Homework 3

References

- Lectures 7-10 (inclusive).

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 can either:
    - Type the answer using the built-in latex capabilities. In this case, simply export the notebook as a pdf and upload it on gradescope; or
    - You can print the notebook (after you are done with all the code), write your answers by hand, scan, turn your response to a single pdf, and upload on gradescope.

Note:

- Please match all the pages corresponding to each of the questions when you submit on gradescope.

Important: If you are running the notebook on Google Colab make sure you make a copy on your Google Drive so that you can resume your work later. To do this click on "File->Save a copy in Drive." Rename the copy as you wish by clicking on the filename you see on the very top. Then make sure you save regularly. If you close your browser, you can resume your work by going to your Google Drive and clicking on the notebook. You can find the notebooks the folder "My Drive/Colab Notebooks."

Student details

- **First Name:** Alex
- **Last Name:** Shank
- **Email:** shank14@purdue.edu

In [1]:
```python
# Here are some modules that you may need - please run this block of code:
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
sns.set_context('talk')
import numpy as np
import scipy
import scipy.stats as st
```

Problem 0

This is not a real problem. You just have to follow the instruction to make sure that you have access to the required data files for completing the subsequent problems. There are two data files needed for this purpose:

- hw_03_p1B_data.txt which is needed for Problem 1/Part B.
- hw_03_p2_data.txt which is needed for Problem 2.

Our goal is to make sure that this Jupyter notebook can see these two files. There are three cases that we are going to consider. The cases depend on how you accessed the notebook and where you are currently running it. You only need to choose and carry out the instructions for the case that is relevant to you:

Instructions for Google Colab

Google Colab gives you a computational session with temporary storage. The data must be visible from there. The best way to do this is to put the data in your Google drive and access the drive from this session. Here is how:

- First, right click on the two files above and then "Save As." Make sure that the selected format is "text." Remember where you save the files in your local computer.
- Second, go to your Google Drive. We need to select a folder to put the files. To keep things simple, let's just dump everything in "My Drive/Colab Notebooks," i.e., the same folder that contains the copy of this Jupyter notebook which you should have already made (if not, please see the **important** section at the very top. Once you have entered this folder in your Google Drive (just double click on it), drag and drop the two files in there. This is how your Google Drive should look like:

Google Drive screenshot

- Now we need to make the Google Drive visible from this computational session. We do this by *mounting* the drive. You need to run this code and follow the instructions:

In [2]:
```python
# The following code does not run unless you are running the notebook on Google Colab
# from google.colab import drive
# drive.mount('/content/drive')
```

- Finally, change directories so that the data files are in the current working directory:

In [3]:
```python
# # Only if you are running on Google Colab...
# # Print working directory before changing
# !pwd
# # Change to the desired directory on Google drive
# %cd "/content/drive/My Drive/Colab Notebooks"
# # Print working directory after changing
# !pwd
# # List the contents of the directory - Make sure it does contain the files we want:
# !ls
```

- Move to the section called **Loading the data**.

Instructions for personal computers

If you have just downloaded the notebook and you are running it locally

- Locate the folder in which you have the notebook.
- Right click on the two files above and then "Save As." Make sure that the selected format is "text" and that you save them in the same folder that contains this notebook.
- Move to the section called **Loading the data**.

If you have cloned the entire github repository of the class to your local computer

There is nothing to do. You already have the files in the same folder as this notebook. If the code below that loads the data does not work, then you are probably not running this notebook from the copy in the github repository.

- Move to the section called **Loading the data**.

Loading the data

If you have done everything correctly up to this point you must have the files in the right directory and accessible from this notebook. Let's test this:

In [4]:
```python
# If this fails you have a problem with the first file
data_p1B = np.loadtxt('hw_03_p1B_data.txt')
# print(data_p1B)
```

In [5]:
```python
# If this fails you have a problem with the second file
data_p2 = np.loadtxt('hw_03_p2_data.txt')
# print(data_p1B)
```

Problem 1

Part A

Let X be a continuous random variable with CDF:

$$F(x) = p(X \le x).$$

Show that the random variable

$$Z = F(X)$$

is distributed uniformly in $[0,1]$. Hint: Show that:

$$F_Z(z) := p(Z \le z) = z.$$

Proof:

If we know the random variable X has CDF F, then we can show that X is equivalent to $F^{-1}(U)$, where $U \sim U([0,1])$.

Sub-Proof:

$$F(x) = P(X \le x) = P(F^{-1}(U) \le x) = P(F(F^{-1}(U)) \le F(x))$$
$$= P(U \le F(x)) = F(x) \text{ because } P(U \le k) = k$$

Now, we can conclude that $Z = F(X) = F(F^{-1}(U)) = U$
By definition, $F_u(u) = P(U \le u) = u$, so

$$F_z(z) = P(Z \le z) = z$$

Part B

Note (Before you attempt this part of the problem): To do this problem your Jupyter notebook needs to be able to see this data file: hw_03_p1B_data.txt. For this to happen the file must be in the same folder as the Jupyter notebook notebook. There are two cases:

- If you are using Google Colab:
    - In this Jupyter notebook do "File->Save A Copy in Drive"
    - Write click on hw_03_p1B_data.txt and click "Save Link As." Make sure that you save the link as text and that you make a note where it is.
    - Open your Google Drive in a separate tab. Go to the folder "Colab Notebooks" which was automatically created when you saved a copy of the notebook. Drag and drop the hw_03_p1B_data.txt file you just downloaded in that folder.
- If you are running the Jupyter notebook on your own computer:
    - If you cloned the github repository, then the file will be there.
    - If you downloaded the notebook manually, then download the file as per the instructions for Google Colab and put it in the same folder as the Jupyter notebook.

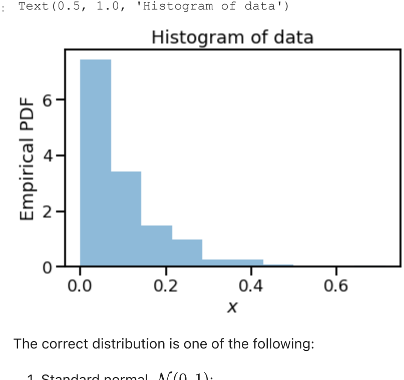
The theorem you have just proved is very useful when you want to test if a set of observations $x_1, x_2, \dots, x_N$ has been indeed independent realizations of a random variable X with CDF $F(x)$. Part A proves that, if this hypothesis is valid, then the transformed dataset $z_1, z_2, \dots, z_N$, where

$$z_i = F(x_i),$$

should be distributed uniformly in $[0,1]$. In other words, the empirical histrogram of $z_1, z_2, \dots, z_N$ should be a flat line. Use this observation to find the distribution from which this dataset was sampled:

In [6]:
```python
# If the following fails, please read the intructions above
data = np.loadtxt('hw_03_p1B_data.txt')
fig, ax = plt.subplots()
ax.hist(data, density=True, alpha=0.5)
ax.set_xlabel('$x$')
ax.set_ylabel('Empirical PDF')
ax.set_title('Histogram of data')
```

Out[6]: Text(0.5, 1.0, 'Histogram of data')



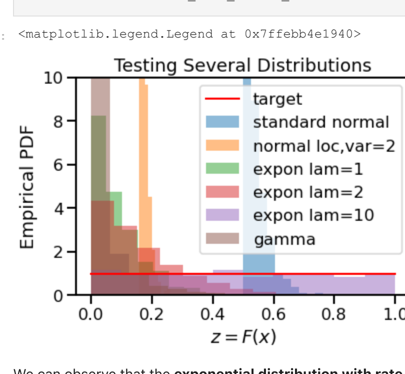
The correct distribution is one of the following:

1. Standard normal, $\mathcal{N}(0,1)$;
2. Normal with mean 2 and variance 2, $\mathcal{N}(2,2)$; or
3. Exponential with rate parameter 1, $\mathcal{E}(1)$; or
4. Exponential with rate parameter 2, $\mathcal{E}(2)$; or
5. Exponential with rate parameter 10, $\mathcal{E}(10)$; or
6. Gamma distribution with parameters $\alpha = 2.$ and $\beta = 3.$.

Systematically go over these distributions and try to determine which one generated the data. All the required CDF's and inverse CDF's are implemented in scipy.stats. Check also scipy.stats.rv_continuous.) Please pay special attention to the defintiion of the probability distributions of the various random variables and how you can control their parameters. As a hint, here is how you can test for $\mathcal{N}(0,1)$:

In [7]:
```python
# test all the suggested distributions
transformed_data_list = [
    st.norm(loc=0, scale=1).cdf(data),
    st.norm(loc=2, scale=2).cdf(data),
    st.expon(scale=1).cdf(data),
    st.expon(scale=1.0 / 2).cdf(data),
    st.expon(scale=1.0 / 10).cdf(data),
    st.gamma(a=2.0, scale=1.0 / 3.0).cdf(data)
]

# legend labels
transformed_data_legend_list = [
    "target",
    "standard normal",
    "normal loc,var=2",
    "expon lam=1",
    "expon lam=2",
    "expon lam=10",
    "gamma"
]

# plot everything
fig, ax = plt.subplots()
ax.plot(np.linspace(0, 1,50), np.ones(50), lw=2, color='r')
for transformed_data in transformed_data_list:
    ax.hist(transformed_data, density=True, alpha=0.5)

# zoom in a bit, set labels
ax.set_ylim([0, 10])
ax.set_xlabel('$z = F(x)$')
ax.set_ylabel('Empirical PDF')
ax.set_title('Testing Several Distributions');
ax.legend(transformed_data_legend_list)
```

Out[7]: <matplotlib.legend.Legend at 0x7ffebb4e1940>



We can observe that the **exponential distribution with rate parameter of 10** fits the uniform distribution closest. If there were two distributions that were both viable options, we could compute errors between the two histograms and the standard uniform. That is not necessary in this case, though.

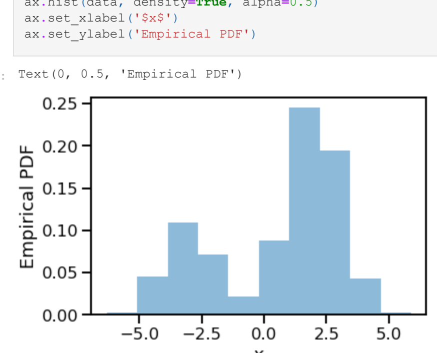
Problem 2

Note: This problem also requires a data file. This file: hw_03_p2_data.txt. Please follow the same directions we gave in Part B of Problem 1 above.

Consider the following data set $x_1, \dots, x_N$:

In [8]:
```python
# If the following fails, please look at the note above.
data = np.loadtxt('hw_03_p2_data.txt')
data = np.array(data)
fig, ax = plt.subplots()
ax.hist(data, density=True, alpha=0.5)
ax.set_xlabel('$x$')
ax.set_ylabel('Empirical PDF')
```

Out[8]: Text(0, 0.5, 'Empirical PDF')



Your goal is to generate a procedure that samples from the same distribution as the observed data. This is a variation of the standard *density estimation* problem. In general, this is a very difficult problem and we will see various ways to solve it later on. In this problem, you will develop a simple method that relies on the empirical CDF of the observed data. Needless to say, this method works only for one dimensional cases in which you have a lot of observations.

The empirical CDF of our data set, $x_1, \dots, x_N$ is defined to be:

$$\hat{F}_N(x) = \frac{\text{Number of observations} \le x}{N} = \frac{1}{N}\sum_{i=1}^{N} 1_{[x_i, +\infty]}(x),$$

where $1_A(x)$ is the indicator function of the set A. Using the, so called, strong law of large numbers, we can show that $\hat{F}_N(x)$ converges to the true CDF of the data as $N \to +\infty$.

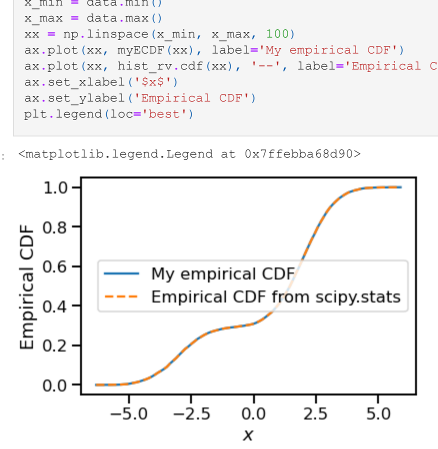
Part A

Complete the code that calculates the empirical CDF:

In [9]:
```python
def myECDF_base(x):
    """
    Make this code work if ``x`` is a simple scalar.

    :param x:    The point at which you want to observe the PDF.
    :returns:    The value of the empirical CDF at ``x``.
    """
    N = data.shape[0]
    return np.count_nonzero(data < x) / N

# Vectorize your function (i.e., make it work with 1D numpy arrays).
# See this: https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.vectori
myECDF = np.vectorize(myECDF_base)
```

In [10]:
```python
# You can test your results by comparing the empirical CDF you can get with of scipy.
# The two should match almost exactly
hist_rv = st.rv_histogram(np.histogram(data, bins=1000))
fig, ax = plt.subplots()
# The range in which the x's takes values:
x_min = data.min()
x_max = data.max()
xx = np.linspace(x_min, x_max, 100)
ax.plot(xx, myECDF(xx), label='My empirical CDF')
ax.plot(xx, hist_rv.cdf(xx), '--', label='Empirical CDF from scipy.stats')
ax.set_xlabel('$x$')
ax.set_ylabel('Empirical CDF')
plt.legend(loc='best')
```

Out[10]: <matplotlib.legend.Legend at 0x7ffebba68d90>



Now use the *inverse transform sampling* method to generate samples from same distribution as the original data. That is, you can now generate uniform samples:

$$u_i \sim U([0,1]),$$

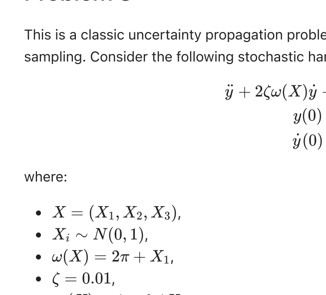
and transform them as:

$$\hat{x}_i = \hat{F}^{-1}(u_i).$$

The $\hat{x}_i$'s generated in this way should have the same distribution of the data you started with. Verify this by comparing the histrogram of $1,000$ $\hat{x}_i$ samples with the original data of this problem.

In [13]:
```python
fig, ax = plt.subplots()
U = st.uniform()
xs = U.rvs(1000)
ys = myiECDF(xs)
ax.hist(ys, density=True, alpha=0.5)
ax.hist(data, density=True, alpha=0.5)
ax.set_xlabel('$x$')
ax.set_ylabel('Empirical PDF')
ax.legend(["Inverse Transform Samples", "Orig. Data"])
```

Out[13]: <matplotlib.legend.Legend at 0x7ffebbac1640>



We can observe that our inverse transform samples have a similar distribution to our original data.

Problem 3

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

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

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

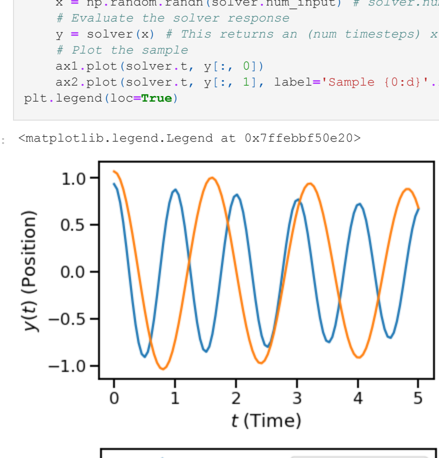
Our goal is to propagate uncertainty through this dynamical system, i.e., estimate the mean and variance of its solution. A solver for this dynamical system is given below:

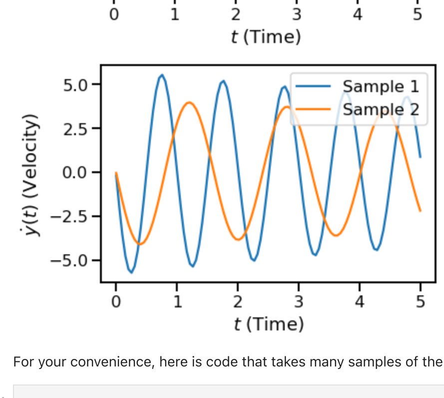
In [14]:
```python
class Solver(object):
    def __init__(self, nt=100, T=5):
        """
        This is the initializer of the class.

        Arguments:
            nt - The number of timesteps.
            T  - The final time.
        """
        self.nt = nt
        self.T = T
        self.t = np.linspace(0, T, nt) # The timesteps on which we will get the soluti
        # The following are not essential, but they are convenient
        self.num_input = 3            # The number of inputs the class accepts
        self.num_output = nt          # The number of outputs the class returns

    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
        """
        ##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
        y0 = np.array([y10, y20])    #initial conditions

        #coefficient matrix
        zeta = 0.01
        k = omega**2    ##spring constant
        c = 2*zeta*omega    ##damping coeff.
        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
```

Let's plot a few samples of the forward model to demonstrate how the solver works.

In [15]:
```python
# 1. Create the solver object
solver = Solver()
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):
    # Sample the random inputs (they are just standard normal)
    x = np.random.randn(solver.num_input) # solver.num_input is just 3
    # Evaluate the solver response
    y = solver(x) # This returns an (num timesteps) x (num states) array (100 x 2 her
    # Plot the sample
    ax1.plot(solver.t, y[:, 0])
    ax2.plot(solver.t, y[:, 1], label='Sample {0:d}'.format(i+1))
plt.legend(loc=True)
```

Out[15]: <matplotlib.legend.Legend at 0x7ffebbf50e20>





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

In [16]:
```python
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
    """
    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:

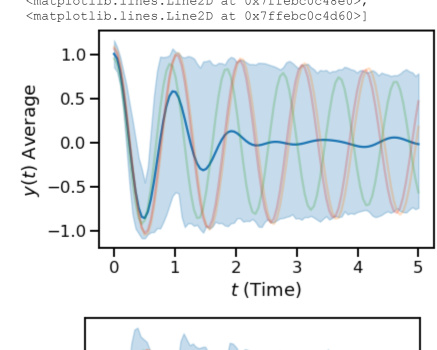
In [17]:
```python
samples = take_samples_from_solver(50)
# print(samples.shape)
# print(samples)
```

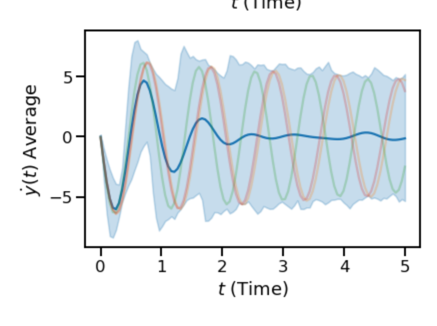
Part A

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).

In [18]:
```python
samples = take_samples_from_solver(100)

# number of samples sets to plot for confirmation
num_cases = 3

# hold average position and average velocity at each timestamp
avgs = np.empty([samples.shape[0], samples.shape[2]])
for i in range(samples.shape[0]):
    avgs[i,0] = np.mean(samples[:,i,0])
    avgs[i,1] = np.mean(samples[:,i,1])

# hold 2.5 and 97.5 percentiles
confidence_bounds = np.empty([samples.shape[0], samples.shape[2]+2])
for i in range(samples.shape[0]):
    confidence_bounds[i,0] = np.percentile(samples[:,i,0], 2.5)
    confidence_bounds[i,1] = np.percentile(samples[:,i,0], 97.5)
    confidence_bounds[i,2] = np.percentile(samples[:,i,1], 2.5)
    confidence_bounds[i,3] = np.percentile(samples[:,i,1], 97.5)

# plot some positions and the average at each timestep
fig1, ax1 = plt.subplots()
ax1.set_xlabel('$t$ (Time)')
ax1.set_ylabel('$y(t)$ Average')
ax1.plot(solver.t, avgs[:,0])
timesteps = np.array([solver.t for i in range(num_cases)])
ax1.fill_between(solver.t, confidence_bounds[:,0], confidence_bounds[:,1], color=sns.c
ax1.plot(timesteps.T, samples[0:num_cases,:,0].T, alpha=0.25)

# plot some velocities and the average at each timestep
fig2, ax2 = plt.subplots()
ax2.set_xlabel('$t$ (Time)')
ax2.set_ylabel('$\dot{y}(t)$ Average')
ax2.plot(solver.t, avgs[:,1])
ax2.fill_between(timesteps[0,:], confidence_bounds[:,2], confidence_bounds[:,3], colo
ax2.plot(timesteps.T, samples[0:num_cases,:,1].T, alpha=0.25)
```

Out[18]: [<matplotlib.lines.Line2D at 0x7ffebc0c4e80>,
<matplotlib.lines.Line2D at 0x7ffebc0c48e0>,
<matplotlib.lines.Line2D at 0x7ffebc0c4d60>]



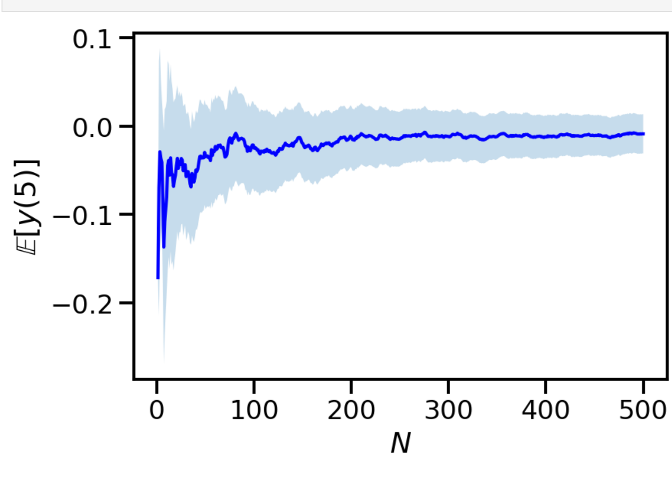


Part B

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

Solution:

In [19]:
```python
# NOTE: The following is based on hand-on activities provided by the professor
N = 500
t5_pos = take_samples_from_solver(N)[:, 5, 0]

# samples averages
t5_pos_avg = np.cumsum(t5_pos) / np.arange(1, N + 1)

# sample square averages
t5_pos2_avg = np.cumsum(t5_pos ** 2) / np.arange(1, N + 1)

# sample variances
t5_pos_sig2 = t5_pos2_avg - t5_pos_avg ** 2

# samples 2.5 and 97.5 percentiles
t5_pos_2_5 = t5_pos_avg - 2.0 * np.sqrt(t5_pos_sig2 / np.arange(1, N + 1))
t5_pos2_97_5 = t5_pos_avg + 2.0 * np.sqrt(t5_pos_sig2 / np.arange(1, N + 1))

# plot position estimates and confidence intervals
fig, ax = plt.subplots(dpi=150)
ax.fill_between(np.arange(1, N + 1), t5_pos_2_5, t5_pos2_97_5, alpha=0.25)
ax.plot(np.arange(1, N+1), t5_pos_avg, 'b', lw=2)
ax.set_xlabel('$N$')
ax.set_ylabel(r'$\mathbb{E}[y(5)]$');
```



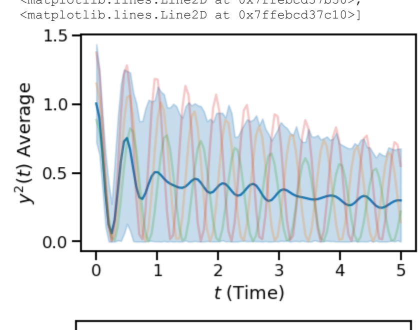
Part C

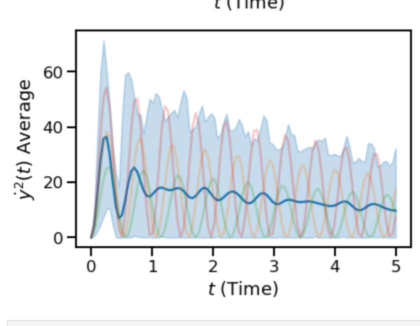
Repeat part A and B for the squared response. That is, do exactly the same thing as above, but consider $y^2(t)$ and $\dot{y}^2(t)$ instead of $y(t)$ and $\dot{y}(t)$. How many samples do you need to estimate the mean squared response at $t = 5$s with negligible epistemic uncertainty?

Solution:

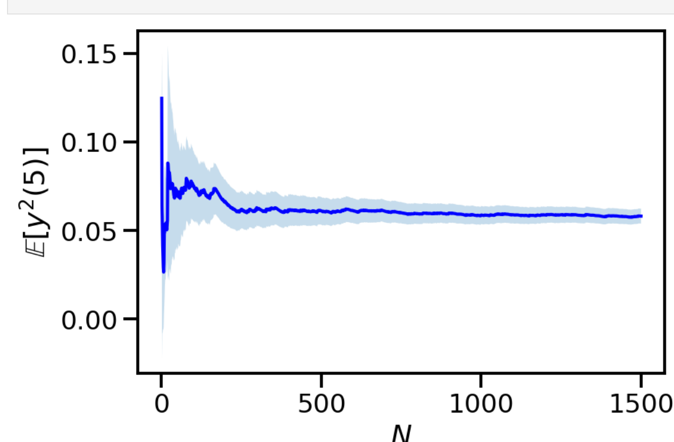
```python
# redoing Part A
K = 100
samples = take_samples_from_solver(K)

# square the value of all our samples
for k in range(K):
    samples[k,:,0] = np.square(samples[k,:,0])
    samples[k,:,1] = np.square(samples[k,:,1])

# number of samples sets to plot for confirmation
num_cases = 3

# hold average position and average velocity at each timestamp
avgs = np.empty([samples.shape[0], samples.shape[2]])
for i in range(samples.shape[0]):
    avgs[i,0] = np.mean(samples[:,i,0])
    avgs[i,1] = np.mean(samples[:,i,1])

# hold 2.5 and 97.5 percentiles
confidence_bounds = np.empty([samples.shape[0], samples.shape[2]+2])
for i in range(samples.shape[0]):
    confidence_bounds[i,0] = np.percentile(samples[:,i,0], 2.5)
    confidence_bounds[i,1] = np.percentile(samples[:,i,0], 97.5)
    confidence_bounds[i,2] = np.percentile(samples[:,i,1], 2.5)
    confidence_bounds[i,3] = np.percentile(samples[:,i,1], 97.5)

# plot some positions and the average at each timestep
fig1, ax1 = plt.subplots()
ax1.set_xlabel('$t$ (Time)')
ax1.set_ylabel('$y^2(t)$ Average')
ax1.plot(solver.t, avgs[:,0])
timesteps = np.array([solver.t for i in range(num_cases)])
ax1.fill_between(timesteps[0,:], confidence_bounds[:,0], confidence_bounds[:,1], colo
ax1.plot(timesteps.T, samples[0:num_cases,:,0].T, alpha=0.25)

# plot some velocities and the average at each timestep
fig2, ax2 = plt.subplots()
ax2.set_xlabel('$t$ (Time)')
ax2.set_ylabel('$\dot{y}^2(t)$ Average')
ax2.plot(solver.t, avgs[:,1])
ax2.fill_between(timesteps[0,:], confidence_bounds[:,2], confidence_bounds[:,3], colo
ax2.plot(timesteps.T, samples[0:num_cases,:,1].T, alpha=0.25)
```

```
[<matplotlib.lines.Line2D at 0x7ffebcd37af0>,
 <matplotlib.lines.Line2D at 0x7ffebcd37b50>,
 <matplotlib.lines.Line2D at 0x7ffebcd37c10>]
```






```python
# redoing Part B
# NOTE: The following is based on hand-on activities provided by the professor
N = 1500

# all we need to change is the starting position samples (square them)
t5_pos = np.square(take_samples_from_solver(N)[:, 5, 0])

# samples averages
t5_pos_avg = np.cumsum(t5_pos) / np.arange(1, N + 1)

# sample square averages
t5_pos2_avg = np.cumsum(t5_pos ** 2) / np.arange(1, N + 1)

# sample variances
t5_pos_sig2 = t5_pos2_avg - t5_pos_avg ** 2

# samples 2.5 and 97.5 percentiles
t5_pos_2_5 = t5_pos_avg - 2.0 * np.sqrt(t5_pos_sig2 / np.arange(1, N + 1))
t5_pos2_97_5 = t5_pos_avg + 2.0 * np.sqrt(t5_pos_sig2 / np.arange(1, N + 1))

# plot position estimates and confidence intervals
fig, ax = plt.subplots(dpi=150)
ax.fill_between(np.arange(1, N + 1), t5_pos_2_5, t5_pos2_97_5, alpha=0.25)
ax.plot(np.arange(1, N+1), t5_pos_avg, 'b', lw=2)
ax.set_xlabel('$N$')
ax.set_ylabel(r'$\mathbb{E}[y^2(5)]$');
```



We can see that $N > 1000$ would be ideal to reach a level of negligible epistemic uncertainty. You can compare this to our starting case that only needed around $N = 400$.

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:

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

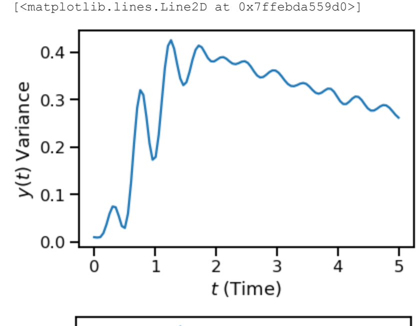
and similarly for $\dot{y}(t)$, to estimate the variance of the position and the velocity with negligible epistemic uncertainty. Plot both quantities as a function of time.

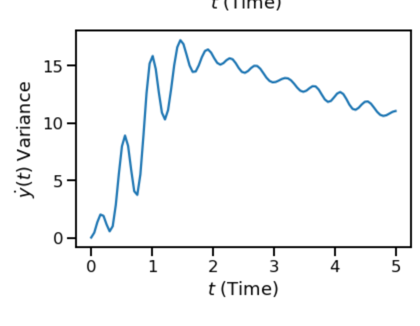
Solution:

We know from earlier parts of this question that we'll need roughly $N > 1000$ samples at each timestep to get an acceptable estimate of the requested variances. Let's use $N = 1500$ to be safe.

```python
# variance of the positions and velocities at each timestep
samples = take_samples_from_solver(1500)
numSamples = samples.shape[0]
numTimesteps = samples.shape[1]
numMeasurements = samples.shape[2]

# hold average position, average velocity, and squared averages at each timestamp
avgs = np.empty([numTimesteps, numMeasurements * 2])
for i in range(numTimesteps):
    avgs[i,0] = np.mean(samples[:,i,0])
    avgs[i,1] = np.mean(np.square(samples[:,i,0]))
    avgs[i,2] = np.mean(samples[:,i,1])
    avgs[i,3] = np.mean(np.square(samples[:,i,1]))

# hold variance of position and velocity at each timestamp
variances = np.empty([numTimesteps, numMeasurements])
for i in range(numTimesteps):
    variances[i,0] = avgs[i,1] - (avgs[i,0] ** 2) # variance of position
    variances[i,1] = avgs[i,3] - (avgs[i,2] ** 2) # variances of velocity

# plot position variance at each timestep
fig1, ax1 = plt.subplots()
ax1.set_xlabel('$t$ (Time)')
ax1.set_ylabel('$y(t)$ Variance')
ax1.plot(solver.t, variances[:,0])

# plot velocity variance at each timestep
fig2, ax2 = plt.subplots()
ax2.set_xlabel('$t$ (Time)')
ax2.set_ylabel('$\dot{y}(t)$ Variance')
ax2.plot(solver.t, variances[:,1])
```

```
[<matplotlib.lines.Line2D at 0x7ffebda559d0>]
```





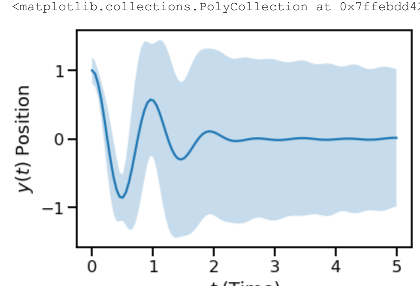
Part E

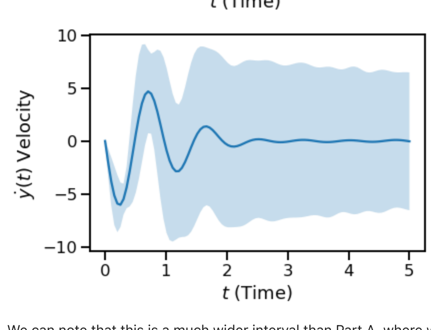
Put together the estimated mean and variance to plot a 95\% predictive interval for the position and the velocity as functions of time.

Solution:

```python
# hold 95% credible interval at each timestamp
positions = np.empty([numTimesteps, 3])
velocities = np.empty([numTimesteps, 3])
for i in range(numTimesteps):
    positions[i,0] = avgs[i, 0]
    positions[i,1] = avgs[i, 0] - 1.96*np.sqrt(variances[i,0])
    positions[i,2] = avgs[i, 0] + 1.96*np.sqrt(variances[i,0])
    velocities[i,0] = avgs[i, 2]
    velocities[i,1] = avgs[i, 2] - 1.96*np.sqrt(variances[i,1])
    velocities[i,2] = avgs[i, 2] + 1.96*np.sqrt(variances[i,1])

# plot position variance at each timestep
fig1, ax1 = plt.subplots()
ax1.set_xlabel('$t$ (Time)')
ax1.set_ylabel('$y(t)$ Position')
ax1.plot(solver.t, positions[:,0])
ax1.fill_between(solver.t, positions[:,1], positions[:,2], alpha=0.25)

# plot velocity variance at each timestep
fig2, ax2 = plt.subplots()
ax2.set_xlabel('$t$ (Time)')
ax2.set_ylabel('$\dot{y}(t)$ Velocity')
ax2.plot(solver.t, velocities[:,0])
ax2.fill_between(solver.t, velocities[:,1], velocities[:,2], alpha=0.25)
```

```
<matplotlib.collections.PolyCollection at 0x7ffebdd428b0>
```





We can note that this is a much wider interval than Part A, where we only used 100 samples (too little to get acceptable results for the squared value used in the variance calculation).

-End-