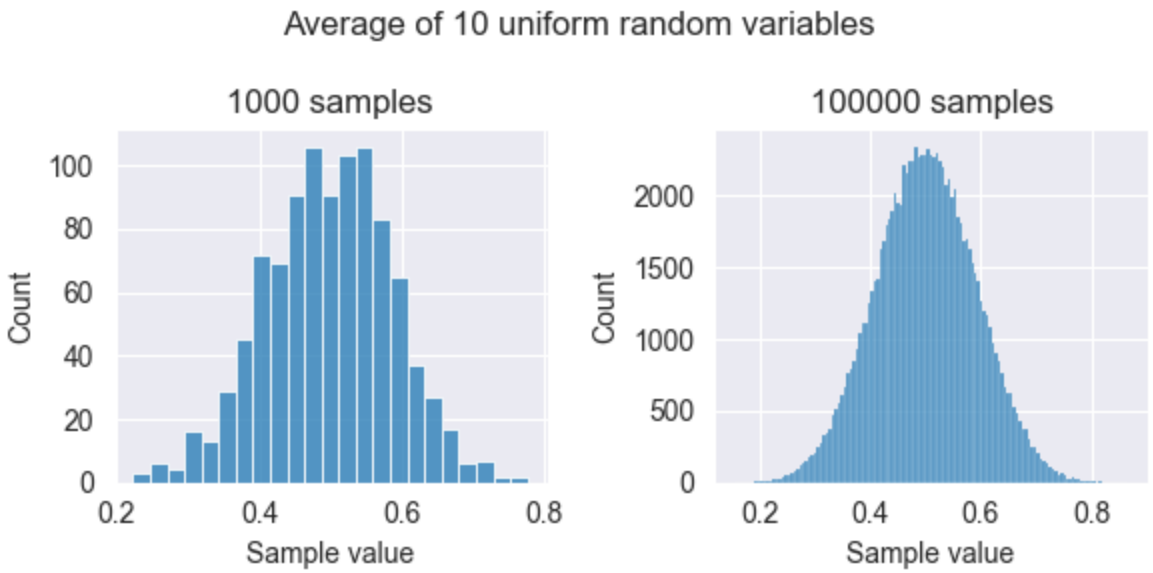


Now the triangle has softened into something more like a bell curve.

Try sample sizes of 4 and 5 as well. When do you judge that the histogram starts looking Normal?

In [6]:
```
plot_random_uniforms(4)
plot_random_uniforms(5)
plot_random_uniforms(10)
```





In my estimation the histogram doesn't look that much more normal after we've hit n = **4 or 5**. So in this case it only takes a relatively small number of random variables before their average begins to look Gaussian.

(c)

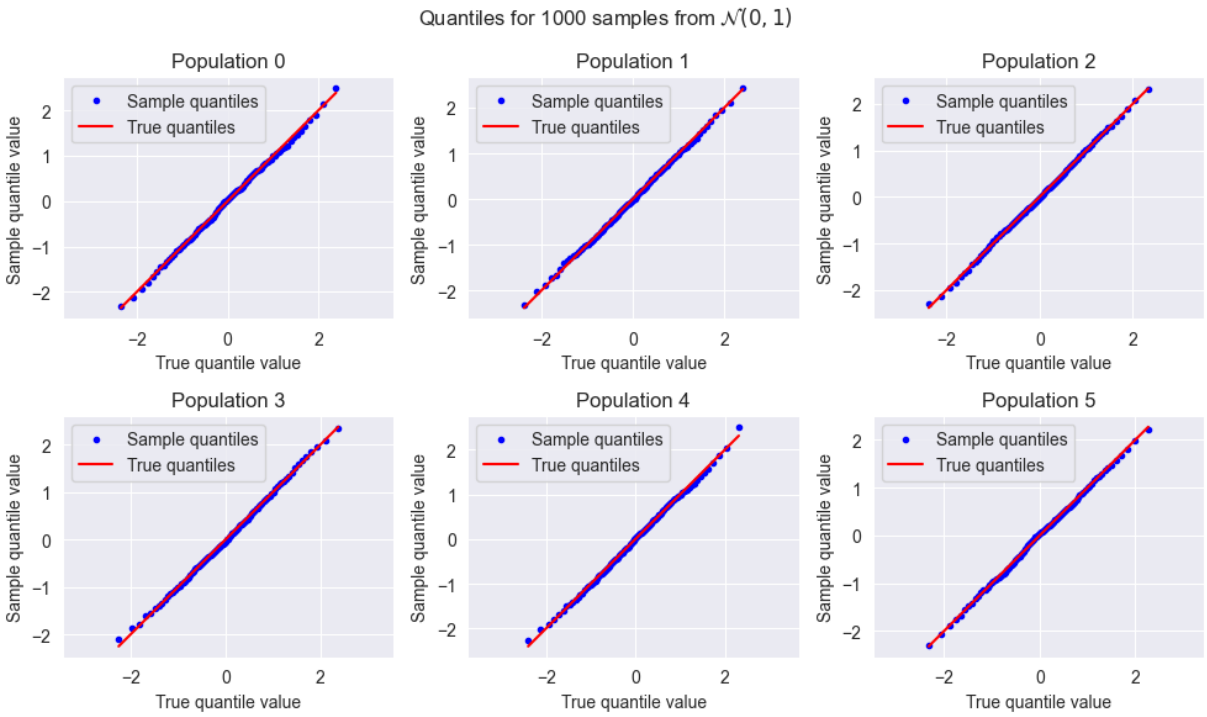
Test the Normality of the distribution a bit more carefully, using a "Q-Q" (quantile-quantile) plot (plot the quantiles of one distribution against another). If the two distributions match, the values should lie on a unit-slope line. For this problem, you can use the matlab function `normplot`, which plots the quantiles of a sample of data against those of a Normal distribution of the same mean and variance.

In [40]:
```
def normplot(samples, quantiles, title, xlabel=None, ylabel=None, linelabel=
    sample_quantiles = np.quantile(samples, q=quantiles)
    norm_dist = stats.norm(loc=np.mean(samples), scale=np.std(samples))
    true_quantiles = norm_dist.ppf(quantiles)
    if title:
        plt.title(title)
    linelabel = linelabel if linelabel else 'True quantiles'
    xlabel = xlabel if xlabel else 'True quantile value'
    ylabel = ylabel if ylabel else 'Sample quantile value'
    plt.plot(true_quantiles, sample_quantiles, 'b.', label='Sample quantiles
    plt.plot(true_quantiles, true_quantiles, 'r-', label=linelabel)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.xlim([sample_quantiles.min(), sample_quantiles.max()])
    plt.axis('equal')
    plt.legend()
```

In [41]:
```
quantiles = np.linspace(0.0, 1.0, 101)
```

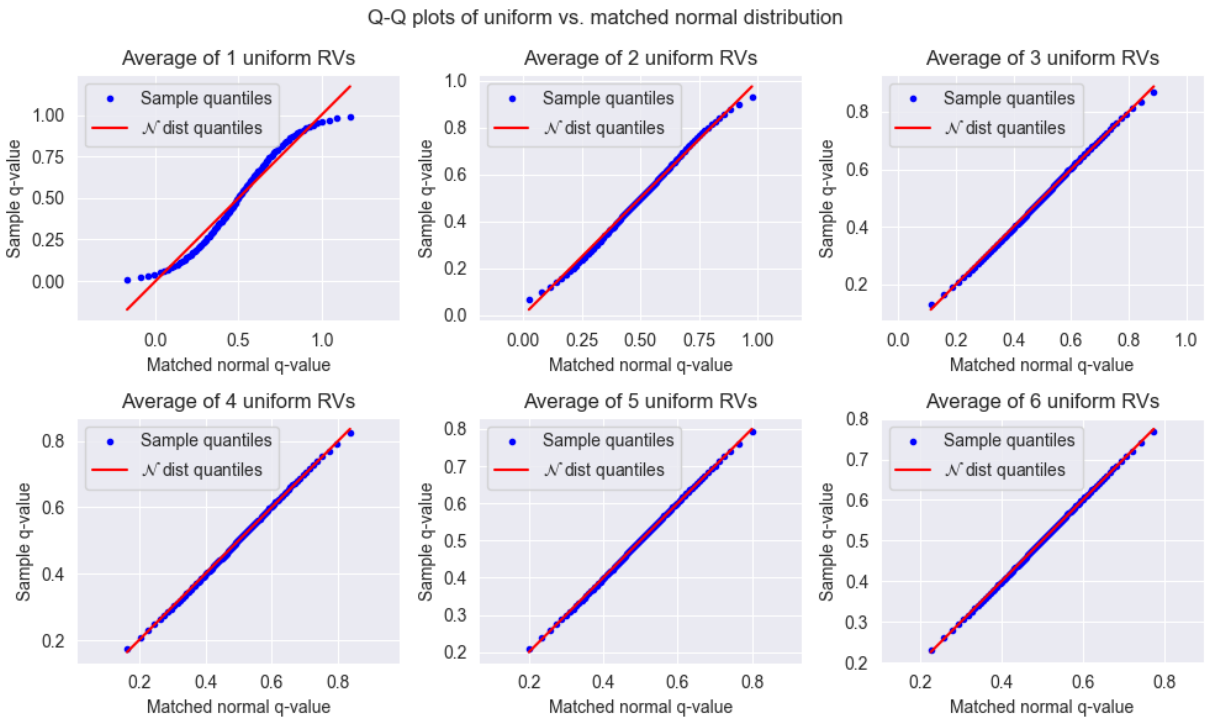
First, try this on a sample of 1,000 values from a normal distribution (use `randn`). The points should fall (close to) a straight line, indicating that the sample is close to normal, as expected. Try this a few times to see how the plot varies (you might want to put them on the same graph, using matlab's `hold on` command).

In [42]:
```
fig, axs = plt.subplots(2, 3, figsize=(10, 6))
plt.suptitle('Quantiles for 1000 samples from $\mathcal{N}(0, 1)$')
for i, ax in enumerate(axs.flat):
    plt.sca(ax)
    X = np.random.randn(1000)
    normplot(X, quantiles, f'Population {i}')
plt.tight_layout()
```



Now call `normplot` on a sample of 1,000 values from a uniform distribution. Explain qualitatively why it has the shape it does (hint: think about the quantiles of the uniform and Normal distributions). Do this for averages of uniform samples of different size (2, 3, 4, ...). Keep increasing sample size until you cannot tell the resulting QQ plot from the QQ plots for samples from the Normal distribution. Roughly how big does the sample have to be?

In [43]:
```
fig, axs = plt.subplots(2, 3, figsize=(10, 6))
plt.suptitle('Q-Q plots of uniform vs. matched normal distribution')
for i, ax in enumerate(axs.flat):
    plt.sca(ax)
    n = i + 1
    X = np.random.rand(1000000, n).mean(axis=1)
    normplot(X, quantiles, f'Average of {n} uniform RVs',
             xlabel='Matched normal q-value', ylabel='Sample q-value',
             linelabel='$\mathcal{N}$ dist quantiles')
plt.tight_layout()
```



The Q-Q plot for samples from a uniform distribution vs. a normal is **S-shaped**. The deviation is especially pronounced toward the extreme quantiles like 0.01 and 0.09. Here, values from the uniform distribution are clipped to be less extreme than the tails of the normal distribution, which may contain values of arbitrary size.

As we increase the number of summed uniform random variables, two things happen:

- The range of quantile values for the sum of RVs drops from [0.0, 1.0] to around [0.2, 0.8] as n increases from 1 to 6. This is because as uniform RVs are added, it becomes increasingly unlikely for their sum to lie near the extremes of the range (that requires all RVs to contribute samples from the same extreme). Intuitively consider die rolls: as the number of dice increases, the probability of "snake eyes" drops.
- The sum of RVs tends toward a normal distribution. To me it becomes indistinguishable once there are around **5 or 6 RVs** in the sum.

In [ ]:

Homework 4 - Question 4 - Luke Arend

In [1]:
```
import numpy as np
import seaborn as sns
from matplotlib import pyplot as plt
```

(a)

Write a function `samples = ndRandn(mean, cov, num)` that generates a set of samples drawn from an N-dimensional Gaussian distribution with the specified mean (an N-vector) and covariance (an NxN matrix). The parameter num should be optional (defaulting to 1) and should specify the number of samples to return. The returned value should be a matrix with num rows each containing a sample of N elements. (Hint: use the MATLAB function `randn` to generate samples from an N-dimensional Gaussian with zero mean and identity covariance matrix X, and then transform these to achieve the desired mean and covariance. Recall that the covariance of $Y = MX$ is $E(YY^T) = MC_XM^T$ where C_X is the covariance of X.)

We want a function which yields samples from a normal random variable with mean μ and covariance C_Y. For this we start with a normal random variable X with μ = 0 and C_X = I (identity matrix). We seek a matrix M with which we can transform X to get Y.

$X = M^{-1}Y$.

$C_Y = MM^T$.

Let C_Y have singular value decomposition $USV^T = C_Y$.

Then $USV^T = MM^T$.

Since C_Y is symmetric, $U = V$ in its singular value decomposition, so $USU^T = MM^T$.

We can freely decompose S into a product of two diagonal matrices each containing the square roots of the values in S: $S = \Lambda\Lambda^T$ where $\Lambda = \sqrt{S}$.

Then we can write $USU^T = U\Lambda\Lambda^TU^T = U\Lambda(U\Lambda)^T = MM^T$. So $M = U\Lambda = U\sqrt{S}$.

The transform $Y = MX$, with $M = U\sqrt{S}$ (from $USV^T = C_Y$) and X drawn from the standard normal, is implemented below.

In [2]:
```
def nd_randn(mean, cov, num=1):
    ndims = len(mean)
```

```
    X = np.random.randn(num, ndims)
    U, s, Vt = np.linalg.svd(cov)
    M = U @ np.diag(np.sqrt(s))
    X = np.random.randn(1000, 2)
    Y = (M @ X.T).T
    Y += mean
    return Y
```

For this, use mean $\mu = [4, 5]$ with $C_Y = [10, -4; -4, 5]$ to sample and scatterplot 1,000 points to verify your function worked as intended.

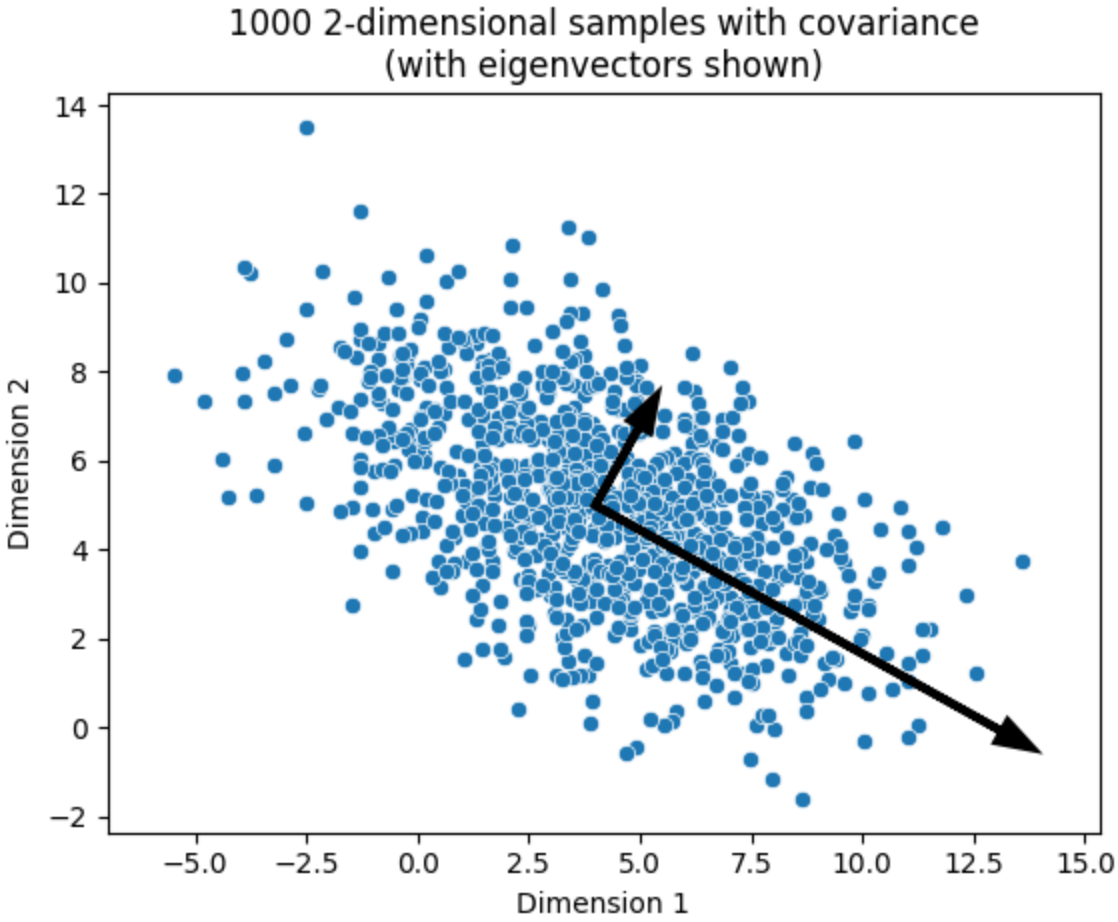
In [3]:
```
mean = np.array([4, 5])
cov = np.array([[10, -4], [-4, 5]])
Y = nd_randn(mean, cov, num=1000)
```

In [5]:
```
mean_Y = np.mean(Y, axis=0)
cov_Y = np.cov(Y.T)
mean_Y, cov_Y
```

Out[5]:
```
(array([4.16000145, 4.8989835 ]),
 array([[ 9.23192352, -3.57934043],
        [-3.57934043,  4.78936171]]))
```

The mean and covariance of our sampled Y are close to the requested mean and covariance $[4, 5]$ and $[[10, -4], [-4, 5]]$). So `nd_randn` worked as expected.

In [7]:
```
sns.scatterplot(x=Y[:, 0], y=Y[:, 1])
(v1, v2), E = np.linalg.eig(np.cov(Y.T))
x1, x2 = E.T
plt.arrow(mean[0], mean[1], v1 * x1[0], v1 * x1[1], color='k',
          linewidth=3, head_width=0.5, length_includes_head=True)
plt.arrow(mean[0], mean[1], v2 * x2[0], v2 * x2[1], color='k',
          linewidth=3, head_width=0.5, length_includes_head=True)
plt.title('1000 2-dimensional samples with covariance\n(with eigenvectors sh
plt.xlabel('Dimension 1')
plt.ylabel('Dimension 2')
plt.axis('equal')
plt.show()
```



(b)

Now consider the marginal distribution of a generalized 2-D Gaussian with mean μ and covariance C in which samples are projected onto a unit vector $\hat{u}$ to obtain a 1-D distribution. Write a mathematical expression for the mean and variance of this marginal distribution as a function of $\hat{u}$.

Let y be the marginal distribution that results from projecting a 2D gaussian $\vec{x} \sim \mathcal{N}(\mu, C)$ onto a unit vector $\hat{u}$.

$y = \hat{u}^\intercal \vec{x}$, so the mean of y is

$\mu_y = \mathbb{E}[y] = \mathbb{E}[\hat{u}^\intercal \vec{x}] = \hat{u}^\intercal \mathbb{E}[\vec{x}] = \hat{u}^\intercal \vec{\mu}.$

The variance of y is

$\sigma_y^2 = \mathbb{E}[(y - \mu_y)^2] = \mathbb{E}[(\hat{u}^\intercal \vec{x} - \hat{u}^\intercal \cdot \vec{\mu})^2] = \mathbb{E}[(\hat{u}^\intercal (\vec{x} - \vec{\mu}))^2] = \mathbb{E}[(\hat{u}^\intercal (\vec{x} - \vec{\mu}))((\vec{x} - \vec{\mu})^\intercal$
.

In [35]:
```
def math_marginal_meanvar(mu, C, u):
    return u @ mu, u @ C @ u
```

```
def sample_marginal_meanvar(X, u):
    Y = X @ u
    return np.mean(Y), np.var(Y)
```

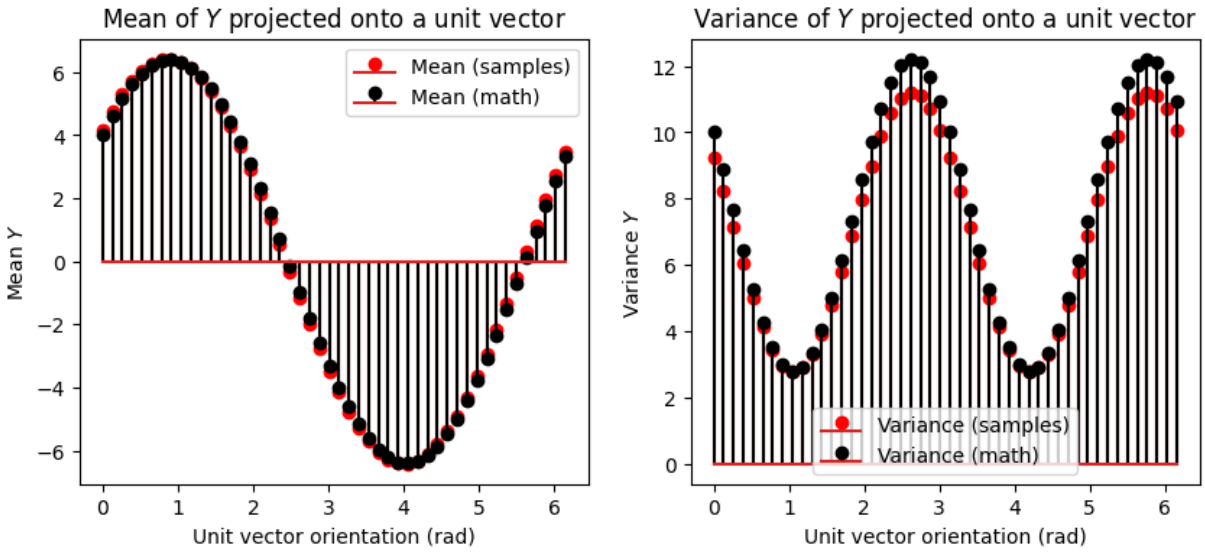
Check it for a set of 48 unit vectors spaced evenly around the unit circle. For each of these, compare the mean and variance predicted from your mathematical expression to the sample mean and variance estimated by projecting your 1,000 samples from part (a) onto $\hat{u}$.

In [44]:
```
t = 2 * np.pi * np.arange(48) / 48
vecs = np.array([np.cos(t), np.sin(t)]).T
```

In [49]:
```
math_means = []
math_vars = []
sample_means = []
sample_vars = []
for v in vecs:
    mean1, var1 = math_marginal_meanvar(mean, cov, v)
    mean2, var2 = sample_marginal_meanvar(Y, v)
    math_means.append(mean1)
    math_vars.append(var1)
    sample_means.append(mean2)
    sample_vars.append(var2)
```

Stem plot the mathematically computed mean and the sample mean (on the same plot), and also plot the mathematical variance and the sample variance, both plotted as a function of the orientation of $\hat{u}$ (relative to the x-axis).

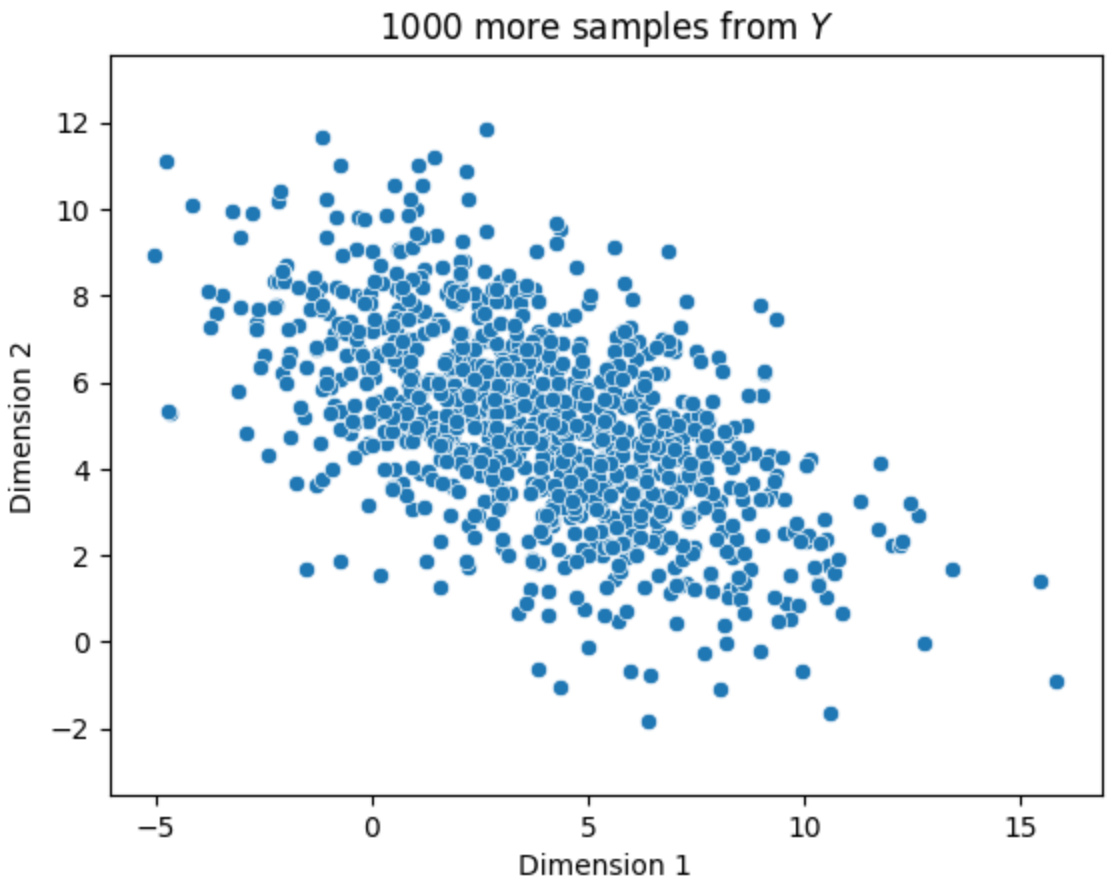
In [75]:
```
fig, axs = plt.subplots(1, 2, figsize=(10, 4))
plt.sca(axs[0])
plt.title('Mean of $Y$ projected onto a unit vector')
plt.stem(t, sample_means, 'r', label='Mean (samples)')
plt.stem(t, math_means, 'k', label='Mean (math)')
plt.legend()
plt.ylabel('Mean $Y$')
plt.xlabel('Unit vector orientation (rad)')

plt.sca(axs[1])
plt.title('Variance of $Y$ projected onto a unit vector')
plt.stem(t, sample_vars, 'r', label='Variance (samples)')
plt.stem(t, math_vars, 'k', label='Variance (math)')
plt.legend()
plt.ylabel('Variance $Y$')
plt.xlabel('Unit vector orientation (rad)')
plt.show()
```



(c)

Now scatterplot 1,000 new samples of a 2-dimensional Gaussian using the same μ and C_Y from part (a).

In [96]:
```
Y = nd_randn(mean, cov, num=1000)
sns.scatterplot(x=Y[:, 0], y=Y[:, 1])
plt.title('1000 more samples from $Y$')
plt.xlabel('Dimension 1')
plt.ylabel('Dimension 2')
plt.axis('equal')
plt.show()
```



Measure the sample mean and covariance of your data points, comparing to the values that you requested when calling the function.

In [97]:
```
sample_mean = np.mean(Y, axis=0)
sample_mean - mean
```

Out[97]:
```
array([-0.17877335,  0.06430002])
```

In [98]:
```
sample_cov = np.cov(Y.T)
sample_cov - cov
```

Out[98]:
```
array([[ 0.08904616,  0.06219317],
       [ 0.06219317, -0.15660276]])
```

For each of the unit vectors from (b), find the two points on the line through the sample mean in the direction of that unit vector for which the Mahalanobis distance from the mean (i.e., the negative of the exponent of the Gaussian density) is equal to one. **incomplete**

Plot a closed contour that connects all those points. Plot a second closed contour using the values of the mean and covariance you used to generate your sample. Try this on three additional random data sets with different means and covariance matrices. Does this contour capture the shape of the data? **incomplete**

How would you, mathematically, compute the direction (unit vector) that maximizes the variance of the marginal distribution?

Find the $\hat{u}$ that maximizes $\hat{u}^\intercal C \hat{u}$, using SVD and the optimization methods we learned in class.

How would you compute the direction that maximizes the distance corresponding to Mahalanobis distance equal to one?

Find the eigenvectors of the covariance matrix (principle component vectors). These will decompose the Gaussian into orthogonal directions of greatest spread. The Mahalanobis distance will be maximized in the direction of the first principle component (the one with the largest singular value).

Compute these directions and verify that they are consistent with your plot. **incomplete**

In [ ]: