In [6]:
```python
import numpy as np
import scipy.linalg as sp

'''
(a) Using the standard Kalman filter, for various k values, what
is the variance of the posterior estimation error?
'''

# Define global vars

# Time-invariant scalar linear system as given in problem statement
N = 1
A = 1.2
H = 1

# Process and measurement noise covariances
V, W = 2, 1

# Process and measurement noise are zero mean, gaussian, and independant

'''
Function that returns value corresponding to Gaussian dist matching
required mean/variance for process and measurement noises.
Note that mean = 0 and var = I for both v(k) and w(k)
'''


def r_normal(Ex, Var): return np.random.normal(Ex, Var, 1)

# Define scalar measurement and time update for Kalman filter implementation


def time_update(xm, Pm):
    xp = A*xm
    Pp = A*Pm*A + V
    return xp, Pp


def meas_update(xp, Pp, z):
    K = Pp*H*(H*Pp*H + W)**-1
    xm = xp + K*(z - H*xp)
    Pm = (np.eye(N) - K*H)*Pp*(np.eye(N) - K*H) + K*W*K
    return xm, Pm


# Define simulation of true system dynamics


def sym_sys(x_true):
    v_k = r_normal(0, V)  # Process noise
    w_k = r_normal(0, W)  # measurement noise
    x_true = A * x_true + v_k
    z = H * x_true + w_k
    return x_true, z


# Initialize estimate and covariance of state (at k = 0)
xm, Pm = 0, np.array([[3]])

# Initialize x_true at k = 0
x_true = r_normal(0, 3)

T_f = [2, 3, 11, 1001]  # Simulation Timesteps (0 included)
Var_est = np.zeros((4, 1))

for config in T_f:

    for k in range(1, config):  # Note that estimate for k=0 is initialized above

        # Simulate system and measurement
        x_true, z = sym_sys(x_true)

        # Kalman filter estimation: time update
        xp, Pp = time_update(xm, Pm)

        # Kalman filter estimation: measurement update
        xm, Pm = meas_update(xp, Pp, z)

    # Print variance of posterior estimation error (Pm(k)) for each config
    print('Variance of posterior estimation error for k = '
          + repr(k) + ' is: ' + repr(round(Pm[0, 0], 4)))
```

```
Variance of posterior estimation error for k = 1 is: 0.8634
Variance of posterior estimation error for k = 2 is: 0.7561
Variance of posterior estimation error for k = 10 is: 0.7554
Variance of posterior estimation error for k = 1000 is: 0.7554
```

In [7]:
```python
'''
(b) What is the steady-state Kalman filter for this system? Provide the
gain, and the steady-state posterior variance.
'''

Pinf = sp.solve_discrete_are(A, H, V, W)  # This is Pp as k -> inf
Kinf = Pinf * H * (H * Pinf * H + W) ** -1

'''
Use time update to transform Pinf (Pp as k -> inf)
into KF posterior variance (Pm as k -> inf)

Note: Pp converges to Pinf as k -> inf. Since K(k) also converges to to Kinf,
Pm must converge to our SSKF posterior variance using a measurement update.
'''
Pm_inf = (np.eye(N) - Kinf*H)*Pp*(np.eye(N) - Kinf*H) + Kinf*W*Kinf

print('Steady state KF gain is: ' + repr(round(Kinf[0, 0], 4)))
print('Steady state KF posterior variance is: ' + repr(round(Pm_inf[0, 0], 4)))
```

```
Steady state KF gain is: 0.7554
Steady state KF posterior variance is: 0.7554
```

In [8]:
```python
'''
(c) Using now the steady-state Kalman filter to estimate the state,
at various values for k, what is the variance of the posterior estimation
 error? Comment on how this compares to the standard Kalman filter of part a.
'''

# Define scalar measurement update for SSKF implementation
# Note that this functions is equivalent to above with SSKF gain


def meas_update_ss(xp, Pp, z):
    xm = xp + Kinf*(z - H*xp)  # equivalent to A * xp + Kinf * z (from notes)
    Pm = (np.eye(N) - Kinf*H)*Pp*(np.eye(N) - Kinf*H) + Kinf*W*Kinf
    return xm, Pm


# Initialize estimate and covariance of state (at k = 0)
xm, Pm = 0, np.array([[3]])

# Initialize x_true at k = 0
x_true = r_normal(0, 3)

T_f = [2, 3, 11, 1001]  # Simulation Timesteps (0 included)
Var_est = np.zeros((4, 1))

for config in T_f:

    for k in range(1, config):  # Note that estimate for k=0 is initialized above

        # Simulate system and measurement
        x_true, z = sym_sys(x_true)

        # Kalman filter estimation: time update
        xp, Pp = time_update(xm, Pm)

        # Kalman filter estimation: measurement update
        xm, Pm = meas_update_ss(xp, Pp, z)

    # Print variance of posterior estimation error (Pm(k)) for each config
    print('Variance of posterior estimation error for k = '
          + repr(k) + ' is: ' + repr(round(Pm[0, 0], 4)))

print('The steady state Kalman Filter had slightly worse performance (higher'
      ' variance of posterior estimation error) than the standard Kalman'
      ' filter for k = 1 and 2. By k = 10 and 1000, both filters performed'
      ' satisfactorily with Pm ~= Pm_inf ')
```

```
Variance of posterior estimation error for k = 1 is: 0.9488
Variance of posterior estimation error for k = 2 is: 0.7568
Variance of posterior estimation error for k = 10 is: 0.7554
Variance of posterior estimation error for k = 1000 is: 0.7554
The steady state Kalman Filter had slightly worse performance (higher variance of posterior estimation error) than the standard Kalm
an filter for k = 1 and 2. By k = 10 and 1000, both filters performed satisfactorily with Pm ~= Pm_inf
```

In [ ]:

In [13]:

```python
import numpy as np
import scipy.linalg as sp
import sys
import warnings
warnings.filterwarnings("ignore")


# (a)Compute the steady-state Kalman filter gain Kinf, for the given system

# Define global vars

# Time-invariant scalar linear system as given in problem statement
N = 1
A = 1
H = 1

# Process and measurement noise covariances
V, W = 1.0, 6.0

# Process and measurement noise are zero mean, gaussian, and independant

Pinf = sp.solve_discrete_are(A, H, V, W)  # No transpose on scalar A, H
Kinf = Pinf * H * (H * Pinf * H + W) ** -1

print('The steady-state Kalman filter gain, Kinf is: '
      + repr(round(Kinf[0, 0], 4)))
```

```
The steady-state Kalman filter gain, Kinf is: 0.3333
```

In [14]:
```
'''
(b) The true system dynamics differ from the ones used to derive the estimator
and are driven by parameter delta. For what parameters delta will the error
e(k) remain bounded as k tends to infinity?
'''

# To test this, we run our KF and analyze e(k) for varying values of delta

'''Function that returns value corresponding to Gaussian dist matching
 required mean/variance for process and measurement noises.'''


def r_normal(Ex, Var): return np.random.normal(Ex, Var, 1)

# Define scalar measurement and time update for Kalman filter implementation


def time_update(xm, Pm):
    xp = A*xm
    Pp = A*Pm*A + V
    return xp, Pp


def meas_update(xp, Pp, z):
    K = Pp*H*(H*Pp*H + W)**-1
    xm = xp + K*(z - H*xp)
    Pm = (np.eye(N) - K*H)*Pp*(np.eye(N) - K*H) + K*W*K
    return xm, Pm


# Define simulation of true system dynamics


def sym_sys(x_true, delta):
    v_k = r_normal(0, V)  # Process noise
    w_k = r_normal(0, W)  # measurement noise
    x_true = (1 + delta) * x_true + v_k
    z = H * x_true + w_k
    return x_true, z


# Initialize estimate and covariance of state (at k = 0)
xm, Pm = 0.0, np.array([[3]])

# Initialize x_true at k = 0
x_true = r_normal(0, 3)

delta_list = np.array([-10, -2.01, -2.0, -1.0,
                       -0.01, 0, 0.01, 1, 10], dtype=float)
T_f = 100000  # Simulation Timesteps (0 included)
e_inf = np.zeros(len(delta_list))

max_float = sys.float_info.max  # Prevent overflow on next step

for i, config in np.ndenumerate(delta_list):

    for k in range(1, T_f):  # Note that k=0 is initialized above

        if xm < max_float:  # prevent stack overflow

            # Simulate system and measurement
            x_true, z = sym_sys(x_true, config)

            # Kalman filter estimation: time update
            xp, Pp = time_update(xm, Pm)

            # Kalman filter estimation: measurement update
            xm, Pm = meas_update(xp, Pp, z)

            # Store e(10,000)
            e_inf[i] = x_true - xm

        else:
            e_inf[i] = float("inf")  # representing an unbounded error

    # Initialize estimate and covariance of state (at k = 0) for next config
    xm, Pm = 0.0, np.array([[3]])

    # Initialize x_true at k = 0 for next config
    x_true = r_normal(0, 3)

print('Our maximum float of: ' + repr(max_float) +
      ' is sufficiently large to approximate unboundedness.')

for i, val in np.ndenumerate(e_inf):

    if val != float("inf"):
```

```python
        print('A delta value of ' + repr(round(delta_list[i], 4)) +
              ' produced a bounded error as k approaches infinity.')

print('From this information and an analytical proof (see writing),'
      ' we can deduce that e(k) remains bounded for delta in [-2, 0].')
```

```
Our maximum float of: 1.7976931348623157e+308 is sufficiently large to approximate unboundedness.
A delta value of -2.0 produced a bounded error as k approaches infinity.
A delta value of -1.0 produced a bounded error as k approaches infinity.
A delta value of -0.01 produced a bounded error as k approaches infinity.
A delta value of 0.0 produced a bounded error as k approaches infinity.
From this information and an analytical proof (see writing), we can deduce that e(k) remains bounded for delta in [-2, 0].
```

In [ ]:

Problem 2(b). Using the true system and your KF estimation $e(k) := x_{true}(k) - \hat{x}(k)$. For what values δ will the error $e(k)$ remain bounded as $k \to \infty$?

- For $e(k)$ to remain bounded with the model mismatch as described, both $e(k)$ and $x_{true}(k)$ need to converge. In other words, the following vector must converge over time.

$$\begin{bmatrix} e(k) \\ x_{true}(k) \end{bmatrix}$$

- Now we write the equations describing these terms in matrix form.

$$\begin{bmatrix} e(k) \\ x_{true}(k) \end{bmatrix} = \begin{bmatrix} (I - K_\infty H)A\,e(k-1) + (I - K_\infty H)v(k-1) - K_\infty w(k) \\ (1+\delta)\,x_{true}(k-1) \end{bmatrix}$$

$$\begin{bmatrix} e(k) \\ x_{true}(k) \end{bmatrix} = \begin{bmatrix} (I - K_\infty H)A & \emptyset \\ \emptyset & 1+\delta \end{bmatrix} \begin{bmatrix} e(k-1) \\ x_{true}(k-1) \end{bmatrix} + \dots$$

Need this matrix to be stable (all eigs in unit circle) for error and x_{true} not to diverge.

- The eigenvalues of a diagonal matrix $\text{diag}(\lambda_1, \lambda_2)$ are λ_1 and λ_2 themselves.

$$\therefore \text{eig}\left(\begin{bmatrix} (I - K_\infty H)A & \emptyset \\ \emptyset & 1+\delta \end{bmatrix}\right) = \text{eig}\left(\begin{bmatrix} (1 - \tfrac{1}{3}(1))(1) & \emptyset \\ \emptyset & 1+\delta \end{bmatrix}\right)$$

$$= \text{eig}\left(\begin{bmatrix} 2/3 & \emptyset \\ \emptyset & 1+\delta \end{bmatrix}\right) \Rightarrow \lambda = 2/3 \text{ and } 1+\delta$$

- For stability, $-1 \leq 1+\delta \leq 1$

$$\boxed{\therefore\ -2 \leq \delta \leq \emptyset}$$

- This is also confirmed numerically in Python.

In [8]:
```python
import numpy as np
import scipy.linalg as sp

'''
You are investigating various trade-offs in the design of an autonomous blimp.
The system dynamics are given, with the system state x(k)
comprising the height (x1 in units of m) and vertical velocity (x2 in units of
m/s) of the blimp, driven by a random acceleration.
'''


'''
(a) Write a program that computes Jp as a function of the usual Kalman filter
matrices A, V , H, and W. Use this program to confirm that the original system
 has Jp ~= 3:085m
'''

# Define global vars

dt, stdev, m1 = 1/10, 5, 10

# Time-invariant scalar linear system as given in problem statement
N = 2
A = np.array([[1, dt], [0, 1]])
H = np.array([[1, 0]])


def noise_orig_sys():
    # Process and measurement noise covariances
    V = stdev**2*np.array([[1/4*dt**4, 1/2*dt**3], [1/2*dt**3, dt**2]])
    W = m1**2
    return V, W
    # Noises are zero mean, gaussian, and independant


def compute_jp():

    '''
    The limit of Var[x1(k)|z(1:k)] as k->inf will be the upper-left component
    of the steady-state KF posterior variance.
    '''
    if len(H) == 1:  # Check if H is scalar
        Pinf = sp.solve_discrete_are(A.T, H.T, V, W)
        Kinf = Pinf * H * (H * Pinf * H + W) ** -1
    else:
        Pinf = sp.solve_discrete_are(A.T, H.T, V, W)
        Kinf = Pinf @ H @ np.linalg.inv(H @ Pinf @ H + W)
    '''
    Use time update to transform Pinf (Pp as k -> inf)
    into KF posterior variance (Pm as k -> inf)
    '''
    Pm_inf = (np.eye(N) - Kinf*H)*Pinf*(np.eye(N) - Kinf*H) + Kinf*W*Kinf

    # Extract upper-left term in Pm_inf to obtain Jp
    return np.sqrt(Pm_inf[0, 0])


V, W = noise_orig_sys()
print('The original system has Jp ~= ' + repr(round(compute_jp(), 3)) + 'm')
```

```
The original system has Jp ~= 3.085m
```

In [9]:
```python
'''
We are considering various ways to improve the system's estimation performance,
enumerated below. Each modification is an independent modification of the
original system. For each of the following suggested modifications to the
system, compute the performance metric Jp
'''

'''
(b) Replace the GPS sensor with that of brand A, which has m1 = 5m
(the noise standard deviation is cut in half).
'''
m1 = 5
V, W = noise_orig_sys()
print('For part (b) Jp ~= ' + repr(round(compute_jp(), 3)) + 'm')
```

```
For part (b) Jp ~= 1.816m
```

In [10]:
```
'''
(c) Replace the GPS sensor with that of brand B, which has dt = 1/20 sec,
and m1 = 10m (i.e. the sensor returns equally noisy measurements,
 but twice as frequently).
'''
dt, m1 = 1/20, 10
A = np.array([[1, dt], [0, 1]])
V, W = noise_orig_sys()
print('For part (c) Jp ~= ' + repr(round(compute_jp(), 3)) + 'm')
```

```
For part (c) Jp ~= 2.208m
```

In [11]:
```
'''
(d) Retain the original GPS sensor, and add an airspeed sensor which gives the
additional measurement z2(k) = x2(k) + w2(k), where w2(k) is a zero-mean,
white noise sequence, independent of all quantities, and with
Var [w2(k)] = m2 ** 2 with m2 = 1m/s. The sensors z1 and z2 return data
at the same instants of time.
'''
dt, m1, m2, H = 1/10, 10, 1, np.eye(N)
A = np.array([[1, dt], [0, 1]])


def noise_part_d():
    # Process and measurement noise covariances
    V = stdev**2*np.array([[1/4*dt**4, 1/2*dt**3], [1/2*dt**3, dt**2]])
    W = np.array([[m1**2, 0], [0, m2**2]])
    return V, W
    # Noises are zero mean, gaussian, and independant


V, W = noise_part_d()
print('For part (d) Jp ~= ' + repr(round(compute_jp(), 3)) + 'm')
```

```
For part (d) Jp ~= 1.001m
```

In [12]:
```
'''
(e) Retain the original GPS sensor, and add a second, independent barometric
height sensor z2(k) = x1(k) + w2(k), where w2(k) is a zero-mean, white noise
sequence, independent of all quantities, and with Var [w2(k)] = m2 ** 2
with m2 = 10m. The sensors z1 and z2 return data at the same instants of
time.
'''

m2, H = 10, np.array([[1, 0], [1, 0]])
V, W = noise_part_d()
print('For part (e) Jp ~= ' + repr(round(compute_jp(), 3)) + 'm')
```

```
For part (e) Jp ~= 2.459m
```

In [13]:
```
'''
(f) Retain the original GPS sensor, and add a second identical GPS sensor from
the same manufacturer, so that z2(k) = x1(k) + w2(k), where w2(k) = w1(k).

Note: This problem can be interpreted two different ways. If the statement
"w2(k) = w1(k)" means that these RVs have the same distributions but are
independant, then the answer to part (f) is as follows.
'''


def noise_part_f_1():
    # Process and measurement noise covariances
    V = stdev**2*np.array([[1/4*dt**4, 1/2*dt**3], [1/2*dt**3, dt**2]])
    W = m1**2*np.eye(N)
    return V, W


V, W = noise_part_f_1()
print('For part (f) interpretation 1, Jp ~= ' + repr(round(compute_jp(), 3))
      + 'm')

'''
If instead the statement "w2(k) = w1(k)" means that these two RV's values are
always equivalent, then the answer to part (f) is a little more complicated.
When w2(k) = w1(k) = m1**2, the covariance matrix for W should be,
m1**2 * [[1, 1], [1, 1]]
However, this covariance matrix is not solvable by the DARE. If we instead
perturb the diagonal terms slightly, then the DARE will solve, but will
approach infinity as the diagonal terms approach 1.
'''


def noise_part_f_2(delta):
    # Process and measurement noise covariances
    V = stdev**2*np.array([[1/4*dt**4, 1/2*dt**3], [1/2*dt**3, dt**2]])
    W = m1**2*np.array([[1 + delta, 1], [1, 1 + delta]])
    return V, W


delta = 0.1
for i in range(0, 5):
    V, W = noise_part_f_2(delta)
    print('When delta = ' + repr(delta)
          + ', Jp ~= ' + repr(round(compute_jp(), 3))
          + 'm')
    delta /= 10

'''
Therefore, for part (f) it makes the most sense to say that Jp -> infinity
when w2(k) = w1(k). This makes sense because we have always assumed that
random noise variables are independant in order
for KF to have a solution. In this case, w1(k) and w2(k) are surely not
independant.
'''
print('For part (f) interpretation 2, Pinf does not converge, Jp -> infinity')
```

```
For part (f) interpretation 1, Jp ~= 2.459m
When delta = 0.1, Jp ~= 8.331m
When delta = 0.01, Jp ~= 76.915m
When delta = 0.001, Jp ~= 768.454m
When delta = 0.0001, Jp ~= 7684.425m
When delta = 1e-05, Jp ~= 76844.191m
For part (f) interpretation 2, Pinf does not converge, Jp -> infinity
```

In [14]:
```
'''
(g) Retain the original sensor, but modify the system design by making
it more aerodynamic and thereby reducing the effect of aerodynamic
disturbances, so that stdev = 1m/s.
'''

stdev, H = 1, np.array([[1, 0]])
V, W = noise_orig_sys()
print('For part (g) Jp ~= ' + repr(round(compute_jp(), 3)) + 'm')
```

```
For part (g) Jp ~= 2.091m
```

In [ ]: