Stochastic Response

Richard M. Murray, 6 Feb 2022 (updated 9 Feb 2023)

This notebook illustrates the implementation of random processes and stochastic response. We focus on a system of the form

$$\dot{X} = AX + FV \qquad X \in \mathbb{R}^n$$

where V is a white noise process and the system is a first order linear system.

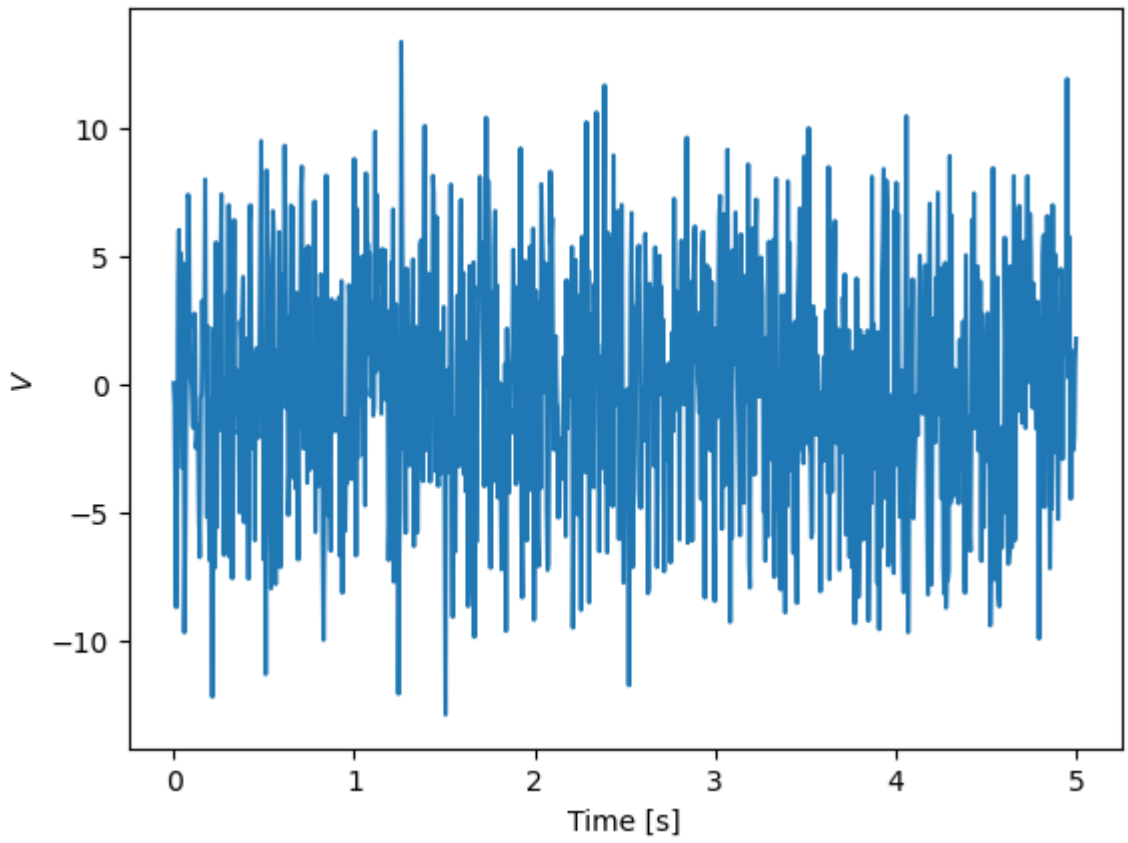
In [1]:
```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
import control as ct
from math import sqrt, exp
```

First order linear system

We start by looking at the stochastic response for a first order linear system

$$\dot{X} = -aX + V, \qquad Y = CX$$
$$\mathbb{E}(V) = 0, \quad \mathbb{E}(V^\mathsf{T}(t_1)V(t_2)) = 0.1\,\delta(t_1 - t_2)$$

In [2]:
```
# First order system
a = 1
c = 1
sys = ct.tf(c, [1, a])

# Create the time vector that we want to use
Tf = 5
T = np.linspace(0, Tf, 1000)
dt = T[1] - T[0]

# Create the basis for a white noise signal
# Note: use sqrt(Q/dt) for desired covariance
Q = np.array([[0.1]])
# V = np.random.normal(0, sqrt(Q[0,0]/dt), T.shape)
V = ct.white_noise(T, Q)

plt.plot(T, V[0])
plt.xlabel('Time [s]')
plt.ylabel('$V$');
```

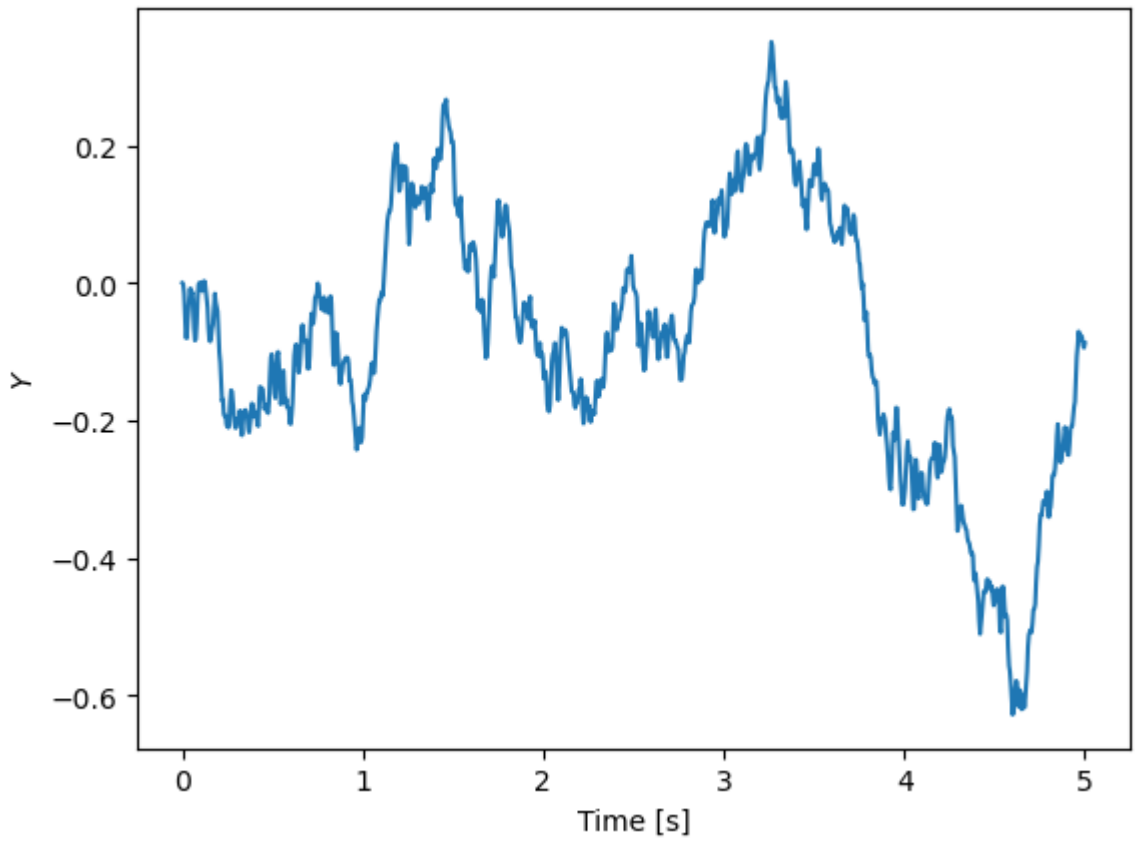


Note that the magnitude of the signal seems to be much larger than Q. This is because we have a Guassian process $\implies$ the signal consists of a sequence of "impulse-like" functions that have magnitude that increases with the time step dt as $1/\sqrt{dt}$ (this gives covariance $\mathbb{E}(V(t_1)V^T(t_2)) = Q\delta(t_2 - t_1)$.

In [3]:
```
# Calculate the sample properties and make sure they match
print("mean(V) [0.0] = ", np.mean(V))
print("cov(V) * dt [%0.3g] = " % Q, np.round(np.cov(V), decimals=3) * dt)
```

```
mean(V) [0.0] =  -0.09905286968849307
cov(V) * dt [0.1] =  0.10293293293293293
```

In [4]:
```
# Response of the first order system
# Scale white noise by sqrt(dt) to account for impulse
T, Y = ct.forced_response(sys, T, V)
plt.plot(T, Y)
plt.xlabel('Time [s]')
plt.ylabel('$Y$');
```

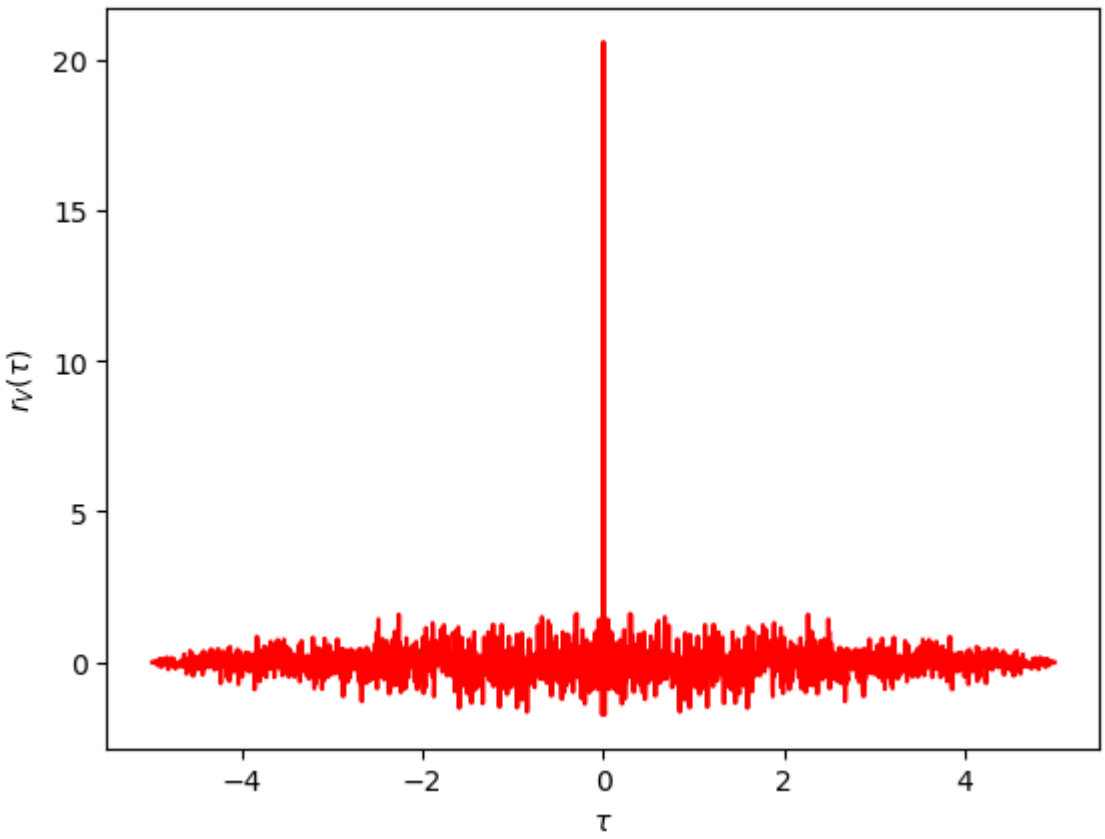


This is a first order system, and so we can use the calculation from the course notes to compute the analytical correlation function and compare this to the sampled data:

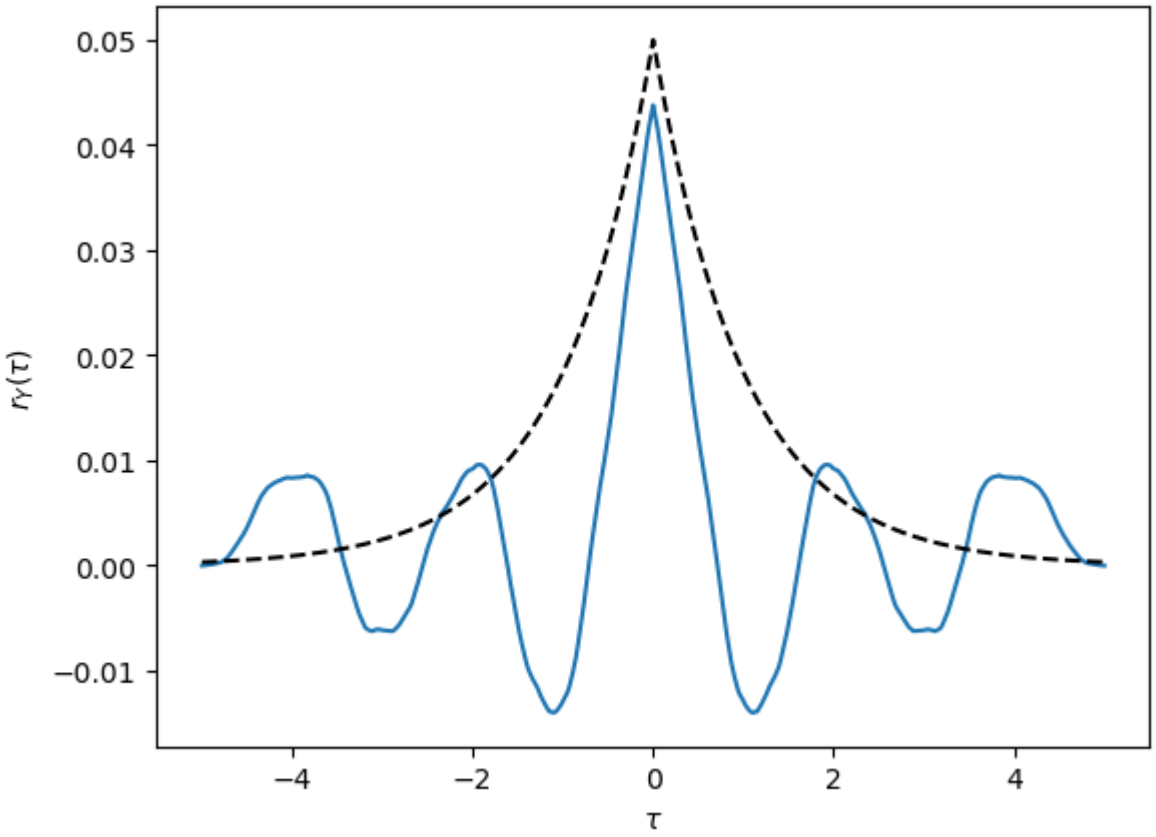
In [5]:
```
# Compare static properties to what we expect analytically
def r(tau):
    return c**2 * Q / (2 * a) * exp(-a * abs(tau))

print("* mean(Y) [%0.3g] = %0.3g" % (0, np.mean(Y)))
print("* cov(Y) [%0.3g] = %0.3g" % (r(0), np.cov(Y)))
```

```
* mean(Y) [0] = -0.0825
* cov(Y) [0.05] = 0.037
```

In [6]:
```
# Correlation function for the input
# Scale by dt to take time step into account
# r_V = sp.signal.correlate(V, V) * dt / Tf
# tau = sp.signal.correlation_lags(len(V), len(V)) * dt
tau, r_V = ct.correlation(T, V)

plt.plot(tau, r_V, 'r-')
plt.xlabel(r'$\tau$')
plt.ylabel(r'$r_V(\tau)$');
```



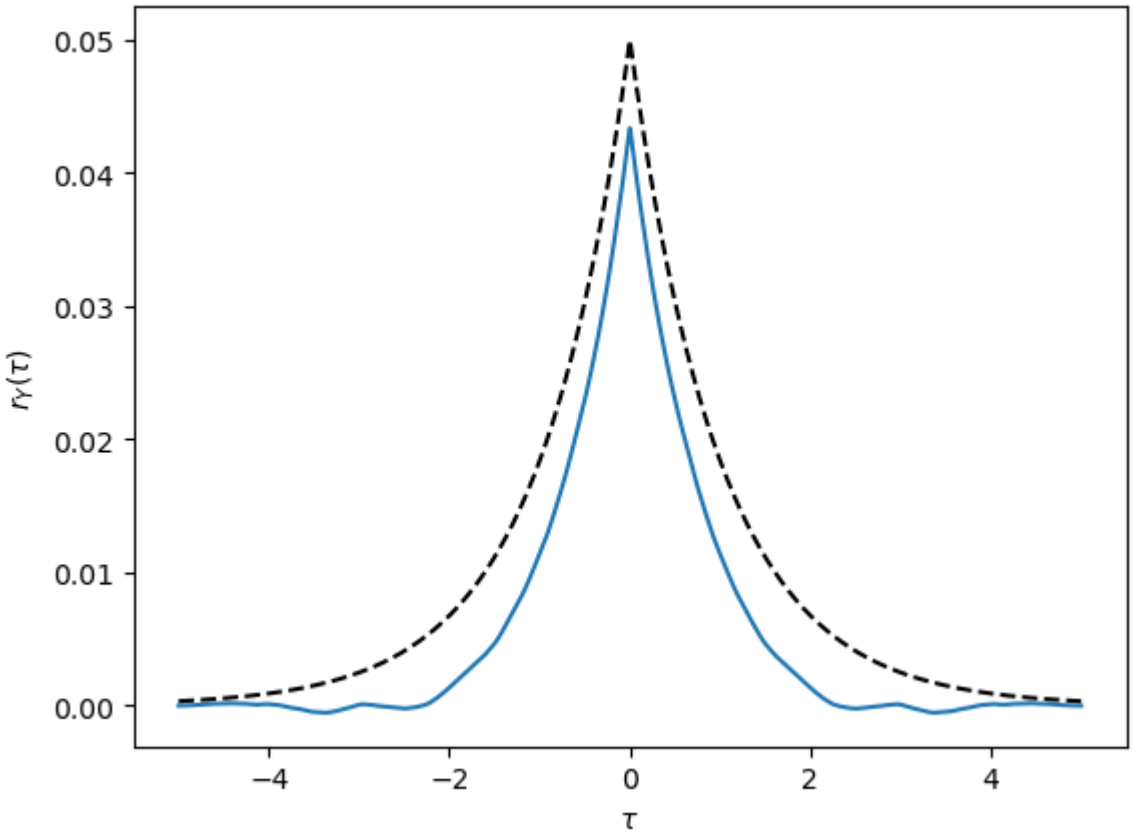
In [7]:
```
# Correlation function for the output
# r_Y = sp.signal.correlate(Y, Y) * dt / Tf
# tau = sp.signal.correlation_lags(len(Y), len(Y)) * dt
tau, r_Y = ct.correlation(T, Y)
plt.plot(tau, r_Y)
plt.xlabel(r'$\tau$')
plt.ylabel(r'$r_Y(\tau)$')

# Compare to the analytical answer
plt.plot(tau, [r(t)[0, 0] for t in tau], 'k--');
```



The analytical curve may or may not line up that well with the correlation function based on the sample. Try running the code again from the top to see how things change based on the specific random sequence chosen at the start.

Note: the *right* way to compute the correlation function would be to run a lot of different samples of white noise filtered through the system dynamics and compute $R(t_1, t_2)$ across those samples.

In [8]:
```
# As a crude approximation, compute the average correlation
r_avg = np.zeros_like(r_Y)
for i in range(100):
    V = ct.white_noise(T, Q)
    _, Y = ct.forced_response(sys, T, V)
    tau, r_Y = ct.correlation(T, Y)
    r_avg = r_avg + r_Y
r_avg = r_avg / i
plt.plot(tau, r_avg)
plt.xlabel(r'$\tau$')
plt.ylabel(r'$r_Y(\tau)$')

# Compare to the analytical answer
plt.plot(tau, [r(t)[0, 0] for t in tau], 'k--');
```



Dryden gust model

Friedland, *Control Systems Design*, Example 10B

Based on experimental data, the power spectral density for the vertical component of random wind velocity in turbulent air can be modeled as

$$S(\omega) = \sigma_z^2 T \frac{1 + 3(\omega T)^2}{[1 + (\omega T)^2]^2},$$

where σ_z and T are parameters that depend on the wind characteristics.

This power spectral density can be modeled using white noise by running it through a linear system with transfer fucntion

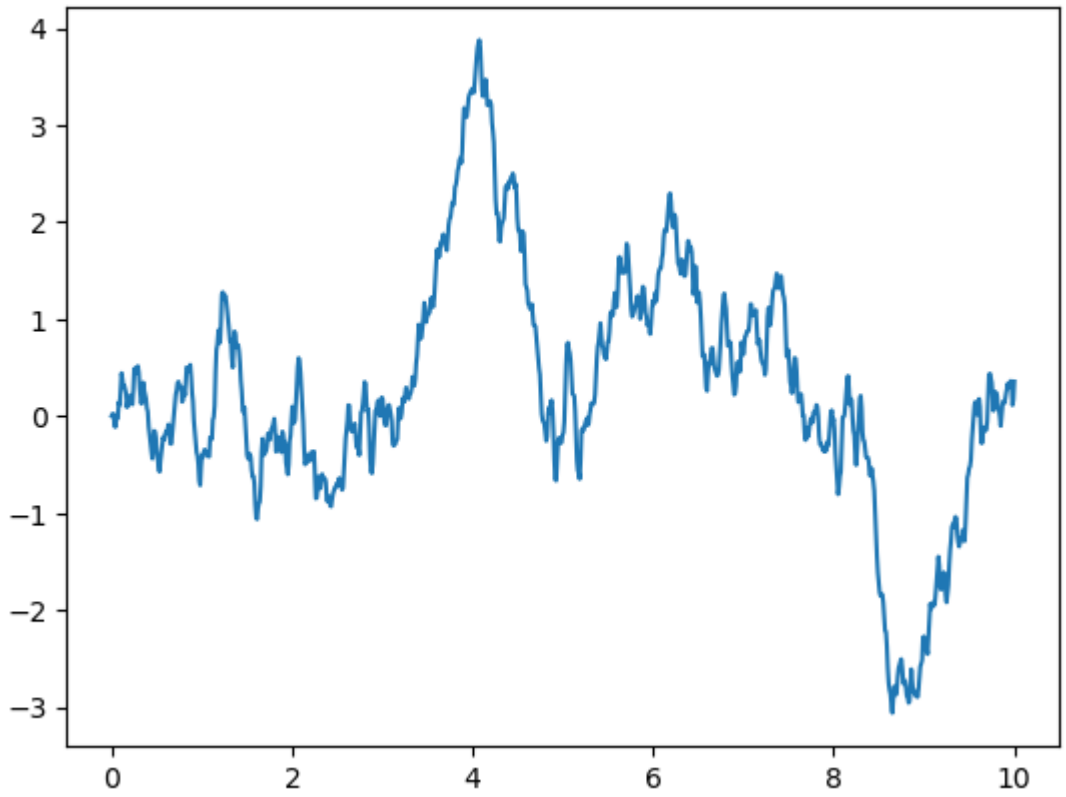
$$H(s) = \frac{1 + \sqrt{3}T}{(1 + Ts)^2}.$$

A state space realization for this transfer function is given by

$$\dot{X} = \begin{bmatrix} 0 & 1 \\ -\frac{1}{T^2} & -\frac{2}{T} \end{bmatrix} X + \begin{bmatrix} 0 \\ 1 \end{bmatrix} V$$
$$Y = \begin{bmatrix} \frac{1}{T^2} & \frac{\sqrt{3}}{T} \end{bmatrix}$$

To create a disturbance signal with the characteristics of the Dryden gust model, we create a linear system with the given parameters and computing the input/output response to white noise:

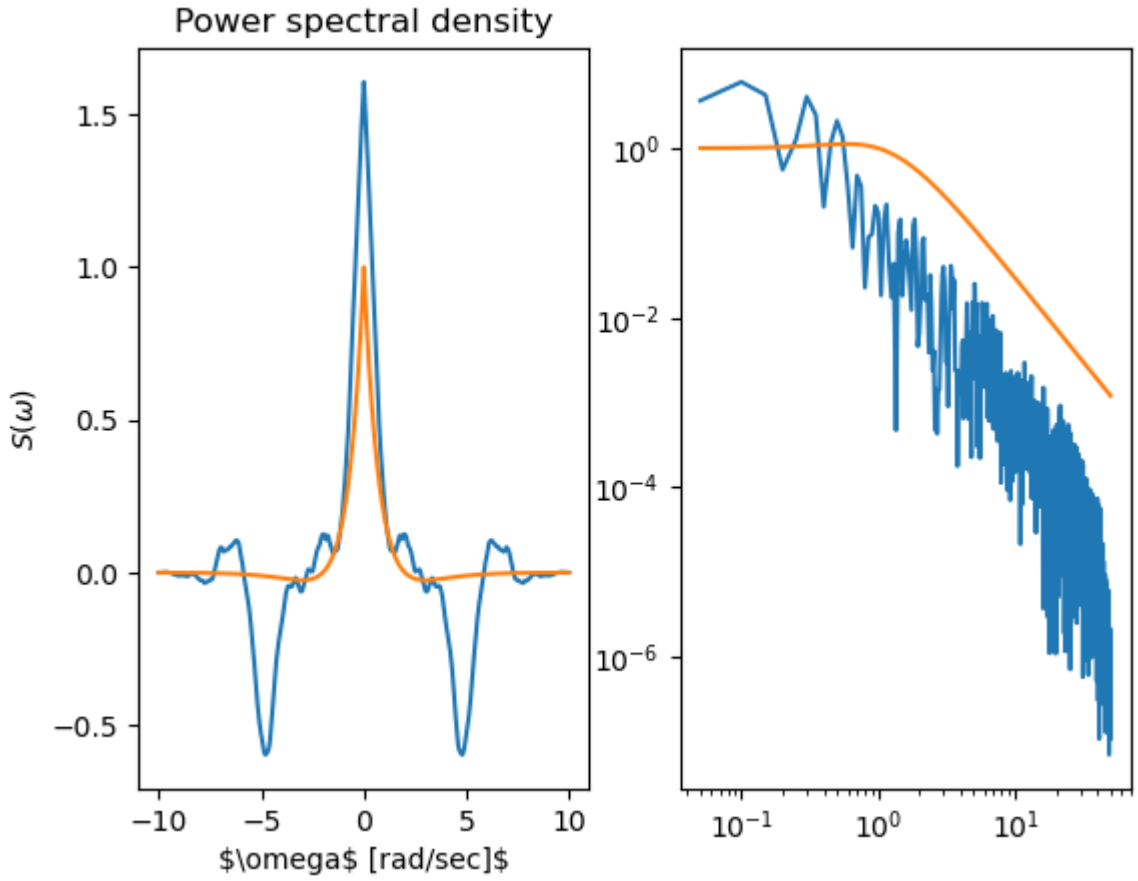
In [9]:
```
sigma_z = 1
T = 1
filter = ct.ss([[0, 1], [-1/T**2, -2/T]], [[0], [1]], [[1/T**2, sqrt(3)/T]],

timepts = np.linspace(0, 10, 1000)
V = ct.white_noise(timepts, sigma_z**2)
resp = ct.input_output_response(filter, timepts, V)

plt.plot(resp.time, resp.outputs);
```



We can compute the correlation function and power spectral density to confirm that we match the desired characteristics:

In [10]:
```
# Compute the correlation function
tau, R = ct.correlation(resp.time, resp.outputs)

# Analytical expression for the correlation function (see Friedland)
def dryden_corrfcn(tau, sigma_z=1, T=1):
    return sigma_z**2 * np.exp(-np.abs(tau)/T) * (1- np.abs(tau)/(2*T))

# Plot the correlation function
fig, axs = plt.subplots(1, 2)
axs[0].plot(tau, R)
axs[0].plot(tau, dryden_corrfcn(tau))
axs[0].set_xlabel("$\tau$")
```

```
axs[0].set_ylabel("$r(\tau)$")
axs[0].set_title("Correlation function")

# Compute the power spectral density
dt = timepts[1] - timepts[0]
S = sp.fft.rfft(R) * dt * 2            # rfft returns omega >= 0 => muliple
omega = sp.fft.rfftfreq(R.size, dt)

# Analytical expression for the correlation function (see Friedland)
def dryden_psd(omega, sigma_z=1., T=1.):
    return sigma_z**2 * T * (1 + 3 * (omega * T)**2) / (1 + (omega * T)**2)*

# Plot the power spectral density
axs[1].loglog(omega[1:], np.abs(S[1:]))
axs[1].loglog(omega[1:], dryden_psd(omega[1:]))
axs[0].set_xlabel("$\omega$ [rad/sec]$")
axs[0].set_ylabel("$S(\omega)$")
axs[0].set_title("Power spectral density");
```



In [ ]: