chandlerrc_hw3

October 2, 2023

1 Homework 3

1.1 References

- Lectures 8-12 (inclusive).

1.2 Instructions

- Type your name and email in the "Student details" section below.
- Develop the code and generate the figures you need to solve the problems using this notebook.
- For the answers that require a mathematical proof or derivation you should type them using latex. If you have never written latex before and you find it exceedingly difficult, we will likely accept handwritten solutions.
- The total homework points are 100. Please note that the problems are not weighed equally.

[ ]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import matplotlib_inline
matplotlib_inline.backend_inline.set_matplotlib_formats('svg')
import seaborn as sns
sns.set_context("paper")
sns.set_style("ticks")

import scipy
import numpy as np
import scipy.stats as st
import urllib.request
import os

def download(
    url : str,
    local_filename : str = None
):
    """Download a file from a url.

    Arguments
    url            -- The url we want to download.
    local_filename -- The filemame to write on. If not
                      specified
```

```
    """
    if local_filename is None:
        local_filename = os.path.basename(url)
    urllib.request.urlretrieve(url, local_filename)
```

1.3 Student details

- **First Name: Robert**
- **Last Name: Chandler**
- **Email: chandl71@purdue.edu**

1.4 Problem 1 - Propagating uncertainty through a differential equation

This is a classic uncertainty propagation problem you must solve using Monte Carlo sampling. Consider the following stochastic harmonic oscillator:

$$\begin{aligned} \ddot{y} + 2\zeta\omega(X)\dot{y} + \omega^2(X)y &= 0, \\ y(0) &= y_0(X), \\ \dot{y}(0) &= v_0(X), \end{aligned}$$

where: $+\ X = (X_1, X_2, X_3)$, $+\ X_i \sim N(0,1)$, $+\ \omega(X) = 2\pi + X_1$, $+\ \zeta = 0.01$, $+\ y_0(X) = 1 + 0.1X_2$, and $+\ v_0 = 0.1X_3$.

In other words, this stochastic harmonic oscillator has an uncertain natural frequency and uncertain initial conditions.

Our goal is to propagate uncertainty through this dynamical system, i.e., estimate the mean and variance of its solution. A solver for this dynamical system is given below:

[ ]:
```
class Solver(object):
    def __init__(
        self,
        nt=100,
        T=5
    ):
        """This is the initializer of the class.

        Arguments:
            nt -- The number of timesteps.
            T  -- The final time.
        """
        self.nt = nt
        self.T = T
        # The timesteps on which we will get the solution
        self.t = np.linspace(0, T, nt)
        # The number of inputs the class accepts
        self.num_input = 3
        # The number of outputs the class returns
        self.num_output = nt
```

```
    def __call__(self, x):
        """This special class method emulates a function call.

        Arguments:
            x -- A 1D numpy array with 3 elements.
                 This represents the stochastic input x = (x1, x2, x3).

        Returns the solution to the differential equation evaluated
        at discrete timesteps.
        """
        # uncertain quantities
        x1 = x[0]
        x2 = x[1]
        x3 = x[2]

        # ODE parameters
        omega = 2*np.pi + x1
        y10 = 1 + 0.1*x2
        y20 = 0.1*x3
        # initial conditions
        y0 = np.array([y10, y20])

        zeta = 0.01
        # spring constant
        k = omega**2
        # damping coeff
        c = 2*zeta*omega
        # coefficient matrix
        C = np.array([[0, 1],[-k, -c]])

        #RHS of the ODE system
        def rhs(y, t):
            return np.dot(C, y)

        y = scipy.integrate.odeint(rhs, y0, self.t)

        return y
```

First, let's demonstrate how the solver works:

[ ]:
```
solver = Solver()

x = np.random.randn(solver.num_input)

y = solver(x)
```

```
print(y)
```

```
[[ 1.01540597e+00 -3.80155740e-02]
 [ 9.65606622e-01 -1.91640859e+00]
 [ 8.25076648e-01 -3.60255965e+00]
 [ 6.07623724e-01 -4.93868224e+00]
 [ 3.34222810e-01 -5.80096188e+00]
 [ 3.09944816e-02 -6.11101991e+00]
 [-2.73289349e-01 -5.84302363e+00]
 [-5.49934359e-01 -5.02579809e+00]
 [-7.73028661e-01 -3.73977219e+00]
 [-9.21868634e-01 -2.10907810e+00]
 [-9.82878780e-01 -2.89579046e-01]
 [-9.50847752e-01  1.54602074e+00]
 [-8.29368182e-01  3.22456217e+00]
 [-6.30443773e-01  4.58875955e+00]
 [-3.73305832e-01  5.51193172e+00]
 [-8.25556272e-02  5.90975714e+00]
 [ 2.14188056e-01  5.74796296e+00]
 [ 4.88911405e-01  5.04524806e+00]
 [ 7.15849104e-01  3.87119413e+00]
 [ 8.73899687e-01  2.33939228e+00]
 [ 9.48586247e-01  5.96462712e-01]
 [ 9.33380534e-01 -1.19197186e+00]
 [ 8.30269510e-01 -2.85701808e+00]
 [ 6.49515597e-01 -4.24245532e+00]
 [ 4.08638202e-01 -5.21938607e+00]
 [ 1.30717322e-01 -5.69822028e+00]
 [-1.57816690e-01 -5.63687984e+00]
 [-4.29695579e-01 -5.04447322e+00]
 [-6.59389196e-01 -3.98012156e+00]
 [-8.25501754e-01 -2.54707699e+00]
 [-9.12763601e-01 -8.82716296e-01]
 [-9.13433113e-01  8.54620978e-01]
 [-8.27980224e-01  2.50068980e+00]
 [-6.64991581e-01  3.90086101e+00]
 [-4.40311024e-01  4.92464048e+00]
 [-1.75501168e-01  5.47782354e+00]
 [ 1.04224741e-01  5.51115323e+00]
 [ 3.72402386e-01  5.02468743e+00]
 [ 6.03819136e-01  4.06748984e+00]
 [ 7.76883352e-01  2.73270263e+00]
 [ 8.75637796e-01  1.14849368e+00]
 [ 8.91229368e-01 -5.34242687e-01]
 [ 8.22700046e-01 -2.15625193e+00]
 [ 6.77028603e-01 -3.56498518e+00]
 [ 4.68423794e-01 -4.62894030e+00]
 [ 2.16940257e-01 -5.24993024e+00]
```

```
[-5.34482384e-02 -5.37213512e+00]
[-3.17133146e-01 -4.98710399e+00]
[-5.49295587e-01 -4.13426000e+00]
[-7.28241373e-01 -2.89688892e+00]
[-8.37427136e-01 -1.39401697e+00]
[-8.66988306e-01  2.31030992e-01]
[-8.14628027e-01  1.82429369e+00]
[-6.85787119e-01  3.23575432e+00]
[-4.93083486e-01  4.33345983e+00]
[-2.55078326e-01  5.01585024e+00]
[ 5.51041304e-03  5.22114662e+00]
[ 2.63975455e-01  4.93293063e+00]
[ 4.95961864e-01  4.18141326e+00]
[ 6.79760992e-01  3.04029944e+00]
[ 7.98340939e-01  1.61957148e+00]
[ 8.40923502e-01  5.48965940e-02]
[ 8.03961840e-01 -1.50532186e+00]
[ 6.91430363e-01 -2.91401427e+00]
[ 5.14403716e-01 -4.03930218e+00]
[ 2.89968968e-01 -4.77683832e+00]
[ 3.95776200e-02 -5.05947493e+00]
[-2.13003697e-01 -4.86336546e+00]
[-4.43948204e-01 -4.20994618e+00]
[-6.31615681e-01 -3.16363649e+00]
[-7.58579296e-01 -1.82550086e+00]
[-8.13242635e-01 -3.23494890e-01]
[-7.90897129e-01  1.19976383e+00]
[-6.94123568e-01  2.60053197e+00]
[-5.32503918e-01  3.74749945e+00]
[-3.21674837e-01  4.53409279e+00]
[-8.18161772e-02  4.88837061e+00]
[ 1.64279531e-01  4.77959298e+00]
[ 3.93372057e-01  4.22086601e+00]
[ 5.83967264e-01  3.26763617e+00]
[ 7.18332866e-01  2.01220234e+00]
[ 7.84147052e-01  5.74786887e-01]
[ 7.75626873e-01 -9.07970689e-01]
[ 6.94033209e-01 -2.29599720e+00]
[ 5.47508527e-01 -3.45901296e+00]
[ 3.50266859e-01 -4.28875424e+00]
[ 1.21216319e-01 -4.70904484e+00]
[-1.17852442e-01 -4.68278042e+00]
[-3.44338413e-01 -4.21518617e+00]
[-5.36965991e-01 -3.35306354e+00]
[-6.77782725e-01 -2.18012199e+00]
[-7.53831389e-01 -8.08859582e-01]
[-7.58340839e-01  6.30220518e-01]
[-6.91326318e-01  2.00102466e+00]
```

```
[-5.59546236e-01  3.17473402e+00]
[-3.75823487e-01  4.04190483e+00]
[-1.57799170e-01  4.52266745e+00]
[ 7.37602675e-02  4.57407441e+00]
[ 2.96940142e-01  4.19392217e+00]
[ 4.90750675e-01  3.42070804e+00]]
```

Notice the dimension of y:

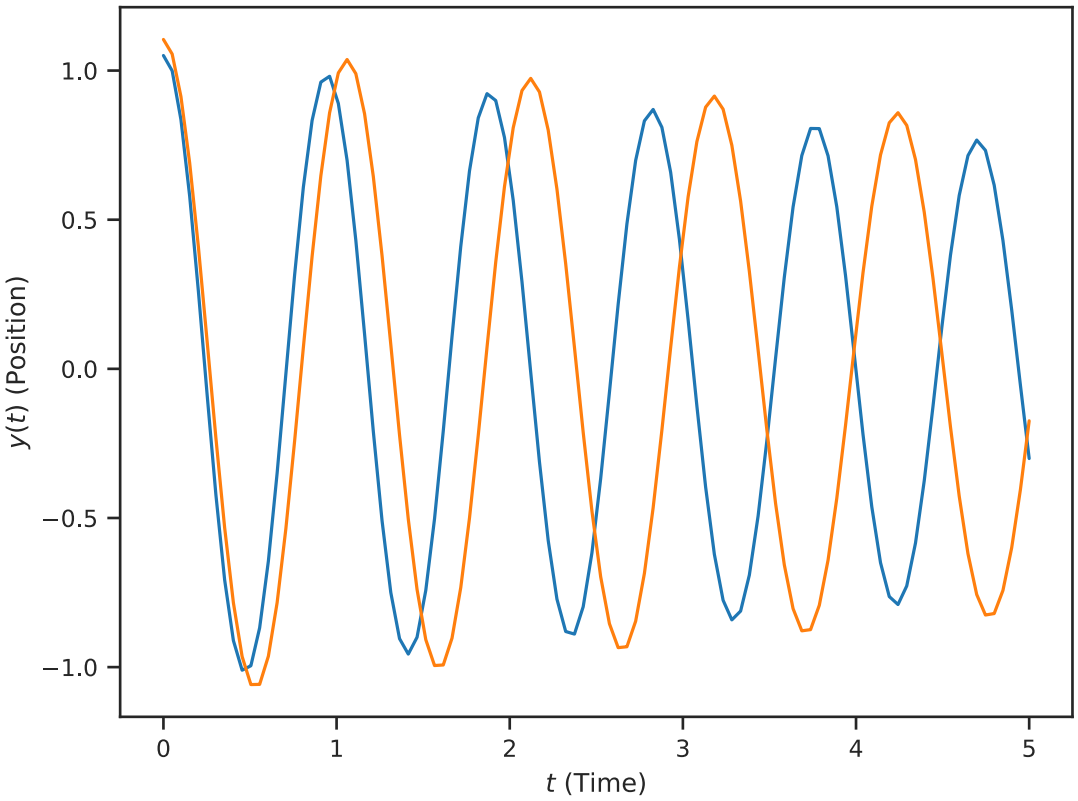
[ ]:
```
y.shape
```

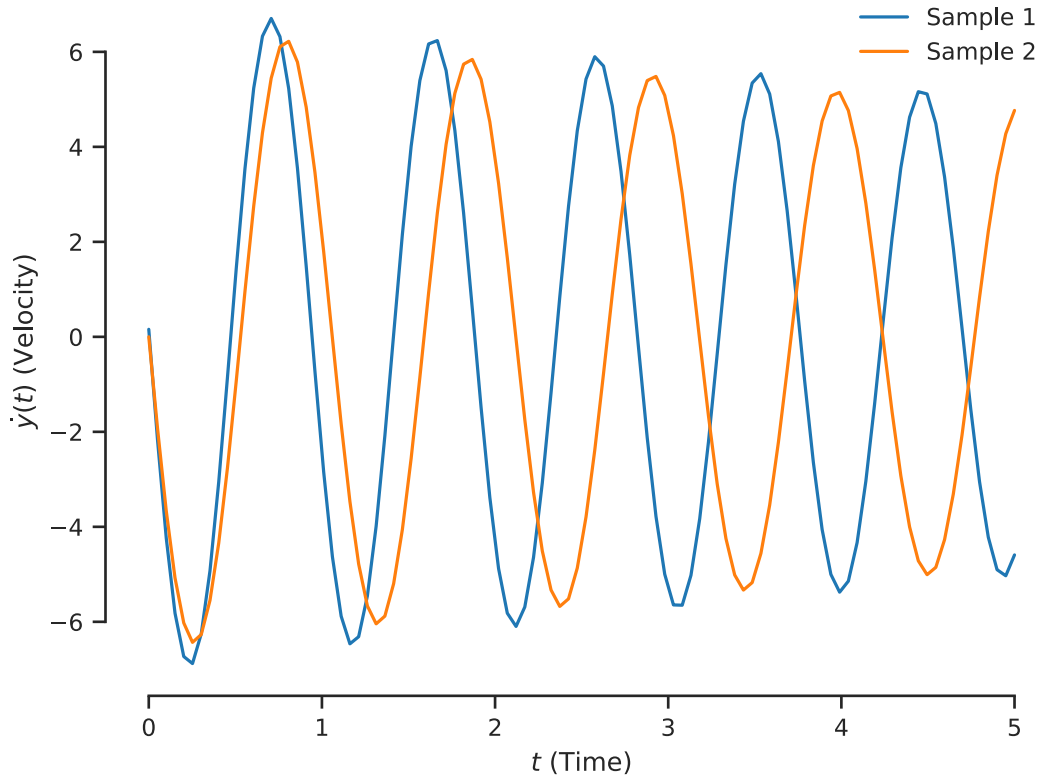
[ ]:
```
(100, 2)
```

The 100 rows corresponds to timesteps. The 2 columns correspond to position and velocity.

Let's plot a few samples:

[ ]:
```
fig1, ax1 = plt.subplots()
ax1.set_xlabel('$t$ (Time)')
ax1.set_ylabel('$y(t)$ (Position)')

fig2, ax2 = plt.subplots()
ax2.set_xlabel('$t$ (Time)')
ax2.set_ylabel('$\dot{y}(t)$ (Velocity)')

for i in range(2):
    x = np.random.randn(solver.num_input)
    y = solver(x)

    ax1.plot(solver.t, y[:, 0])
    ax2.plot(
        solver.t, y[:, 1],
        label=f'Sample {i+1:d}')
plt.legend(loc="best", frameon=False)
sns.despine(trim=True);
```


1.0
0.5
0.0
−0.5
−1.0
y(t) (Position)
0
1
2
3
4
5
t (Time)



For your convenience, here is code that takes many samples of the solver at once:

[ ]:
```
def take_samples_from_solver(num_samples):
    """Takes ``num_samples`` from the ODE solver.

    Returns them in an array of the form:
    ``num_samples x 100 x 2``
    (100 timesteps, 2 states (position, velocity))
    """
    samples = np.ndarray((num_samples, 100, 2))
    for i in range(num_samples):
        samples[i, :, :] = solver(
            np.random.randn(solver.num_input)
        )
    return samples
```

It works like this:

[ ]:
```
samples = take_samples_from_solver(50)
print(samples.shape)
```

```
(50, 100, 2)
```

Here, the first dimension corresponds to different samples. Then we have timesteps. And finally, we have either position or velocity.

As an example, the velocity of the 25th sample at the first ten timesteps is:

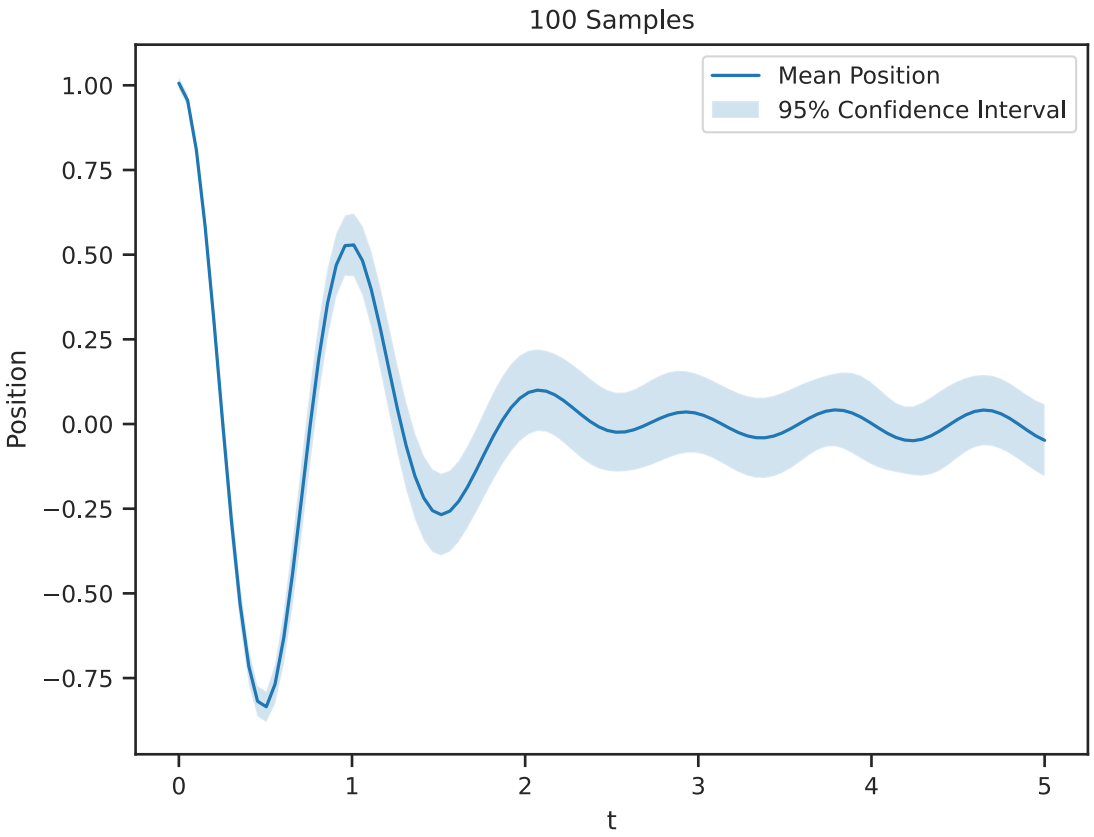
[ ]:
```
samples[24, :10, 1]
```

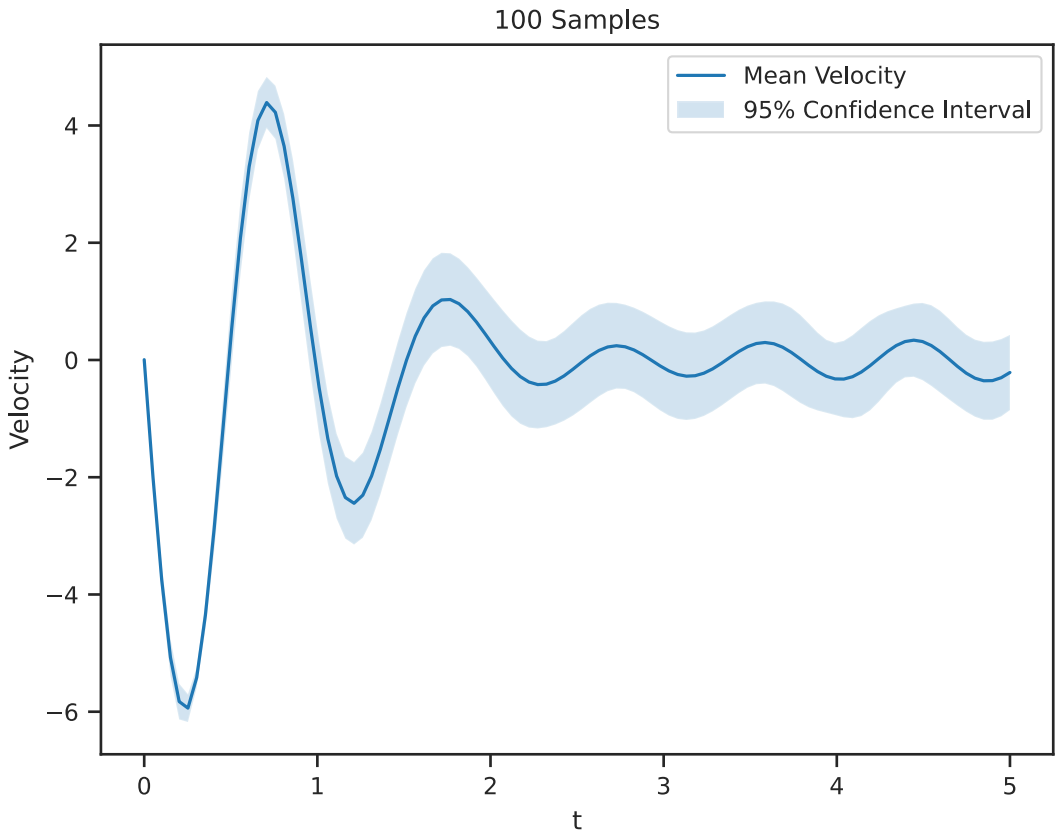
[ ]:
```
array([-0.15981544, -2.17836954, -3.95637924, -5.30974593, -6.09990503,
       -6.247999  , -5.74272521, -4.641078  , -3.06195493, -1.17334043])
```

1.4.1 Part A

Take 100 samples of the solver output and plot the estimated mean position and velocity as a function of time along with a 95% epistemic uncertainty interval around it. This interval captures how sure you are about the mean response when using only 100 Monte Carlo samples. You need to use the central limit theorem to find it (see the lecture notes).

[ ]:
```
N = 100
samples = take_samples_from_solver(N)
# Sampled positions are: samples[:, :, 0]
# Sampled velocities are: samples[:, :, 1]
# Sampled position at the 10th timestep is: samples[:, 9, 0]
# etc.

y_mean = samples.mean(axis=0)
y_var = samples.var(axis=0)

y_95_upper = y_mean + 2 * np.sqrt(y_var / N)
y_95_lower = y_mean - 2 * np.sqrt(y_var / N)

for i in [0, 1]:
    fig, ax = plt.subplots()
    measure = 'Position' if i == 0 else 'Velocity'
    ax.plot(solver.t, y_mean[:, i], label=f"Mean {measure}")
    ax.fill_between(solver.t, y_95_upper[:, i], y_95_lower[:, i], alpha=0.2,␣
 ↪label="95% Confidence Interval")
    ax.set_xlabel("t")
    ax.set_ylabel(measure)
    ax.set_title(f"{N} Samples")
    ax.legend();
```


100 Samples
Mean Position
95% Confidence Interval
Position
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
0
1
2
3
4
5
t



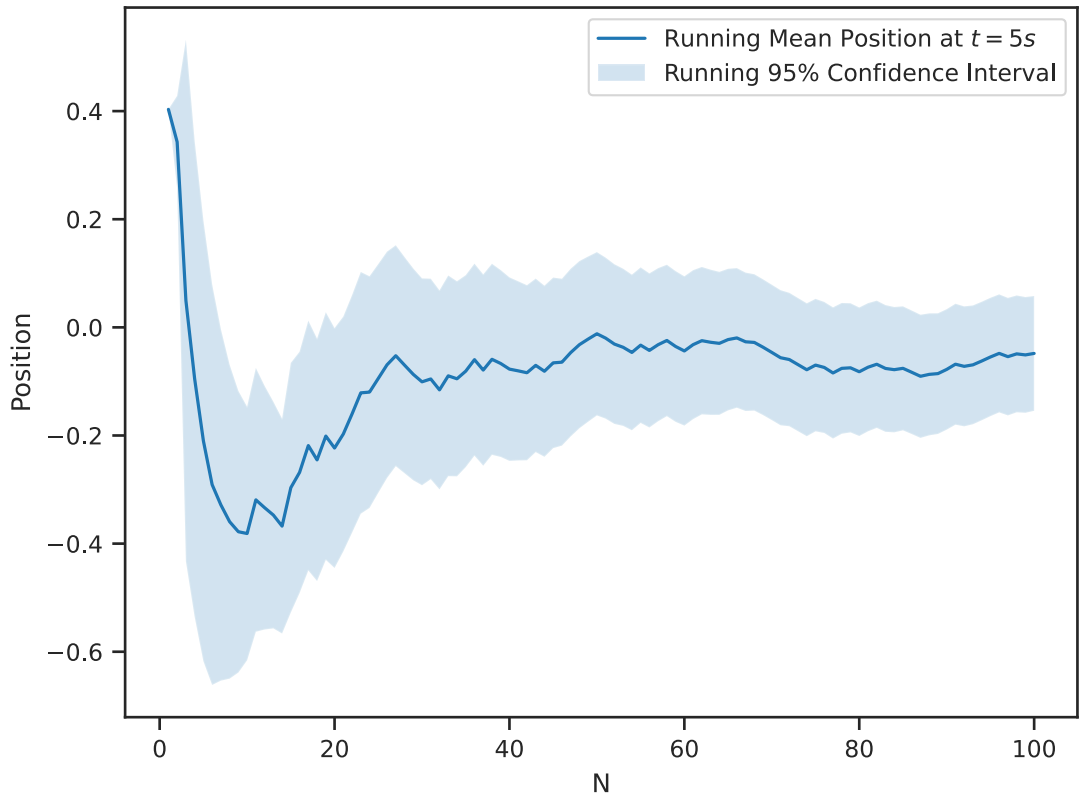
1.4.2 Part B

Plot the epistemic uncertainty about the mean position at $t = 5$ s as a function of the number of samples.

Solution:

[ ]:
```
pos_5s = samples[:, solver.t == 5, 0].squeeze()

Ns = np.arange(1, pos_5s.size + 1)

pos_mean_running = np.zeros_like(pos_5s)
pos_var_running = np.zeros_like(pos_5s)
for i, n in enumerate(Ns):
    pos_mean_running[i] = pos_5s[0:n].mean()
    pos_var_running[i] = pos_5s[0:n].var()

pos_95_upper = pos_mean_running + 2 * np.sqrt(pos_var_running / Ns)
pos_95_lower = pos_mean_running - 2 * np.sqrt(pos_var_running / Ns)

fig, ax = plt.subplots()
```

```
ax.plot(Ns, pos_mean_running, label="Running Mean Position at $t=5s$")
ax.fill_between(Ns, pos_95_upper, pos_95_lower, alpha=0.2, label="Running 95%␣
 ↪Confidence Interval")
ax.set_xlabel("N")
ax.set_ylabel("Position")
ax.legend();
```



1.4.3 Part C

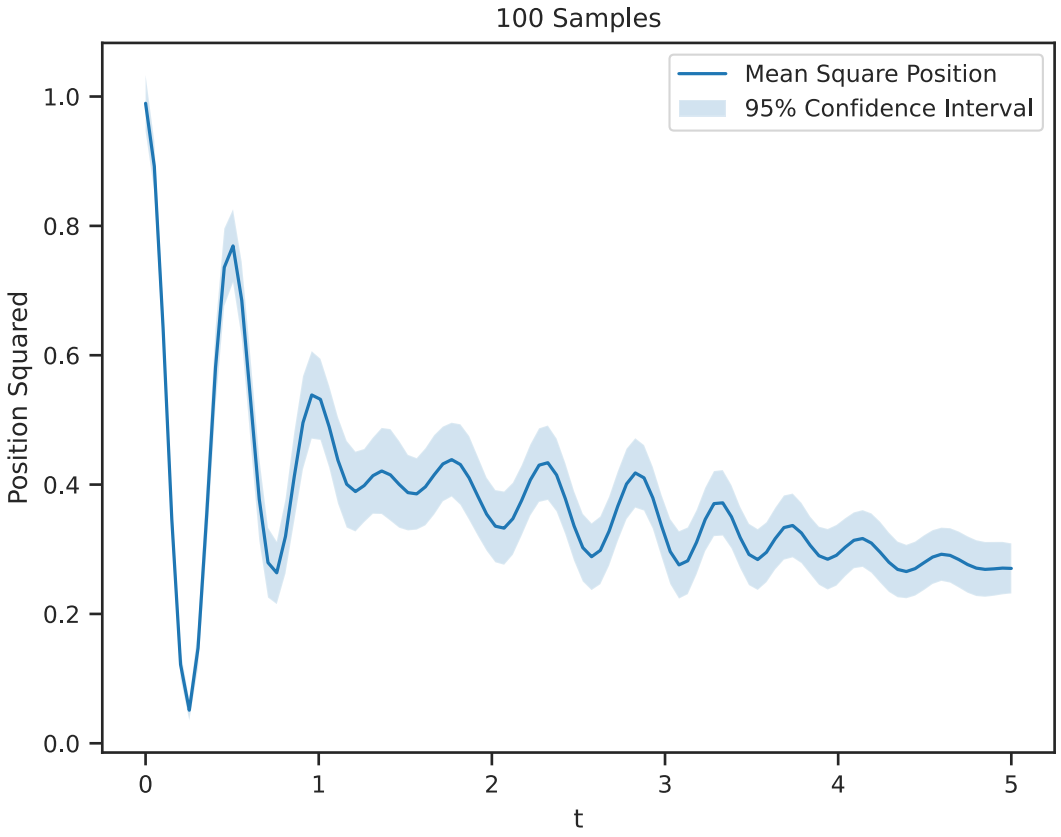
Repeat parts A and B for the squared response. That is, do the same thing as above, but consider $y^2(t)$ and $\dot{y}^2(t)$ instead of $y(t)$ and $\dot{y}(t)$. How many samples do you need to estimate the mean squared response at $t = 5$ s with negligible epistemic uncertainty?

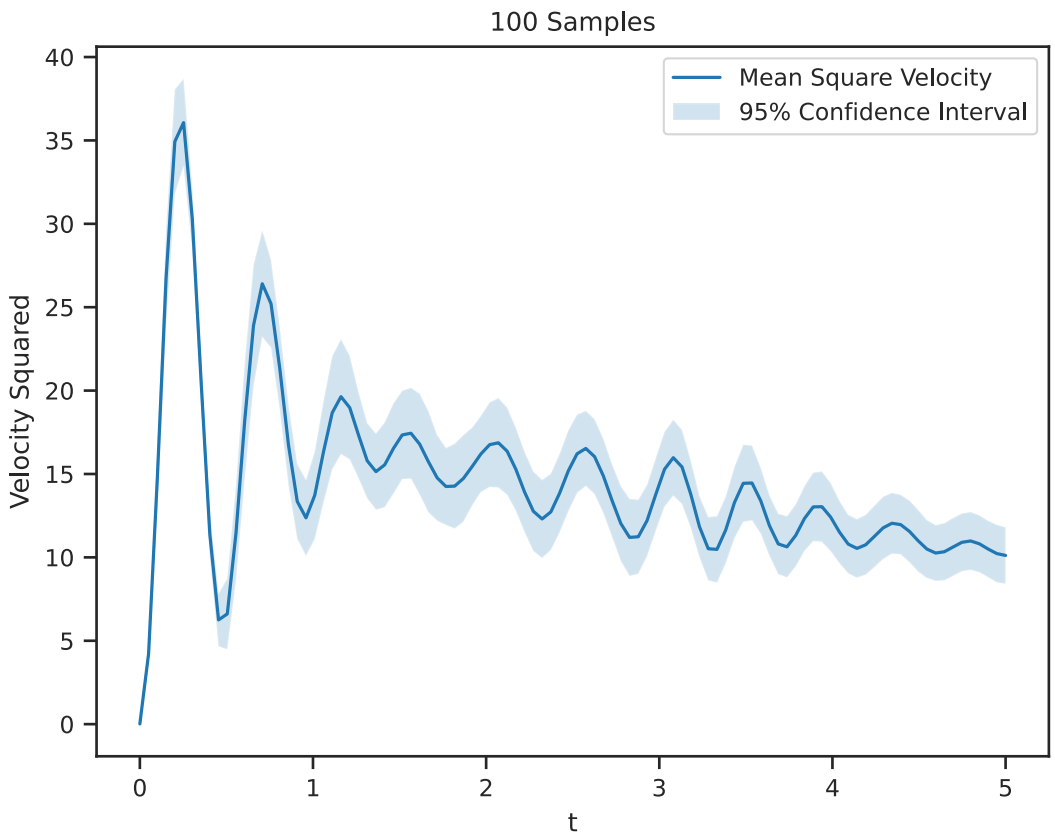
Solution:

Part A for Squared Response

[ ]:
```
samples = take_samples_from_solver(N) ** 2

y_mean = samples.mean(axis=0)
y_var = samples.var(axis=0)
```

```
y_95_upper = y_mean + 2 * np.sqrt(y_var / N)
y_95_lower = y_mean - 2 * np.sqrt(y_var / N)

for i in [0, 1]:
    fig, ax = plt.subplots()
    measure = 'Position' if i == 0 else 'Velocity'
    ax.plot(solver.t, y_mean[:, i], label=f"Mean Square {measure}")
    ax.fill_between(solver.t, y_95_upper[:, i], y_95_lower[:, i], alpha=0.2,␣
 ↪label="95% Confidence Interval")
    ax.set_xlabel("t")
    ax.set_ylabel(f"{measure} Squared")
    ax.set_title(f"{N} Samples")
    ax.legend();
```

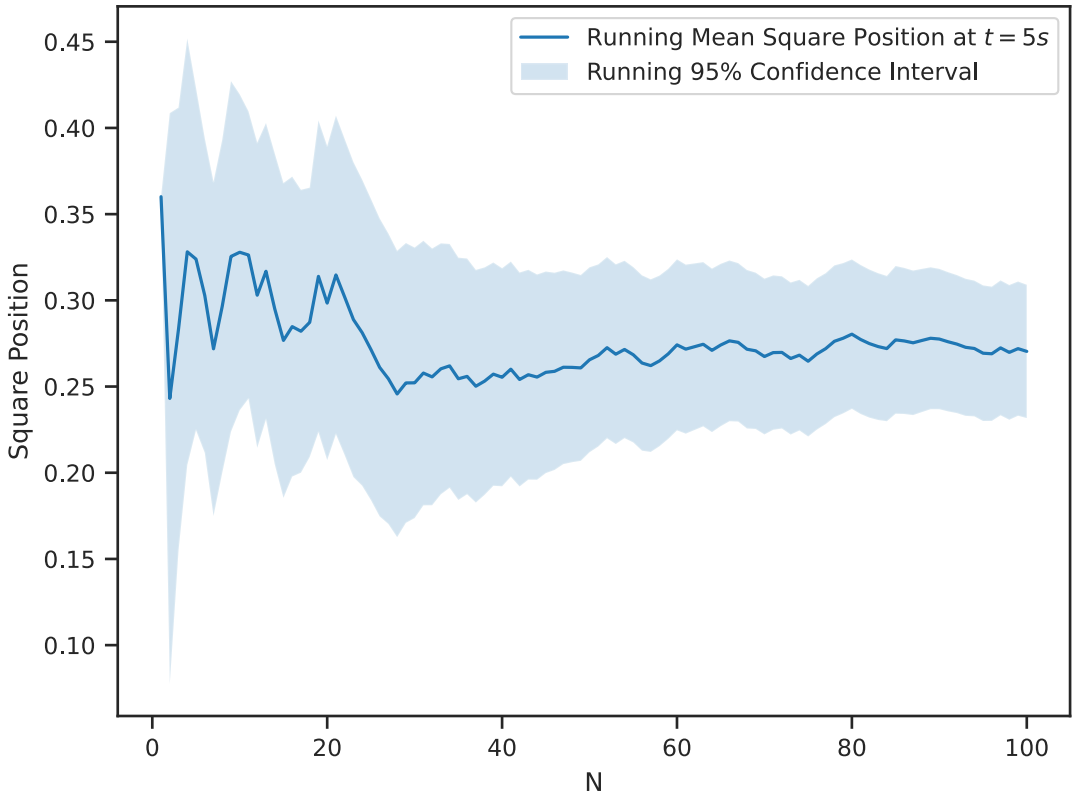




Part B for Squared Response

[ ]:
```
pos_5s = samples[:, solver.t == 5, 0].squeeze()

Ns = np.arange(1, pos_5s.size + 1)

pos_mean_running = np.zeros_like(pos_5s)
pos_var_running = np.zeros_like(pos_5s)
for i, n in enumerate(Ns):
    pos_mean_running[i] = pos_5s[0:n].mean()
    pos_var_running[i] = pos_5s[0:n].var()

pos_95_upper = pos_mean_running + 2 * np.sqrt(pos_var_running / Ns)
pos_95_lower = pos_mean_running - 2 * np.sqrt(pos_var_running / Ns)

fig, ax = plt.subplots()
ax.plot(Ns, pos_mean_running, label="Running Mean Square Position at $t=5s$")
ax.fill_between(Ns, pos_95_upper, pos_95_lower, alpha=0.2, label="Running 95%␣
 ↪Confidence Interval")
ax.set_xlabel("N")
```

```
ax.set_ylabel("Square Position")
ax.legend();
```



Defining the number of samples required to estimate the mean squared response at t = 5s with negligible epistemic uncertainty depends on how you define "negligible"… we could use the same definition used in lecture 10.4 and say that our 95% confidence interval being less wide than some value ϵ is a sufficient criterion. We must choose an appropriate ϵ, though. The value of 1 used in 10.4 is not appropriate for this problem: a quick visual inspection of the plot above shows that the confidence interval is never wider than 1. In 10.4, ϵ was chosen to be around 1% of the visible y-range of the running mean plot. We can choose ours similarly, using 1% of `ax.get_ylim`. We can use the same formula from 10.4 to estimate N:

$$N \geq \frac{16\bar{\sigma}^2}{\epsilon^2}$$

[ ]:
```
from IPython.display import Markdown as md

ylim = ax.get_ylim()
epsilon = 0.01 * (ylim[1] - ylim[0])

N_sufficient = int(np.ceil(16 * pos_var_running[-1] / epsilon**2))
```

```
md(
    f"Epsilon = {epsilon:0.5f}. By this criteron, $N={N_sufficient}$ is enough␣
 ↪samples to state that there is a 95% chance "
    f"that the true square position lies within the symmetric band {epsilon:0.
 ↪5f} units wide surrounding the "
    f"sample mean square position."
)
```

[ ]:

Epsilon = 0.00412. By this criteron, $N = 35305$ is enough samples to state that there is a 95% chance that the true square position lies within the symmetric band 0.00412 units wide surrounding the sample mean square position.

1.4.4 Part D

Now that you know how many samples you need to estimate the mean of the response and the square response, use the formula:

$$\mathbb{V}[y(t)] = \mathbb{E}[y^2(t)] - (\mathbb{E}[y(t)])^2,$$

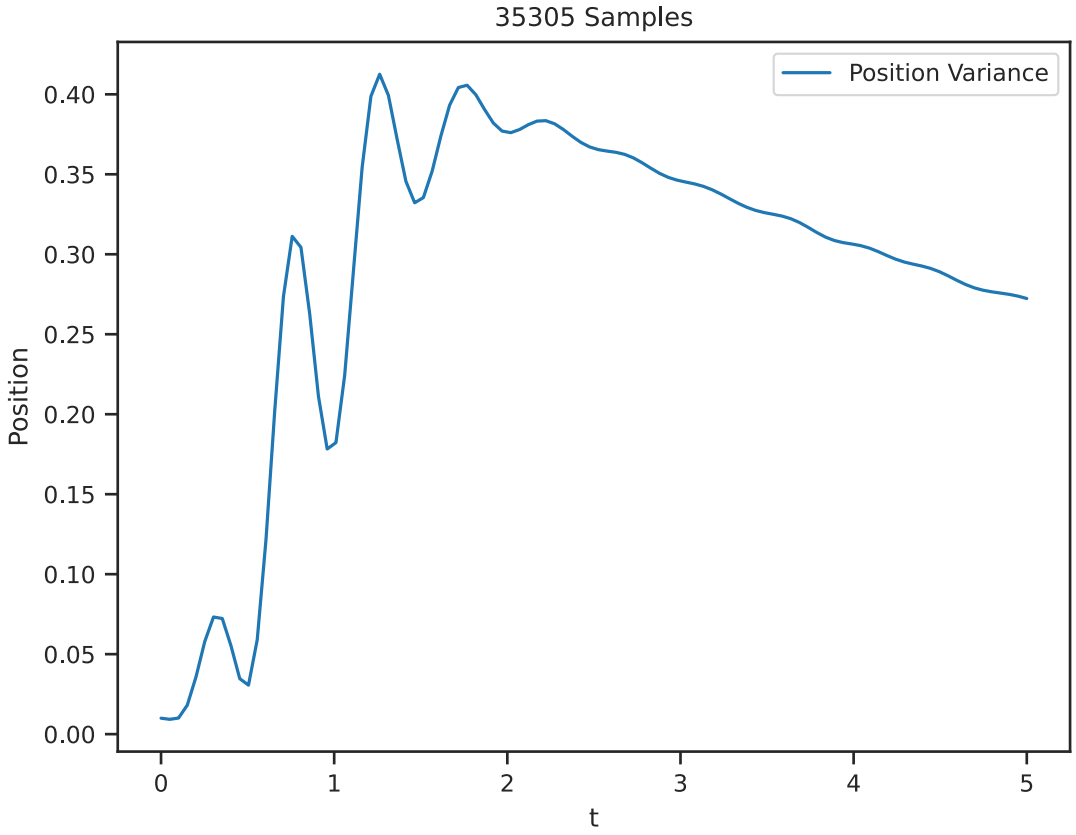
and similarly, for $\dot{y}(t)$, to estimate the position and velocity variance with negligible epistemic uncertainty. Plot both quantities as a function of time.

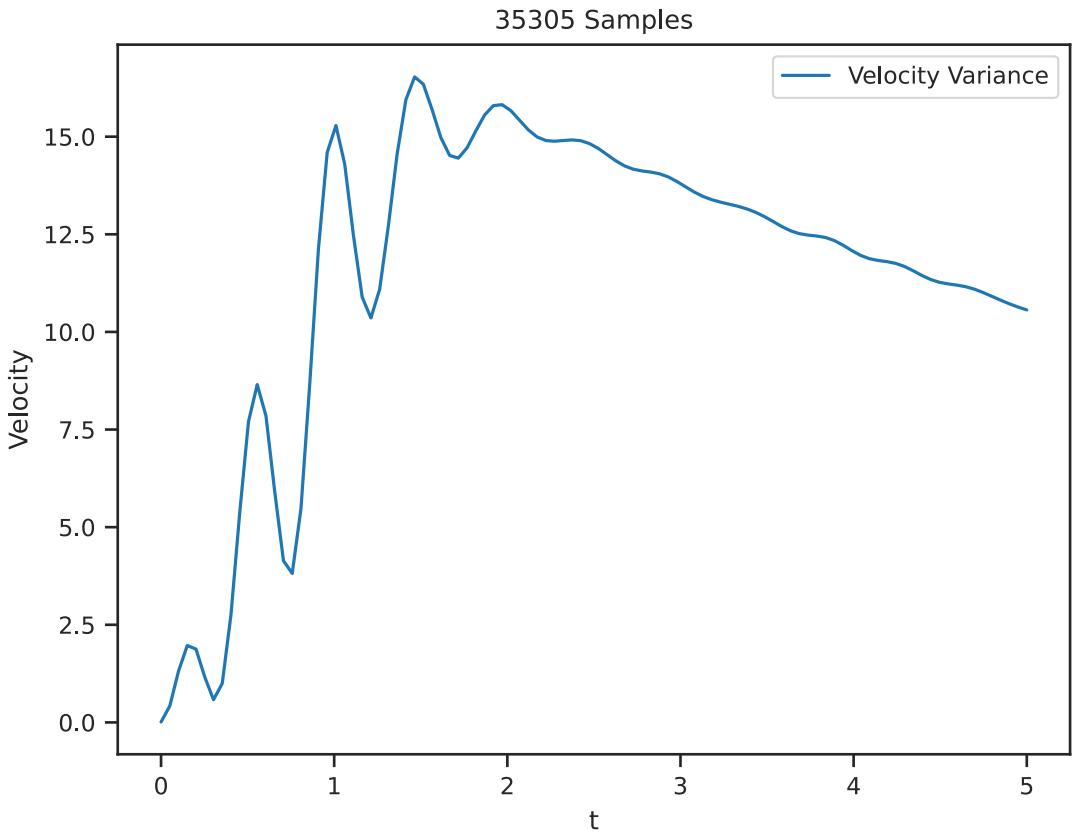
Solution:

[ ]:

```
N = N_sufficient

samples = take_samples_from_solver(N)

y_t = samples[:, :, 0]
y_dot_t = samples[:, :, 1]

mean_y = y_t.mean(axis=0)
mean_y_dot = y_dot_t.mean(axis=0)

var_y = (y_t ** 2).mean(axis=0) - mean_y ** 2
var_y_dot = (y_dot_t ** 2).mean(axis=0) - mean_y_dot ** 2

fig, ax = plt.subplots()
ax.plot(solver.t, var_y, label="Position Variance")
ax.set_xlabel("t")
ax.set_ylabel("Position")
ax.set_title(f"{N} Samples")
ax.legend()

fig, ax = plt.subplots()
ax.plot(solver.t, var_y_dot, label="Velocity Variance")
ax.set_xlabel("t")
```

```
ax.set_ylabel("Velocity")
ax.set_title(f"{N} Samples")
ax.legend();
```





1.4.5 Part E

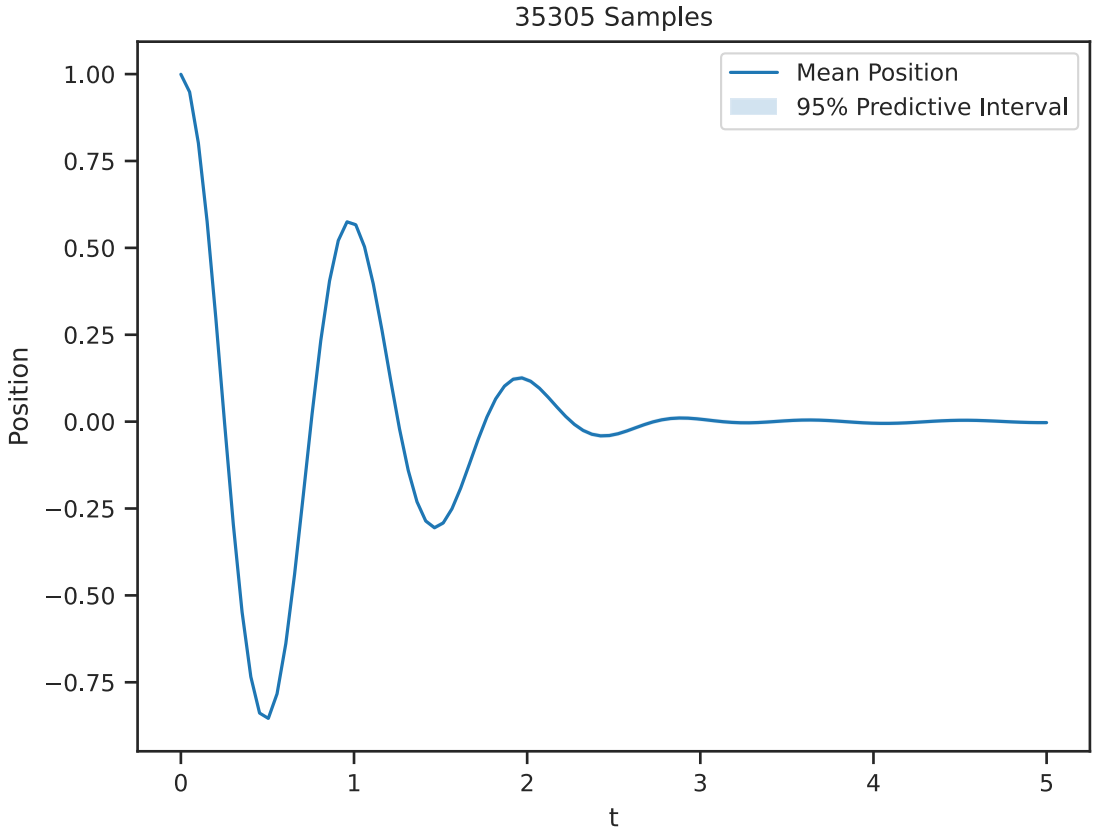
Put together the estimated mean and variance to plot a 95% predictive interval for the position and the velocity as functions of time.

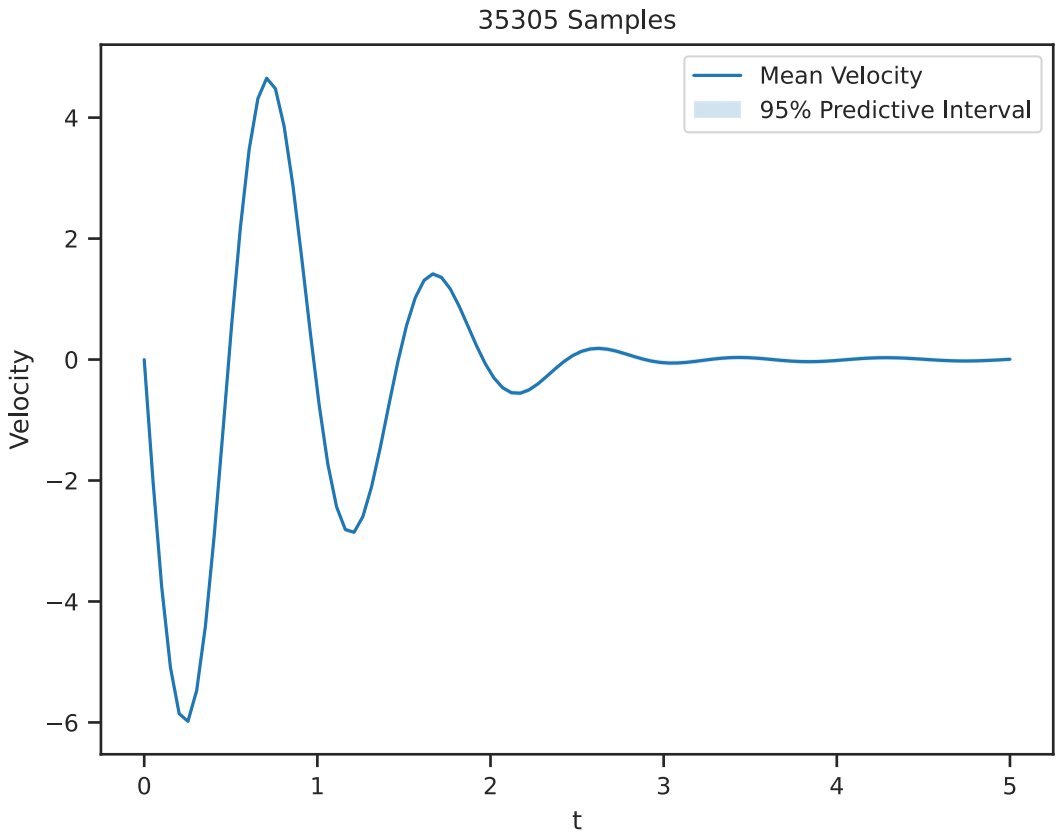
Hint: You need to use the Central Limit Theorem. Check out the corresponding textbook example.

Solution:

[ ]:
```
y_95_upper = mean_y + 2 * np.sqrt(var_y / N)
y_95_lower = mean_y - 2 * np.sqrt(var_y / N)

y_dot_95_upper = mean_y_dot + 2 * np.sqrt(var_y_dot / N)
y_dot_95_lower = mean_y_dot - 2 * np.sqrt(var_y_dot / N)

fig, ax = plt.subplots()
ax.plot(solver.t, mean_y, label="Mean Position")
ax.fill_between(solver.t, y_95_upper, y_95_lower, alpha=0.2, label="95%␣
 ↪Predictive Interval")
ax.set_title(f"{N} Samples")
ax.set_xlabel("t")
```

```
ax.set_ylabel("Position")
ax.legend()

fig, ax = plt.subplots()
ax.plot(solver.t, mean_y_dot, label="Mean Velocity")
ax.fill_between(solver.t, y_dot_95_upper, y_dot_95_lower, alpha=0.2, label="95%␣
 ↪Predictive Interval")
ax.set_title(f"{N} Samples")
ax.set_xlabel("t")
ax.set_ylabel("Velocity")
ax.legend();
```





The 95% interval is practically invisible on these plots because it is so narrow since our N is so high. This shows that our value for N from part C does indeed ensure negligible epistemic uncertainty.

1.5 Problem 2 - Earthquakes again

The San Andreas fault extends through California, forming the boundary between the Pacific and the North American tectonic plates. It has caused some of the most significant earthquakes on Earth. We are going to focus on Southern California, and we would like to assess the probability of a significant earthquake, defined as an earthquake of magnitude 6.5 or greater, during the next ten years.

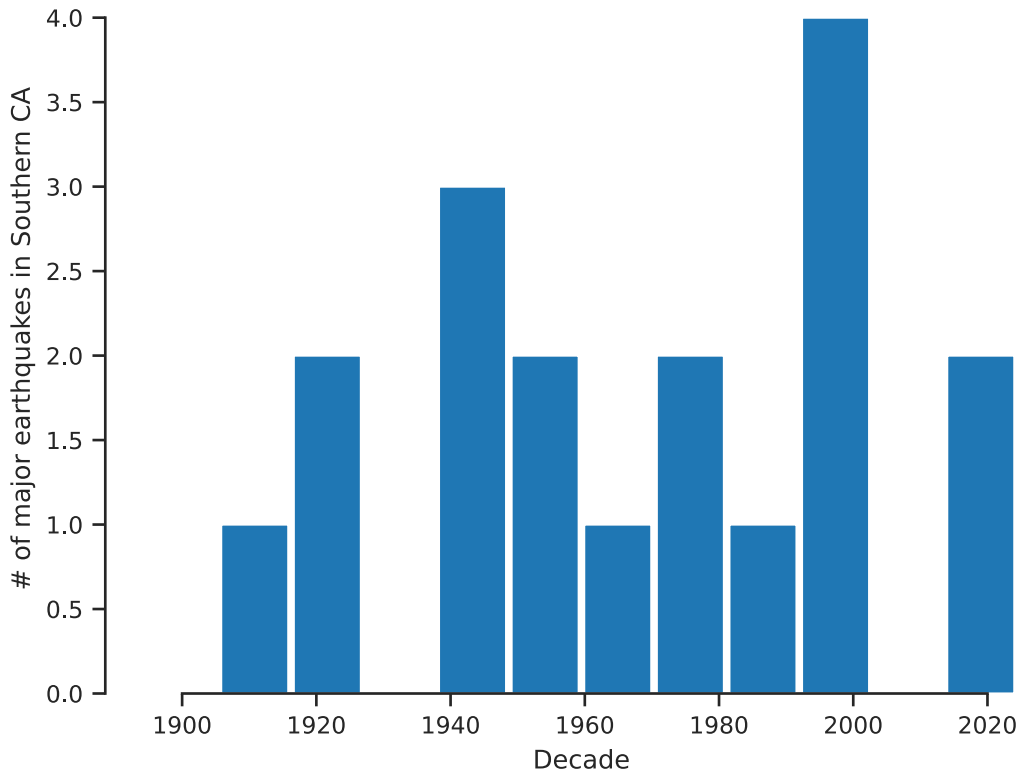
A. The first thing we will do is review a database of past earthquakes that have occurred in Southern California and collect the relevant data. We will start at 1900 because data before that time may be unreliable. Go over each decade and count the occurrence of a significant earthquake (i.e., count the number of orange and red colors in each decade). We have done this for you.

[ ]:
```
eq_data = np.array([
    0, # 1900-1909
    1, # 1910-1919
    2, # 1920-1929
    0, # 1930-1939
```

```
    3, # 1940-1949
    2, # 1950-1959
    1, # 1960-1969
    2, # 1970-1979
    1, # 1980-1989
    4, # 1990-1999
    0, # 2000-2009
    2  # 2010-2019
])
```

Let's visualize them:

[ ]:
```
fig, ax = plt.subplots()
ax.bar(
    np.linspace(1900, 2019, eq_data.shape[0]),
    eq_data,
    width=10
)
ax.set_xlabel('Decade')
ax.set_ylabel('# of major earthquakes in Southern CA')
sns.despine(trim=True);
```



A. The right way to model the number of earthquakes X_n in a decade n is using a Poisson distribution with unknown rate parameter λ, i.e.,

$$X_n|\lambda \sim \text{Poisson}(\lambda).$$

The probability mass function is:

$$p(x_n|\lambda) \equiv p(X_n = x_n|\lambda) = \frac{\lambda^{x_n}}{x_n!}e^{-\lambda}.$$

Here we have $N = 12$ observations, say $x_{1:N} = (x_1, \dots, x_N)$ (stored in `eq_data` above). Find the *joint probability mass function* (otherwise known as the likelihood) $p(x_{1:N}|\lambda)$ of these random variables.

Hint: Assume that all measurements are independent. Then, their joint PMF is the product of the individual PMFs. You should be able to simplify the expression.

Answer:

$$p(x_{1:N}|\lambda) = \prod_{n=1}^{N} p(x_n|\lambda) = \prod_{n=1}^{N} \frac{\lambda^{x_n}}{x_n!}e^{-\lambda} = \lambda^{\sum_{n=1}^{N} x_n} \frac{e^{-N\lambda}}{\prod_{n=1}^{N} x_n!}$$

B. The rate parameter λ (number of significant earthquakes per ten years) is positive. What prior distribution should we assign to it if we expect it to be around 2? A convenient choice is to pick a Gamma. See also the scipy.stats page for the Gamma because it results in an analytical posterior. We write:

$$\lambda \sim \text{Gamma}(\alpha, \beta),$$

where α and β are positive *hyper-parameters* that we must set to represent our prior state of knowledge. The PDF is:

$$p(\lambda) = \frac{\beta^\alpha \lambda^{\alpha-1} e^{-\beta\lambda}}{\Gamma(\alpha)},$$

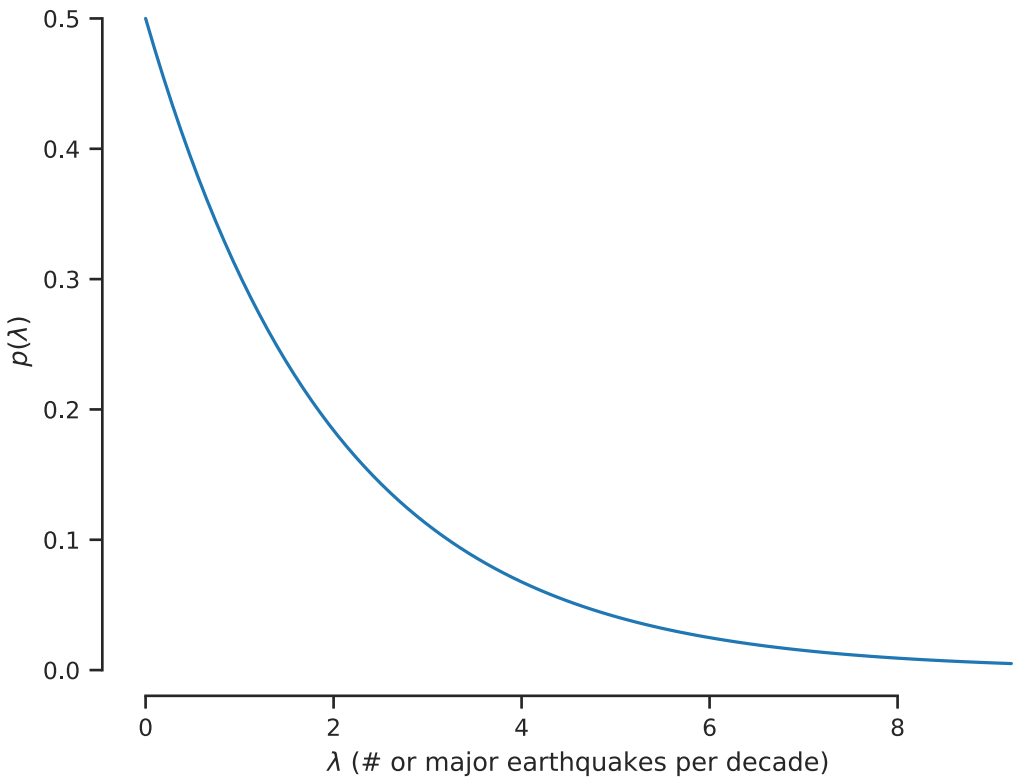
where we are not conditioning on α and β because they should be fixed numbers. Use the code below to pick reasonable values for α and β. **Just enter your choice of α and β in the code block below.**

Hint: Notice that the maximum entropy distribution for a positive parameter with known expectation is the Exponential, e.g., see the Table in this wiki page. Then, notice that the Exponential is a particular case of the Gamma (set $\alpha = 1$).

[ ]:
```
import scipy.stats as st

# We expect the rate parameter to be around 2, so this is its expected value
E_rate = 2.0
```

```
# Leave alpha as 1 since an Exponential gives the maximum entropy given our␣
 ↪prior knowledge
alpha = 1.0

# According to the definition of the Exponential distribution, our expectation␣
 ↪should equal 1/lambda,
# so lambda = 1/E_rate, and in this form, the beta of our Gamma distribution is␣
 ↪equal to the lambda
# of a typical Exponential distribution
beta = 1 / E_rate

# This is the prior on lambda:
lambda_prior = st.gamma(alpha, scale=1.0 / beta)

# Let's plot it:
lambdas = np.linspace(0, lambda_prior.ppf(0.99), 100)
fig, ax = plt.subplots()
ax.plot(lambdas, lambda_prior.pdf(lambdas))
ax.set_xlabel('$\lambda$ (# or major earthquakes per decade)')
ax.set_ylabel('$p(\lambda)$')
sns.despine(trim=True)

lambda_prior.expect()
```

[ ]: 2.0



C. Show that the posterior of λ conditioned on $x_{1:N}$ is also a Gamma, but with updated hyperparameters. **Hint:** When you write down the posterior of λ you can drop any multiplicative term that does not depend on it as it will be absorbed in the normalization constant. This will simplify the notation a little bit. **Answer:**

$$\text{posterior} = p(\lambda|x_{1:N}) \propto p(x_{1:N}|\lambda)p(\lambda) = \left(\lambda^{\sum_{n=1}^{N} x_n} \frac{e^{-N\lambda}}{\prod_{n=1}^{N} x_n!}\right)\left(\frac{\beta^{\alpha}\lambda^{\alpha-1}e^{-\beta\lambda}}{\Gamma(\alpha)}\right)$$

Dropping all terms that do not depend on λ:

$$p(\lambda|x_{1:N}) \propto \left(\lambda^{\sum_{n=1}^{N} x_n} e^{-N\lambda}\right)\left(\lambda^{\alpha-1}e^{-\beta\lambda}\right) = \left(\lambda^{\alpha-1+\sum_{n=1}^{N} x_n} e^{-(N+\beta)\lambda}\right)$$

And this final term is proportional to the PDF of another Gamma, so we can say that:

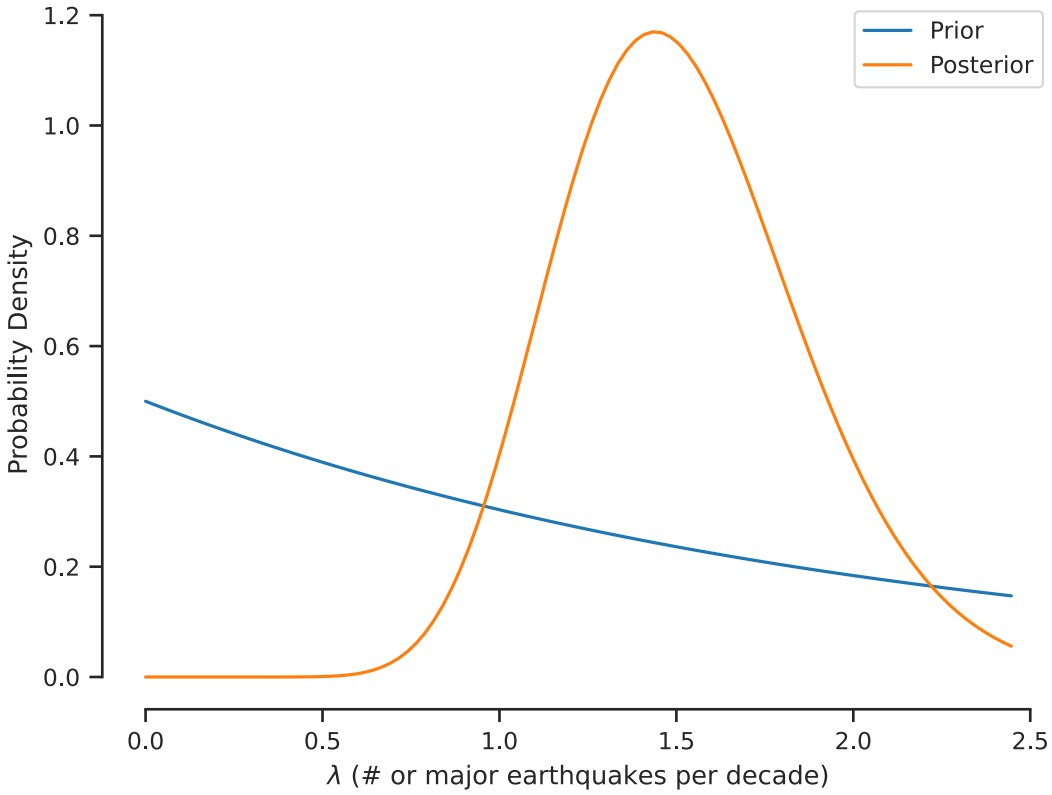
$$\lambda|x_{1:N} \sim \text{Gamma}\left(\alpha + \sum_{n=1}^{N} x_n, (N+\beta)\right)$$

D. Prior-likelihood pairs that result in a posterior with the same form as the prior are known as conjugate distributions. Conjugate distributions are your only hope for analytical Bayesian inference. As a verification check, look at the Wikipedia page for conjugate priors, locate the

Poisson-Gamma pair, and verify your answer above. *Nothing to report here. Just do it as a verification check.*

E. Plot the prior and the posterior of λ on the same plot.

[ ]:
```
# Your expression for alpha posterior here:
alpha_post = alpha + eq_data.sum()
# Your expression for beta posterior here:
beta_post = len(eq_data) + beta
# The posterior
lambda_post = st.gamma(alpha_post, scale=1.0 / beta_post)

# Plot it
lambdas = np.linspace(0, lambda_post.ppf(0.99), 100)
fig, ax = plt.subplots()
ax.plot(lambdas, lambda_prior.pdf(lambdas), label="Prior")
ax.plot(lambdas, lambda_post.pdf(lambdas), label="Posterior")
ax.set_xlabel('$\lambda$ (# or major earthquakes per decade)')
ax.set_ylabel('Probability Density')
ax.legend()
sns.despine(trim=True);
```



F. Let's determine the predictive distribution for the number of significant earthquakes during the next decade. This is something we did not do in class, but it will reappear in future lectures. Let X be the random variable corresponding to the number of significant earthquakes during the next decade. We need to calculate:

$$p(x|x_{1:N}) = \text{our state of knowledge about } X \text{ after seeing the data.}$$

How do we do this? We use the sum rule:

$$p(x|x_{1:N}) = \int_0^\infty p(x|\lambda, x_{1:N})p(\lambda|x_{1:N})d\lambda = \int_0^\infty p(x|\lambda)p(\lambda|x_{1:N})d\lambda,$$

where going from the middle step to the rightmost one, we assumed that the number of earthquakes occurring in each decade is independent. You can carry out this integration analytically (it gives a negative Binomial distribution), but we are not going to bother with it.

Below, you will write code to characterize it using Monte Carlo sampling. You can take a sample from the posterior predictive by:

- sampling a λ from its posterior $p(\lambda|x_{1:N})$.
- sampling an x from the likelihood $p(x|\lambda)$.

This is the same procedure we used for replicated experiments.

Complete the code below:

[ ]:
```
def sample_posterior_predictive(n, lambda_post):
    """Sample from the posterior predictive.

    Arguments
    n           -- The number of samples to take.
    lambda_post -- The posterior for lambda.

    Returns n samples from the posterior
    """
    samples = np.empty((n,), dtype="i")
    for i in range(n):
        lambda_sample = lambda_post.rvs()
        samples[i] = st.poisson.rvs(mu=lambda_sample)
    return samples
```

Test your code here:

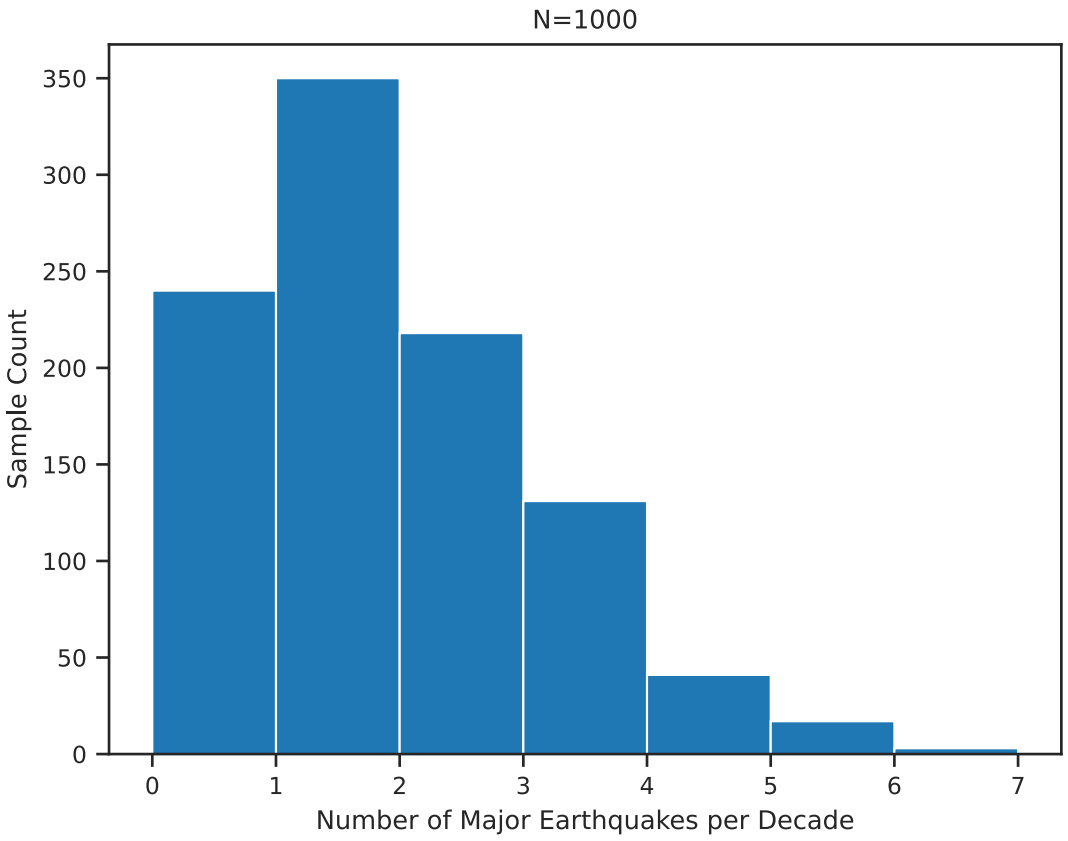
[ ]:
```
samples = sample_posterior_predictive(10, lambda_post)
samples
```

[ ]:
```
array([1, 3, 1, 3, 2, 1, 1, 2, 1, 4], dtype=int32)
```

G. Plot the predictive distribution $p(x|x_{1:N})$.

Hint: Draw 1,000 samples using `sample_posterior_predictive` and then draw a histogram.

[ ]:
```
N = 1000
samples = sample_posterior_predictive(N, lambda_post)
fig, ax = plt.subplots()
ax.hist(samples, bins=np.arange(8));
ax.set_title(f'N={N}')
ax.set_xlabel("Number of Major Earthquakes per Decade")
ax.set_ylabel("Sample Count");
```



H. What is the probability that at least one major earthquake will occur during the next decade?

Hint: You may use a Monte Carlo estimate of the probability. Ignore the uncertainty in the estimate.

[ ]:
```
num_samples = 10000
samples = sample_posterior_predictive(num_samples, lambda_post)

# Count how many of the samples had earthquakes
count = 0
for i in range(num_samples):
    if samples[i] >=1:
        count += 1
```

```
p_at_least_1 = count / num_samples

print(f"p(X >= 1 | data) = {p_at_least_1}")
```

```
p(X >= 1 | data) = 0.7591
```

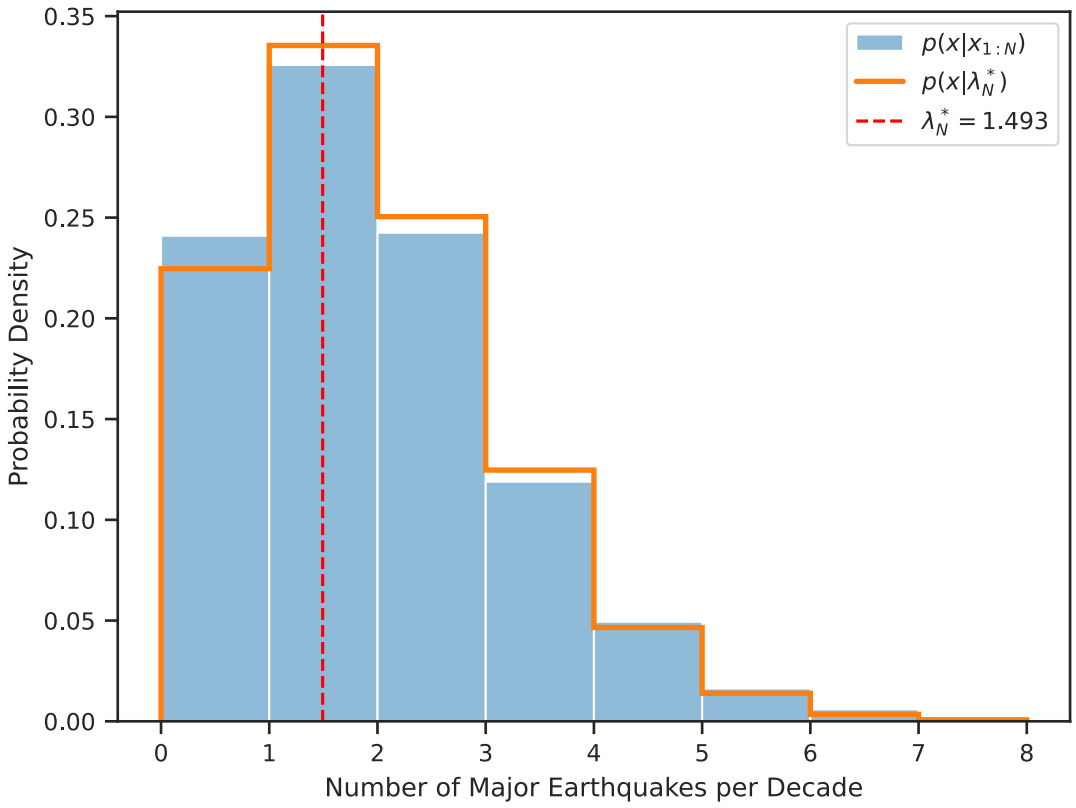
I. Find a 95% credible interval for λ.

[ ]:
```
lambda_95_lower = lambda_post.ppf(0.025)
lambda_95_upper = lambda_post.ppf(0.975)

print(f"A 95% credible interval for lambda is [{lambda_95_lower:.4f},␣
 ↪{lambda_95_upper:.4f}]")
```

```
A 95% credible interval for lambda is [0.9151, 2.2758]
```

J. Find the λ that minimizes the absolute loss (see lecture), and call it λ_N^*. Then, plot the fully Bayesian predictive $p(x|x_{1:N})$ in the same figure as $p(x|\lambda_N^*)$.

[ ]:
```
# Median minimizes absolute loss
lambda_star_N = lambda_post.median()

fig, ax = plt.subplots()
x_points = np.linspace(0, 8, 1000)
ax.hist(samples, density=True, bins=np.arange(8), alpha=0.5, label="$p(x|x_{1:
 ↪N})$")
# ax.hist(x=np.arange(8), bins=np.arange(8), weights=st.poisson.pmf(np.
 ↪arange(8), mu=lambda_star_N), label=r"$p(x|\lambda^*_N)$", alpha=0.5)
ax.stairs(
    values=st.poisson.pmf(np.arange(8), mu=lambda_star_N),
    linewidth=2,
    label=r"$p(x|\lambda^*_N)$",
)

fig.draw_without_rendering()
ax.autoscale(enable=False)
ax.vlines(
    lambda_post.median(),
    ymin=ax.get_ylim()[0] - 5,
    ymax=ax.get_ylim()[1] + 5,
    linestyles="--",
    colors="red",
    label=rf"$\lambda^*_N={lambda_star_N:.3f}$",
)

ax.set_xlabel("Number of Major Earthquakes per Decade")
ax.set_ylabel("Probability Density")
ax.legend();
```



L. Draw replicated data from the model and compare them to the observed data.

Hint: Complete the missing code at the places indicated below.

[ ]:
```
def replicate_experiment(post_rv, n=len(eq_data), n_rep=9):
    """Replicate the experiment.

    Arguments
    post_rv -- The random variable object corresponding to
               the posterior from which to sample.
    n       -- The number of observations.
    nrep    -- The number of repetitions.

    Returns:
    A numpy array of size n_rep x n.
    """
    x_rep = np.empty((n_rep, n), dtype="i")
    for i in range(n_rep):
        x_rep[i, :] = st.poisson.rvs(size=n, mu=post_rv.rvs())
    return x_rep
```

Try your code here:

[ ]:
```
n_rep = 9
x_rep = replicate_experiment(lambda_post, n_rep=n_rep)
x_rep
```

[ ]:
```
array([[0, 1, 3, 3, 3, 0, 2, 0, 3, 2, 2, 0],
       [0, 1, 1, 1, 2, 3, 2, 5, 2, 4, 3, 2],
       [1, 2, 0, 2, 2, 1, 3, 3, 2, 2, 3, 1],
       [2, 2, 1, 1, 2, 2, 1, 2, 0, 0, 0, 2],
       [2, 1, 2, 1, 2, 4, 3, 2, 0, 1, 3, 0],
       [0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0],
       [2, 3, 1, 2, 1, 2, 1, 2, 2, 3, 1, 1],
       [2, 1, 2, 1, 0, 1, 0, 0, 1, 1, 1, 1],
       [6, 1, 2, 0, 0, 3, 0, 2, 2, 1, 0, 0]], dtype=int32)
```

If it works, then try the following visualization:

[ ]:
```
fig, ax = plt.subplots(
    5,
    2,
    sharex='all',
    sharey='all',
    figsize=(20, 20)
)
ax[0, 0].bar(
    np.linspace(1900, 2019, eq_data.shape[0]),
    eq_data,
    width=10,
    color='red'
)
for i in range(1, n_rep + 1):
    ax[int(i / 2), i % 2].bar(
        np.linspace(1900, 2019, eq_data.shape[0]),
        x_rep[i-1],
        width=10
    )
```

M. Plot the histograms and calculate the Bayesian p-values of the following test quantities:

- Maximum number of consecutive decades with no earthquakes.
- Maximum number of successive decades with earthquakes.

Hint: You may reuse the code from Posterior Predictive Checking.

[ ]:
```
def perform_diagnostics(post_rv, data, test_func, n_rep=1000):
    """Calculate Bayesian p-values.

    Arguments
    post_rv    -- The random variable object corresponding to
                  the posterior from which to sample.
    data       -- The training data.
```

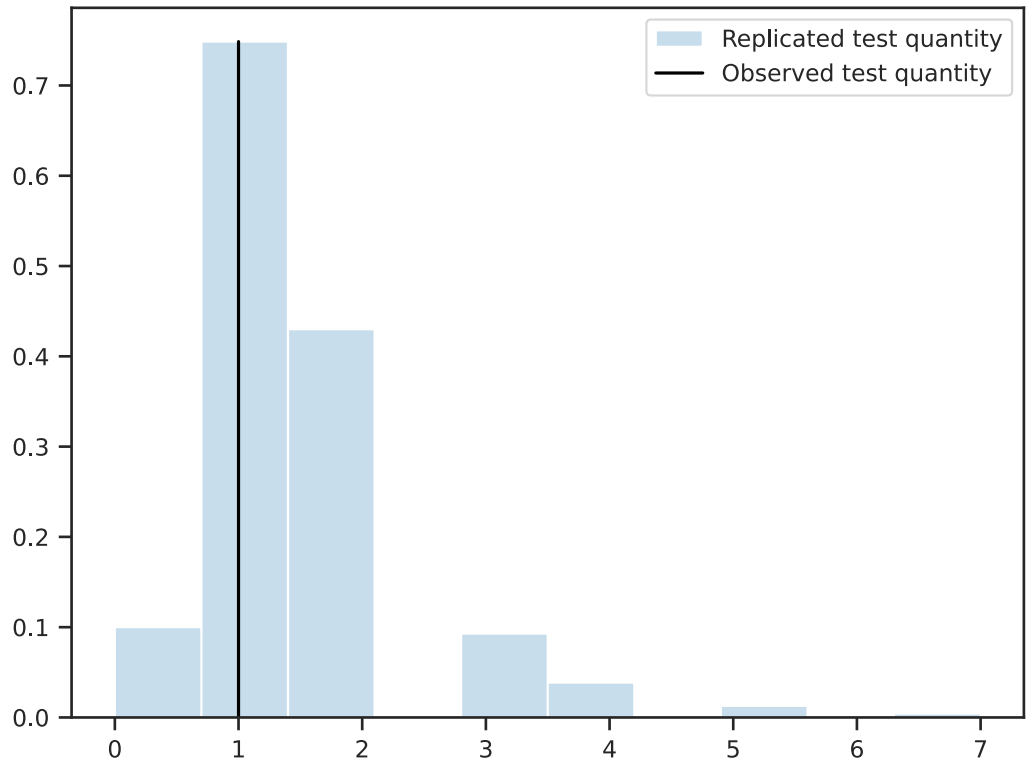
```
    test_func -- The test function.
    n         -- The number of observations.
    nrep      -- The number of repetitions.

    Returns a dictionary that includes the observed value of
    the test function (T_obs), the Bayesian p-value (p_val),
    the replicated test statistic (T_rep),
    and all the replicated data (data_rep).
    """
    T_obs = test_func(data)
    n = data.shape[0]
    data_rep = replicate_experiment(post_rv, n_rep=n_rep)
    T_rep = np.array(
        tuple(
            test_func(x)
            for x in data_rep
        )
    )
    p_val = (
        np.sum(np.ones((n_rep,))[T_rep > T_obs]) / n_rep
    )
    return dict(
        T_obs=T_obs,
        p_val=p_val,
        T_rep=T_rep,
        data_rep=data_rep
    )


def plot_diagnostics(diagnostics):
    """Make the diagnostics plot.

    Arguments:
    diagnostics -- The dictionary returned by perform_diagnostics()
    """
    fig, ax = plt.subplots()
    tmp = ax.hist(
        diagnostics["T_rep"],
        density=True,
        alpha=0.25,
        label='Replicated test quantity'
    )[0]
    ax.plot(
        diagnostics["T_obs"] * np.ones((50,)),
        np.linspace(0, tmp.max(), 50),
        'k',
        label='Observed test quantity'
```

```
    )
    plt.legend(loc='best');


def do_diagnostics(post_rv, data, test_func, n_rep=1000):
    """Calculate Bayesian p-values and make the corresponding
    diagnostic plot.

    Arguments
    post_rv   -- The random variable object corresponding to
                 the posterior from which to sample.
    data      -- The training data.
    test_func -- The test function.
    n         -- The number of observations.
    nrep      -- The number of repetitions.

    Returns a dictionary that includes the observed value of
    the test function (T_obs), the Bayesian p-value (p_val),
    and the replicated experiment (data_rep).
    """
    res = perform_diagnostics(
        post_rv,
        data,
        test_func,
        n_rep=n_rep
    )

    T_obs = res["T_obs"]
    p_val = res["p_val"]

    print(f'The observed test quantity is {T_obs}')
    print(f'The Bayesian p_value is {p_val:.4f}')

    plot_diagnostics(res)
```

[ ]:
```
# Here is the first test function for you
def T_eq_max_neq(x):
    """Return the maximum number of consecutive decades
    with no earthquakes."""
    count = 0
    result = 0
    for i in range(x.shape[0]):
        if x[i] != 0:
            count = 0
        else:
            count += 1
            result = max(result, count)
```

```
    return result

do_diagnostics(lambda_post, eq_data, T_eq_max_neq)
```

```
The observed test quantity is 1
The Bayesian p_value is 0.4060
```



[ ]:
```
def T_max_consecutive_eq_decades(x):
    """Return the maximum number of consecutive decades
    with earthquakes."""
    count = 0
    result = 0
    for i in range(x.shape[0]):
        if x[i] == 0:
            count = 0
        else:
            count += 1
            result = max(result, count)
    return result

do_diagnostics(lambda_post, eq_data, T_max_consecutive_eq_decades)
```

```
The observed test quantity is 6
```

```
The Bayesian p_value is 0.3610
```