Practical SVD System ID

Consider the following discrete-time system:

$$x[i+1] = ax[i] + b_1 u_1[i] + b_2 u_2[i]$$

for some unknown system parameters a, b_1, b_2. We can estimate the unknown parameters by setting up a system of linear equations and solving it, namely

$$D\vec{p} = \vec{s}$$

where $\vec{p} = \begin{bmatrix} a \\ b_1 \\ b_2 \end{bmatrix}$ contain our unknown parameters. Previously, we discussed that if $u_2[i] = \alpha u_1[i]$ for some constant $\alpha \in \mathbb{R}$, then we cannot use our traditional least squares approach for performing system ID. Since $D \in \mathbb{R}^{m \times n}$ would not have full column rank in this case, we have to resort to other methods for estimating $\vec{p}$. From the previous homework, we know how to apply the SVD to solve this problem assuming $\text{rank}(D) < \min(m, n)$. Let us apply this approach here.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(1)
```

Below, we define some inputs which we will use to identify the system. Notice that we set $u_2[i] = 2u_1[i]$.

In [2]:
```
u1_inp = np.array([1, -1, 2, 3, 2, 2, 1, -2, 5])
u2_inp = 2*u1_inp
x0 = 0
```

Next, we will generate data by applying inputs to our system and observing the states. Our actual system parameters are $a = 0.5, b_1 = -2, b_2 = 3$. Let's see how close our solution gets to these actual parameters.

In [3]:
```
def generate_data(x0, u1_inp, u2_inp):
    xs = [x0]
    for i in range(len(u1_inp)):
        x_next = 0.5*xs[-1] - 2*u1_inp[i] + 3*u2_inp[i] + np.random.norr
        xs.append(x_next)
    return np.array(xs)
```

In [4]:
```
xs = generate_data(x0, u1_inp, u2_inp)
```

In [5]:
```
xs = np.expand_dims(xs, axis=1)
u1_inp = np.expand_dims(u1_inp, axis=1)
u2_inp = np.expand_dims(u2_inp, axis=1)
```

First, define our D matrix and $\vec{s}$ vector using the generated data. The numpy array `xs` has data arranged like $\begin{bmatrix} x[0] \\ x[1] \\ \vdots \\ x[9] \end{bmatrix}$, the numpy array `u1_inp` has data arranged like $\begin{bmatrix} u_1[0] \\ u_1[1] \\ \vdots \\ u_1[8] \end{bmatrix}$, and the numpy array `u2_inp` has data arranged like $\begin{bmatrix} u_2[0] \\ u_2[1] \\ \vdots \\ u_2[8] \end{bmatrix}$

HINT: You may find the following function useful:
https://numpy.org/doc/stable/reference/generated/numpy.hstack.html (https://numpy.org/doc/stable/reference/generated/numpy.hstack.html)

In [14]:
```
D = np.hstack((xs[0:9], u1_inp, u2_inp))
s = xs[1:]
```

We expect D to have dimensions 9×3. **Use the cell below to confirm that. Then, compute the rank of D.** Make sure it matches with what you expect for our choice of inputs.

HINT: You may find the following function useful:
https://numpy.org/doc/stable/reference/generated/numpy.linalg.matrix_rank.html (https://numpy.org/doc/stable/reference/generated/numpy.linalg.matrix_rank.html)

In [16]:
```
m, n = D.shape
r = np.linalg.matrix_rank(D)
print(f'Dimensions of D: {m} by {n}')
print('Rank of D:', r)
```

```
Dimensions of D: 9 by 3
Rank of D: 2
```

Next, write code to compute the SVD of D. Let U be denoted with `U`, Σ be denoted with `Sig`, and V be denoted with `V`.

HINT: You may find the following functions useful:
https://numpy.org/doc/stable/reference/generated/numpy.linalg.svd.html (https://numpy.org/doc/stable/reference/generated/numpy.linalg.svd.html), https://numpy.org/doc/stable/reference/generated/numpy.diag.html (https://numpy.org/doc/stable/reference/generated/numpy.diag.html). Take careful note of the outputs of `np.linalg.svd`.

In [17]:
```
U, Sig, V = np.linalg.svd(D)
```

Using the previous cell, write code to find U_r (denoted `U_r`) and Σ_r (denoted `Sig_r`).

In [40]:
```
U_r = U[:, :r]
Sig_r = np.diag(Sig)[:r, :r]
```

From the previous homework, we know that

$$\vec{p}_\star = V \begin{bmatrix} \Sigma_r^{-1} U_r^\top \vec{s} \\ \vec{0}_{n-r} \end{bmatrix}$$

is the lowest-norm choice of $\vec{p}$ that minimizes $\|D\vec{p} - \vec{s}\|$. **Write this solution in code (where $\vec{p}_\star$ is denoted by `p_star`). Are the values of a, b_1, b_2 what you expect? Why or why not?**

HINT: You may find the following functions useful:
https://numpy.org/doc/stable/reference/generated/numpy.linalg.inv.html (https://numpy.org/doc/stable/reference/generated/numpy.linalg.inv.html),
https://numpy.org/doc/stable/reference/generated/numpy.vstack.html (https://numpy.org/doc/stable/reference/generated/numpy.vstack.html)

In [47]:
```
i = np.linalg.inv(Sig_r) @ U_r.T @ s
z = np.vstack((i, np.zeros(n-r)))
p_star = V.T @ z
print("optimal p", p_star)
print("norm of p", np.linalg.norm(p_star))
```

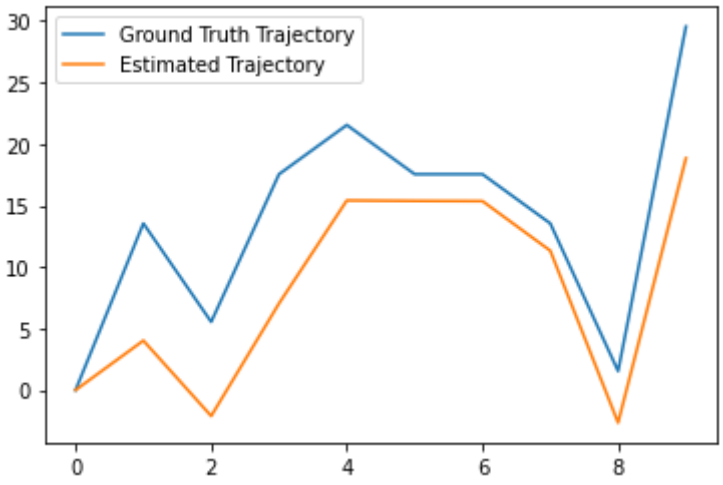
```
optimal p [[0.475805  ]
 [0.80410947]
 [1.60821895]]
norm of p 1.8599329605723245
```

We can see how closely our estimated system's trajectory follows the true system's trajectory:

In [48]:
```
def plot_trajectory_vs_ground_truth(p_star, u1_inp, u2_inp, x0):
    xs_gt = [x0]
    xs_est = [x0]
    a_star = p_star[0,0]
    b_1_star = p_star[1,0]
    b_2_star = p_star[2,0]
    for i in range(len(u1_inp)):
        x_next = 0.5*xs[-1] - 2*u1_inp[i] + 3*u2_inp[i]
        xs_gt.append(x_next)
        x_next_est = a_star*xs_est[-1] + b_1_star*u1_inp[i] + b_2_star*u
        xs_est.append(x_next_est)
    plt.plot(xs_gt, label="Ground Truth Trajectory")
    plt.plot(xs_est, label="Estimated Trajectory")
    plt.legend(loc="best")
```

In [49]:
```
plot_trajectory_vs_ground_truth(p_star, u1_inp, u2_inp, 0)
```

```
/srv/conda/envs/notebook/lib/python3.9/site-packages/numpy/core/shape
_base.py:65: VisibleDeprecationWarning: Creating an ndarray from ragg
ed nested sequences (which is a list-or-tuple of lists-or-tuples-or n
darrays with different lengths or shapes) is deprecated. If you meant
to do this, you must specify 'dtype=object' when creating the ndarra
y.
  ary = asanyarray(ary)
```



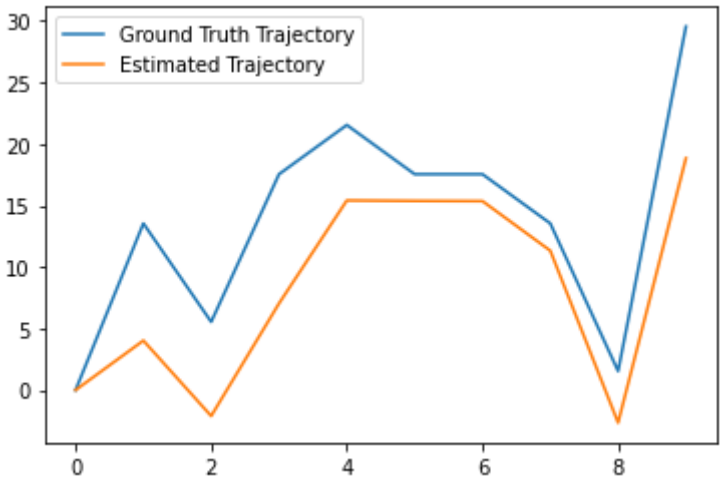
We can try replacing the vector of zeros ($\vec{0}_{n-k}$) with another arbitrary vector $\vec{k} \in \mathbb{R}^{n-k}$ (denoted `random_k`). That is, we can find

$$\tilde{\vec{p}} = V \begin{bmatrix} \Sigma_r^{-1} U_r^\top \vec{s} \\ \vec{k} \end{bmatrix}$$

Will using the values of a, b_1, b_2 in $\tilde{\vec{p}}$ instead of $\vec{p}$ change the trajectory of the system for the given inputs? Why or why not? Fill in the code below to verify your answer.

Based on the results below, using a random vector does not change the trajectory of the system as the estimated trajectory follows a similar path as before when the vector was a 0 vector. The random k vector serves as a noise factor which does not have an effect on the trajectory.

In [53]:
```
random_k = np.random.randn(n-r, 1)
p_tilde = V.T @ (np.vstack((i, random_k)))
plot_trajectory_vs_ground_truth(p_tilde, u1_inp, u2_inp, 0)
```



We can observe this phenomenon more generally by testing several random vectors in place of $\vec{0}_{n-k}$. First, we generate a large set of arbitrary $\vec{k}$ vectors. We can compute the highest loss (i.e., the highest value of $\|D\tilde{\vec{p}} - \vec{s}\|$) among all randomly generated $\vec{k}$, and we can look at the lowest norm (i.e., the lowest value of $\|\tilde{\vec{p}}\|$) among all randomly generated $\vec{k}$. **What do you expect for each of these quantities (i.e., for $\|D\tilde{\vec{p}} - \vec{s}\|$ and $\|\tilde{\vec{p}}\|$)? Fill in the code below to verify your answer.**

You can expect that the varied k vector impact the loss of the quantity as it contributes to noise.

In [59]:
```
def insert_random_vectors(num_trials=100, rand_k_range=5):
    highest_loss = -float('inf')
    lowest_norm = float('inf')
    for _ in range(num_trials):
        random_k = rand_k_range*np.random.randn(n-r, 1)
        p_test = V.T @ (np.vstack((i, random_k)))
        highest_loss = max(highest_loss, np.linalg.norm(D @ p_test - s)
        lowest_norm = min(lowest_norm, np.linalg.norm(p_test))
    return highest_loss, lowest_norm
```

In [60]:
```
highest_loss, lowest_norm = insert_random_vectors()
print("highest loss", highest_loss)
print("lowest norm", lowest_norm)
```

```
highest loss 3.7040159978284977
lowest norm 1.8599570050155712
```

In [ ]:
```

```