EE 263 Homework 3

Luis A. Perez

Memory of a linear dynamical, time-invariant system

Solution:

(a) Let us first consider how we might check if a valid impose response of fixed size M exists. The first thing to note is that the covolution operator given in the problem statent can actually be written as a linear system:

$$\bar{y} = Ah$$

where $\bar{y} \in \mathbb{R}^{T-M}, h \in \mathbb{R}^M$ and $A \in \mathbb{R}^{(T-M)\times M}$. Note that we remove the first $\{y_1, \cdots, y_M\}$. In fact, we can use the matrix A as defined below:

$$A = \begin{bmatrix} u_M & u_{M-1} & u_{M-2} & \cdots & u_1 \\ u_{M+1} & u_M & u_{M-1} & \cdots & u_2 \\ u_{M+2} & u_{M+1} & u_M & \cdots & u_2 \\ \vdots & \vdots & \vdots & \ddots & \\ u_{T-2} & u_{T-3} & u_{T-4} & \cdots & u_{T-M-1} \\ u_{T-1} & u_{T-2} & u_{T-3} & \cdots & u_{T-M} \end{bmatrix}$$

$$y_{-1} = \begin{bmatrix} y_{M+1} \\ \vdots \\ y_T \end{bmatrix}$$

$$h = \begin{bmatrix} h_1 \\ \vdots \\ h_M \end{bmatrix}$$

We can see by inspection above that Ah performs the needed convolutions betweens u and h to obtain $\bar{y}$, by the properties of matrix multiplication (eg, to obtain y_i, we compute the dot product of the $i-M$-th row of A with h, which is exactly what our convolution dictates, and for $i \leq M$, the convolution is not fully defined so we ignore them).

In the problem statement, we're given the fact that $T > 2M$, so this means that there will always be at least $T - M > 2M - M = M$ rows in A, meaning it will always be a tall and skinny matrix.

As such, we can use the pseudo inverse to find the closests solution for h, computing:

$$\bar{h} = (A^T A)^{-1} A^T \bar{y}$$

Finally, once we've computed this h, we can re-compute that $\bar{y}$ given our inputs forming A, and see if this equals our what we started with. In other words, we have that:

$$||A\bar{h} - \bar{y}|| \leq \epsilon \implies M \text{ is a valid value}$$

(ϵ is needed to deal with floating point imprecision)

The above gives us a way to check if M is sufficient. To finallize our method, we simply iterate over the possible values of M from $M = 1, \cdots \frac{T}{2} - 1$ in order until we find a valid value.

(b) Applying the process we described above, we find that $M = 7$ is the smallest value that works. We also have:

$$h = \begin{bmatrix} 0.63 \\ 0.27 \\ 0.02 \\ 0.37 \\ 0.96 \\ 0.95 \\ 0.46 \end{bmatrix}$$

Norm preserving implies orthonormal columns

Solution: We show that if $A \in \mathbb{R}^{m\times n}$ satifies $||Ax|| = ||x||$ for all $x \in \mathbb{R}^n$, then A has orthonormal columns.

This boils down to showing that $A^TA = I$, which can be broken into two parts. Let A_i be the i-th column of A. Then we need to show that $A_i^TA_i = 1$, and that $A_i^TA_j = 0$ for $i \neq j$. Let us consider the former first:

$$\begin{aligned} A_i^TA_i &= (Ae_i)^T(Ae_i) && \text{(Multiplying } A \text{ by } e_i \text{ extracts the } i\text{-th column)} \\ &= ||Ae_i||^2 && \text{(Definition of dot product treating } Ae_i \text{ as a vector)} \\ &= ||e_i||^2 && \text{(Multiplication by } A \text{ preserves the norm)} \\ &= 1 \end{aligned}$$

Let us now consider the latter case. We follow the hint:

$$\begin{aligned} ||A(e_i + e_j)||^2 &= ||e_i + e_j|| && (A \text{ preserves norm}) \\ &= 2 \end{aligned}$$

However, we also have that:

$$\begin{aligned} ||A(e_i + e_j)||^2 &= ||Ae_i + Ae_j||^2 && \text{(Linearity of } A) \\ &= ||Ae_i||^2 + ||Ae_j||^2 + 2(Ae_i)^T(Ae_j) && \text{(Definition of vector norm)} \\ &= 2 + 2(Ae_i)^T(Ae_j) && \text{(Using previous results)} \end{aligned}$$

Putting the two results together, we must have that:

$$2 + 2(Ae_i)^T(Ae_j) = 2 \implies 2(Ae_i)^T(Ae_j) = 0 \implies (Ae_i)^T(Ae_j) = 0 \implies A_i^TA_j = 0$$

As such, we've now shown that $A^TA = I$.

Sensor integrity monitor

Solution: We need to find $k \in \mathbb{R}^{k \times m}$ such that:

- $By = 0$ for any y which is consistent.
- $By \neq 0$ for any y which is inconsistent.

The above can be summarized as follow. Requirement (1) means that given $y \in \mathbf{Img}(A)$, we must have $By = 0$. This means that $\mathbf{Img}(A) \subseteq \mathbf{Ker}(B)$.

The second requirement specifies that for any $y \notin \mathbf{Img}(A)$, we must have $By \neq 0$. Phrase another way, this says that if $By = 0$, it must be in the $\mathbf{Img}(A)$. So we have $\mathbf{Ker}(B) \subseteq \mathbf{Img}(A)$.

As such, we're really just tring to construct a matrix B whose kernal is equal to the span of of the columns of A. This is actually a rather straight-forward process as long as we recall the following theorem, which holds for *any* matrix C:

$$\mathbf{Ker}(C^T) = \mathbf{Img}(C)^\perp$$

Thinking about it step-by-step, if the Kernel of B is equal to the Image of A, it must mean that the Image of B is equal to the complement of the Image of A. Using the formula presented abouve, the completement of the Image of A is the null-space of of A^T.

So simply put, we just need to (1) find the $M - N$ vectors spanning the nullspace of A^T and take these to be the rows of B. This will guarantee that the Kernel of B will equal the image of A.

Doing exactly the process described above gives us $B \in \mathbb{R}^{2 \times 5}$ as follows:

$$B = \begin{bmatrix} 0 & -5.47259193 & 2 & 8.47259193 & 1 \\ 0 & 6.97831636 & 6.39848918 & 2.61941742 & 3.19924459 \end{bmatrix}$$

Coin Collector Robot

Solution:

(a) To find the coordinates of the robot at time t, we have to consider what the effect of the constant force is. Given that the robot is unit mass, we have $a_i = f_i$ (acceleration is equal to the force).

Let us now consider the effect of the force f_i for one second (from $t = 2i - 2$ to $t = 2i - 1$) on the final position:

$$\Delta x_i^1 = \underbrace{\frac{1}{2} f_i}_{\text{Change caused by acceleration over 1 second}} + \underbrace{f_i(t - (2i - 1))}_{\text{Change caused by change in velocity}}$$

As such, for all f_i that are applied over the full two second period, we will have:

$$\Delta x_i = \underbrace{2f_i}_{\text{Change caused by acceleration over 2 seconds}} + \underbrace{2f_i(t - 2i)}_{\text{Change caused by change in velocity}}$$

The final position at time t, let's call it x_t, will then be given by $x_t = x_0 + \Delta x$ where Δx (the total change) can be computed with the following:

$$\Delta x = \begin{cases} \sum_{i=1}^{\frac{t}{2}} \Delta x_i & t \text{ is even} \\ \Delta x^1_{\frac{t+1}{2}} + \sum_{i=1}^{\frac{t-1}{2}} \Delta x_i & t \text{ is odd} \end{cases}$$

Furthermore, we know that $x_0 = 0$. As such, the postion at time t is given by (x_t, y_t) where

$$y_t = t$$
$$x_t = \begin{cases} \sum_{i=1}^{\frac{t}{2}} 2f_i + 2f_i(t - 2i) & t \text{ is even} \\ \frac{1}{2} f_{\frac{t+1}{2}} + \sum_{i=1}^{\frac{t-1}{2}} 2f_i + 2f_i(t - 2i) & t \text{ is odd} \end{cases}$$

We note that the above are all linear relations, so if we were so inclined, we could re-write the above as simply $x_t = c_t^T f$ for some vector $c_t \in \mathbb{R}^n$ and $f = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix}$. In

fact, we have:

$$c_t = \begin{bmatrix} 2+2(t-2) \\ 2+2(t-4) \\ 2+2(t-6) \\ \vdots \\ 2+2(6) \\ 2+2(4) \\ 2+2(2) \\ 2 \\ 0 \\ 0 \\ \vdots \end{bmatrix} \in \mathbb{R}^n \qquad (t \text{ is even})$$

$$c_t = \begin{bmatrix} 2+2(t-2) \\ 2+2(t-4) \\ 2+2(t-6) \\ \vdots \\ 2+2(5) \\ 2+2(3) \\ 2+2(1) \\ \frac{1}{2} \\ 0 \\ 0 \\ \vdots \end{bmatrix} \in \mathbb{R}^n \qquad (t \text{ is odd})$$

(b) We are now given a sequence $(x_1, y_1), \cdots, (x_{2n}, y_{2n})$ of $2n$ coins, and we want to determine whether the robot can collect them. For simplicitly, we assume that $y_i = i$ (all we require is that $i \neq j \implies y_i \neq y_j$, since if this were not the case, it's immediate that the coins cannot be collected, but the problem can be presented more simply if we have $y_i = i$).

From our work in part (a), we know that the robot can collect the coins if the following all hold:

$$\begin{aligned} x_1 &= c_1^T f \\ x_2 &= c_2^T f \\ &\vdots \\ x_{n-1} c_{n-1}^T f \\ x_n &= c_n^T f \end{aligned}$$

which we can write as the linear system $x = Af$ where $x \in \mathbb{R}^{2n}, f \in \mathbb{R}^n$ and

$A \in \mathbb{R}^{2n \times n}$ as per the below:

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{2n} \end{bmatrix} = \begin{bmatrix} — & c_1^T & — \\ — & c_2^T & — \\ & \vdots & \\ — & c_{2n}^T & — \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix}$$

As such, we have an overdetermined system of equations. As such, we can compute the closests fit $\hat{f}$:

$$\hat{f} = (A^T A)^{-1} A^T x$$

and check to see if this $\hat{f}$ satisfies our problem statement. In other words, to check feasibility, we need to check if:

$$||A\hat{f} - x|| < \varepsilon$$

forms some sufficiently small ε (due to floating point imprecision).

(c) See attached code. We show that for the data from "robot_coin_collector.m", not all the coins can be collected under this framework.

(d) If we now that only one of the coins cannot be collected, we can simply, we can simply attempt the same process above but with A_{-i} and x_{-i} (eg, we remove the i-th row from A and remove the i-th element from x. This then imposes no restrictions on the robot for time i in the x-position, which relaxes our system.

If we check for all of $i = 1, \cdots, 2n$, we will be able to determine if the robot can collect all but one of the coins (and can even identify the coin).

(e) See attached code where the process described above is implemented.

It turns out that the 7-th coin (1-indexed) is the coin which cannot be collected. More specifically, the bad coin is $(x_7, y_7) = (13.0, 7)$.

In order to collect the remaning $2n - 1$ coins, the forces applied must be:

$$f = \begin{bmatrix} 1 \\ -4 \\ 7 \\ -10 \\ 20 \\ -35 \end{bmatrix}$$

We plot the coin and the robot locations in Figure 1.

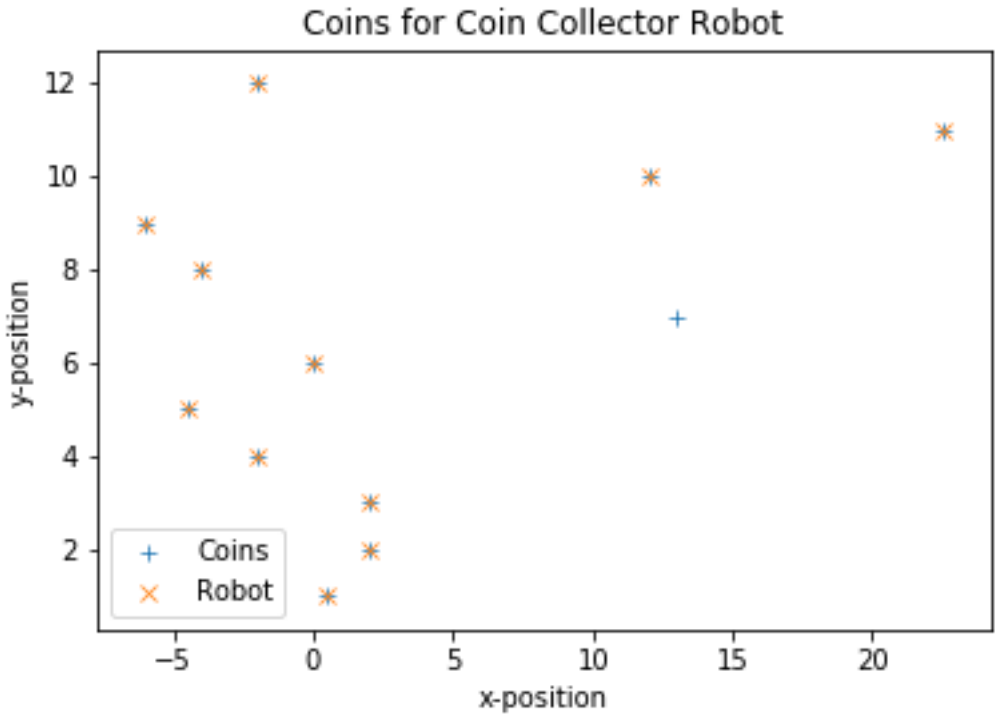


Figure 1: Location of coins and robot with the applied forces f

Solving linear equations via QR Factorization

Solution: The problem statement essentially boils down to solving the system $Rx = z$ where $R \in \mathbb{R}^{n\times n}$ is upper triangular and non-singular (eg, $R_{ii} \neq 0, \forall i$) and $z, x \in \mathbb{R}^n$.

The algoritm proceeds as follows:

1. First, let us find x_n. Let $R_i \in \mathbb{R}^{1\times n}$ be the i-th row of R as a row vector. We have:

$$\begin{aligned} z_n &= R_n x \\ &= \sum_{i=1}^{n} R_{ni} x_i \\ &= R_{nn} x_n && \text{(Since } R \text{ is upper triangular, } R_{ni} = 0 \text{ for } i < n) \\ \implies x_n &= \frac{z_n}{R_{nn}} && \text{(Note this is valid since } R_{nn} \neq 0) \end{aligned}$$

2. Using a similar process, we can now find x_{n-1}. We have:

$$\begin{aligned} z_{n-1} &= R_{n-1}x \\ &= \sum_{i=1}^{n} R_{n-1,i}x_i \\ &= R_{n-1,n-1}x_{n-1} + R_{n-1,n}x_n \\ &\qquad\qquad \text{(Since } R \text{ is upper triangular, } R_{n-1,i} = 0 \text{ for } i < n-1) \\ \implies x_{n-1} &= \frac{z_n - R_{n-1,n}x_n}{R_{n-1,n}} \qquad \text{(Note this is valid since } R_{n-1,n} \neq 0) \end{aligned}$$

While the equation above looks more complicated, we note that all of the variables in the RHS are known (specifically, we found the value of x_n previously).

3. We simply continue the process from above. Let us assume that we've found the values x_k for all $k > i$. Then to compute x_i (the next value), we have:

$$\begin{aligned} z_i &= R_i x \\ &= \sum_{j=1}^{n} R_{ij}x_j \\ &= \sum_{j=i}^{n} R_{ij}x_j \qquad \text{(Since } R \text{ is upper triangular, } R_{ij} = 0 \text{ for } j < i) \\ \implies x_i &= \frac{z_i - \sum_{j=i+1}^{n} R_{ij}x_j}{R_{ii}} \qquad \text{(Note this is valid since } R_{ii} \neq 0) \end{aligned}$$

We know $z_i, R_{ij}, \forall j$, and by our assumption, we know $x_k, \forall k > i$, so we know all the variables in the LHS. As such, it's relatively straight-forwrad to compute x_i.

As such, we can see that the $x_n, x_{n-1}, \cdots, x_1$ produced by our algorithm are valid solutions to the system $z = Rx$. Furthermore, the algorithm cannot fail since R is non-singular (all diagonal entries are non-zero).

Quadratic extrapolation of a time series, using least-squares fit

Solution:

(a) In the problem statement, we are given that $\hat{z}(t+1) = f(t+1)$ where $f(\tau) = a_2\tau^2 + a_1\tau + a_0$ is the quadratic function which minimizes:

$$J = \sum_{\tau=t-9}^{t} (z(\tau) - f(\tau))^2$$

We can rewrite the above in matrix form as follows:

$$J = ||Fa - z||^2$$

where

$$z = \begin{bmatrix} z(t) \\ z(t-1) \\ \vdots \\ z(t-9) \end{bmatrix}$$

$$a = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

$$F = \begin{bmatrix} 1 & t & t^2 \\ 1 & t-1 & (t-1)^2 \\ \vdots & \vdots & \vdots \\ 1 & t-9 & (t-9)^2 \end{bmatrix}$$

If we want to solve for a, we can immediately arrive at the solution (based on what we've covered in lecture):

$$a = (F^TF)^{-1}F^Tz$$

With the above, we can write $\hat{z}(t+1)$ as follows:

$$\begin{aligned} \hat{z}(t+1) &= f(t+1) \\ &= a_1 + a_2(t+1) + a_3(t+1)^2 \\ &= \begin{bmatrix} 1 & t+1 & (t+1) \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \\ &= \begin{bmatrix} 1 & t+1 & (t+1) \end{bmatrix} (F^TF)^{-1}F^Tz \end{aligned}$$

As such, we can see that for a given value of t, we have:

$$\hat{z}(t+1) = \begin{bmatrix} 1 & t+1 & (t+1)^2 \end{bmatrix} (F^T F)^{-1} F^T \begin{bmatrix} z(t) \\ z(t-1) \\ \vdots \\ z(t-9) \end{bmatrix}$$

which implies that we should have:

$$c = \begin{bmatrix} 1 & t+1 & (t+1)^2 \end{bmatrix} (F^T F)^{-1} F^T$$

which at first glance appears to depend on the value of t. However, using code to solve the system symbolically, we see that c does not vary with time, and is infact given by the constant matrix:

$$\begin{aligned} c &= \begin{bmatrix} \frac{9}{10} & \frac{1}{2} & \frac{11}{60} & -\frac{1}{20} & -\frac{1}{5} & -\frac{4}{15} & -\frac{1}{4} & -\frac{3}{20} & \frac{1}{30} & \frac{3}{10} \end{bmatrix} \\ &= \begin{bmatrix} 0.9 & 0.5 & 0.18333333 & -0.05 & -0.2 & -0.26666667 & -0.25 & -0.15 & 0.03333333 & 0.3 \end{bmatrix} \end{aligned}$$

(b) See attached code for details on how we calculate the RMS. The final result is an RMS of:

$$0.050960852617736675$$

(c) We plot the requested serieses in Figure 2. There is no previous problem using inteporlation on the previous three values (as far as we are aware), so we cannot comment on the RMS.

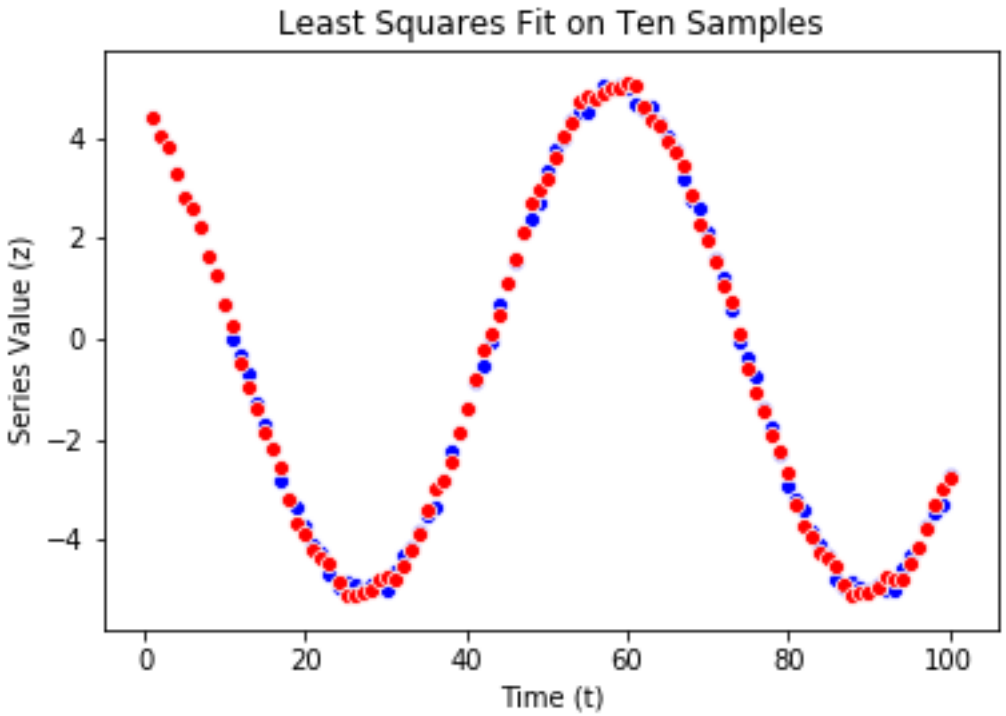


Figure 2: In red is the estimated values $\hat{z}$ using the model described in part (a) of the homework. In blue is the real values of the series, z.

Householder reflection

Solution:

(a) To show that Q is orthogonal, we just need to show that $Q^TQ = I$.

$$\begin{aligned} Q^TQ &= (I - 2uu^T)^T(I - 2uu^T) \\ &= I + 4(uu^T)^T(uu^T) - 2(uu^T)^T - 2uu^T \\ &= I + 4(uu^T)(uu^T) - 2uu^T - 2uu^T \qquad ((uv)^T = v^Tu^T) \\ &= I + 4(uu^T)(uu^T) - 4uu^T \\ &= I \end{aligned}$$

We first

show that $Qu = -u$.

$$\begin{aligned} Qu &= (I - 2uu^T)u \\ &= u - 2uu^T u \\ &= u - 2u && \text{(Using } u^T u = 1) \\ &= -u \end{aligned}$$

We now consider Qv where $u^T v = 0$. We have:

$$\begin{aligned} Qv &= (I - 2uu^T)v \\ &= v - 2uu^T v \\ &= v - 2u(0) && (u^T v = 0) \\ &= v \end{aligned}$$

We now show that $\det Q = -1$.

$$\begin{aligned} \det Q &= \det \left(I - 2uu^T\right) \\ &= \det \left(I + (-2u)u^T\right) \\ &= \det (I) + u^T \text{adj } (I)(-2u) && \text{(By the matrix determinant lemma)} \\ &= 1 - 2u^T u \\ &= 1 - 2 \\ &= -1 \end{aligned}$$

As the hint dictates, we try $u = \frac{x + \alpha e_1}{||x + \alpha e_1||}$. Using this u, we can compute Qx and then pick α such that Qx is parallel to e_1.

$$\begin{aligned} Qx &= (I - 2uu^T)x \\ &= x - 2uu^T x \\ &= x - 2\left(\frac{x + \alpha e_1}{||x + \alpha e_1||}\right)\left(\frac{x + \alpha e_1}{||x + \alpha e_1||}\right)^T x \\ &= x - 2\frac{1}{||x + \alpha e_1||^2}\left(xx^T + \alpha^2 e_1 e_1^T + \alpha x e_1^T + \alpha e_1 x^T\right) x \\ &= x - 2\frac{1}{||x + \alpha e_1||^2}\left(xx^T x + \alpha^2 e_1 e_1^T x + \alpha x e_1^T x + \alpha e_1 x^T x\right) \\ &= x - 2\frac{1}{||x + \alpha e_1||^2}(||x||^2 x + \alpha^2 x_1 e_1 + \alpha x_1 x + \alpha ||x||^2 e_1) \end{aligned}$$

Not finished.

Vector space multiple access

Solution: Not required.

HW3

July 14, 2019

1 HW3

author: Luis Perez
email: luis0@stanford.edu

2 Imports

[1]:
```
import numpy as np
import pprint
from scipy import linalg
import seaborn as sns
```

2.1 Problem 1: Memory of a linear, time-invariant system

2.1.1 Part (b)

[2]:
```
# Number of samples
T = 100
# The input
u = np.array([
    0.7700,
    0.0600,
    0.0100,
    0.8000,
    0.6600,
    0.9600,
    0.0300,
    0.3200,
    0.9700,
    0.9400,
    0.3600,
    0.4100,
    0.7000,
    0.1700,
    0.1500,
    0.5000,
```

```
    0.9600,
    0.7300,
    0.0800,
    0.9000,
    0.6800,
    0.1100,
    0.8500,
    0.1900,
    0.1600,
    0.8900,
    0.0700,
    0.8300,
    0.2600,
    0.9700,
    0.9100,
    0.3800,
    0.4100,
    0.7000,
    0.3200,
    0.7200,
    0.8600,
    0.1700,
    0.2600,
    0.1900,
    0.9100,
    0.9800,
    0.7900,
    0.3300,
    0.4300,
    0.8200,
    0.3800,
    0.5300,
    0.9000,
    0.8300,
    0.7900,
    0.3600,
    0.0300,
    0.1700,
    0.8500,
    0.6800,
    0.2100,
    0.6400,
    0.5000,
    0.5100,
    0.6000,
    0.1900,
    0.7400,
```

```
    0.9800,
    0.2100,
    0.1900,
    0.3300,
    0.1900,
    0.3900,
    0.5900,
    0.8900,
    0.8500,
    0.5200,
    0.3700,
    0.9600,
    0.8400,
    0.4900,
    0.5400,
    0.8700,
    0.0500,
    0.0500,
    0.3800,
    0.4000,
    0.7800,
    0.4500,
    0.4100,
    0.5300,
    0.8900,
    0.9800,
    0.9300,
    0.1900,
    0.9900,
    0.5800,
    0.6500,
    0.2600,
    0.0900,
    0.0400,
    0.1700,
    0.6800,
    0.3900
])

# The output.
y = np.array([
    1.9663,
    2.5611,
    2.0898,
    1.2280,
    1.2092,
    1.4403,
```

```
    1.6056,
    1.0081,
    1.2782,
    2.4515,
    2.7882,
    1.8628,
    1.5105,
    2.1557,
    2.4086,
    1.9909,
    1.7859,
    2.0328,
    1.8014,
    1.0792,
    1.6591,
    2.4087,
    2.1433,
    2.1237,
    1.8788,
    1.7544,
    2.0878,
    1.5912,
    1.6593,
    1.4437,
    1.8176,
    2.1338,
    1.8734,
    1.8083,
    2.4560,
    2.4541,
    2.3812,
    2.1748,
    1.7084,
    1.6541,
    1.8287,
    2.3495,
    2.2745,
    1.6575,
    1.2851,
    1.9121,
    2.8243,
    2.6995,
    2.1301,
    2.1108,
    2.2646,
    2.2775,
    2.0367,
```

```
    1.9813,
    2.3103,
    2.6761,
    2.1503,
    1.1300,
    1.1453,
    1.7349,
    2.0853,
    2.0011,
    1.6036,
    1.9028,
    2.3020,
    1.7725,
    1.4568,
    1.7929,
    2.0215,
    1.8469,
    1.4356,
    1.3920,
    1.5152,
    1.4999,
    1.7441,
    2.6239,
    2.9211,
    2.4077,
    2.0847,
    2.5268,
    2.3471,
    1.9722,
    1.9461,
    1.9477,
    1.7484,
    1.1384,
    0.9787,
    1.5102,
    2.1821,
    2.3770,
    2.2443,
    1.8250,
    2.6027,
    3.0107,
    2.8894,
    2.2339,
    1.9132,
    1.8799,
    1.8463,
    1.6423
```

```
])
assert y.shape[0] == u.shape[0]
assert u.shape[0] == T
```

[3]:
```
def isValidMemory(M, u, y, eps=1e-4):
    """Uses the process described in the homework to check if
    M is a valid memory value for the given inputs u and outputs y.
    """
    # Creates the A matrix described in the handout.
    T = y.shape[0]
    A = linalg.toeplitz(c=u[M-1:T-1], r=np.flip(u[:M]))
    assert u.shape[0] == T
    assert A.shape == (T-M, M)

    # Find the closests hs that work.
    ybar = y[M:]
    hbar = np.dot(np.linalg.inv(np.dot(A.T, A)), np.dot(A.T, ybar))
    return np.linalg.norm(ybar - np.dot(A, hbar)) < eps, hbar
```

[4]:
```
def findSmallestValidM(u,y):
    T = u.shape[0]
    for m in range(1, T // 2):
        isValid, candidate = isValidMemory(m,u,y)
        import pdb
        pdb.set_trace
        if isValid:
            return m, candidate
    assert False
```

[5]:
```
smallestM, weights = findSmallestValidM(u,y)
print("The memory of our given data is %s." % smallestM)
pprint.pprint(weights)
```

```
The memory of our given data is 7.
array([0.63, 0.27, 0.02, 0.37, 0.96, 0.95, 0.46])
```

2.2 Problem 3: Sensor integrity monitor

[6]:
```
# Copied from homework assignment.
A = np.array([
    [1, 2, 3],
    [1, -1, -2],
    [-2, 1, 3],
    [1, -1, -2],
    [1, 1, 0]
])
```

[7]:
```
def findBMatrix(A, eps=1e-3):
    B = linalg.null_space(A.T).T
```

```
    B[np.abs(B) < eps] = 0
    B /= np.abs(B[B != 0]).min()
    return B
```

[8]:
```
B = findBMatrix(A)
```

[9]:
```
def checkMatrixIsIntegrityMonitor(measureMatrix, integrityMonitor, eps=1e-3):
    M, N = measureMatrix.shape
    # Generate all unit vectors in input space.
    inputBasis = np.identity(N)

    # Assert all consistent values will output zero by monitor.
    for i in range(N):
        measurement = np.dot(measureMatrix, inputBasis[:,i])
        assert np.linalg.norm(np.dot(integrityMonitor, measurement)) < eps

        # Inconsistent by adding random noise (high likelihood)
        badMeasurement = measurement + np.random.rand(M)
        assert np.linalg.norm(np.dot(integrityMonitor, badMeasurement)) > eps
```

[10]:
```
checkMatrixIsIntegrityMonitor(A, B)
```

[11]:
```
B
```

[11]:
```
array([[ 0.        , -5.47259193,  2.        ,  8.47259193,  1.        ],
       [ 0.        ,  6.97831636,  6.39848918,  2.61941742,  3.19924459]])
```

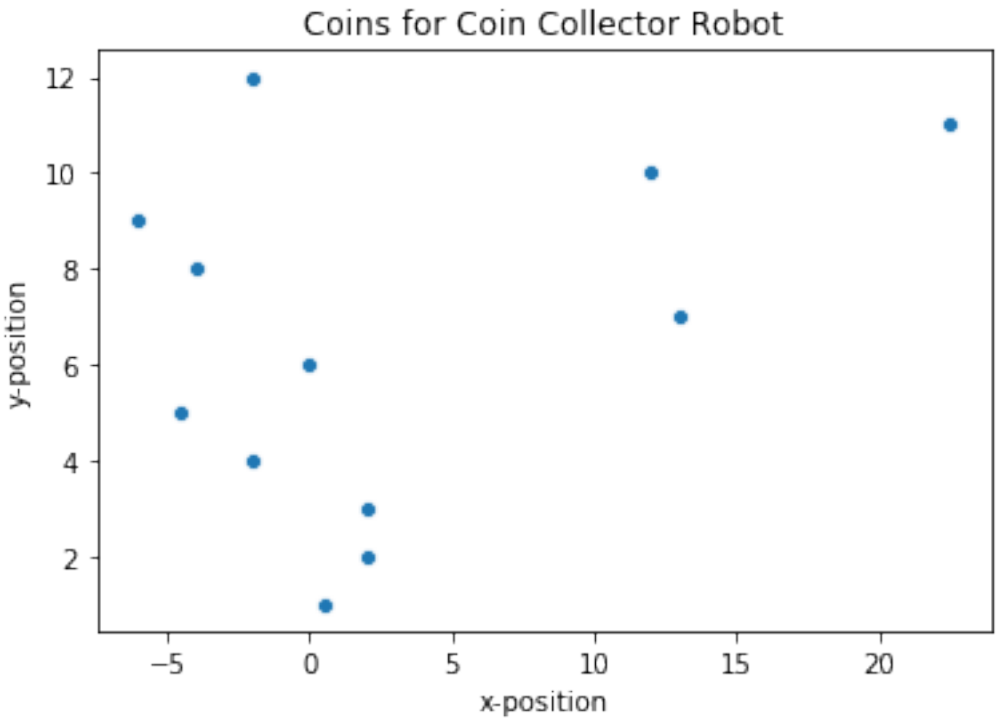
2.3 Problem 4: Coin collector robot

2.3.1 Part (c)

[12]:
```
n=6;
x = [
    0.5000,
    2.0000,
    2.0000,
   -2.0000,
   -4.5000,
         0,
   13.0000,
   -4.0000,
   -6.0000,
   12.0000,
   22.5000,
   -2.0000]
```

[13]:
```
# Plot of the coins.
ax = sns.scatterplot(x=x, y=range(1,2*n + 1))
ax.set_title("Coins for Coin Collector Robot")
ax.set(xlabel='x-position', ylabel='y-position')
```

[13]:
```
[Text(0, 0.5, 'y-position'), Text(0.5, 0, 'x-position')]
```



[14]:
```
def generateAMatrix(n, missingRowIndex=None):
    """
    Generates A as specified in the handout with
    missingRowIndex missing (if set).
    """
    def generateCRow(t):
        c = np.zeros(n)
        for i in range(t // 2):
                c[i] = 2 + 2*(t - 2*(i+1))
        # If t is odd, we need to add the final push.
        if t % 2 != 0:
            c[t // 2] = 0.5
        return c
    return np.stack([generateCRow(i+1) for i in range(2*n)
                     if missingRowIndex is None or missingRowIndex != i])
```

[15]:
```
def isCollectible(x, n, missingRowIndex=None, eps=1e-3):
    """Returns true if the coins with the given x-coordinates are␣
 ↪collectible"""
    A = generateAMatrix(n, missingRowIndex)
    fhat = np.dot(np.linalg.inv(np.dot(A.T, A)), np.dot(A.T,x))
    if linalg.norm(np.dot(A, fhat) - x) < eps:
```

```
            return fhat
        return None
```

[16]:
```
if isCollectible(x, n):
    print("The coins can be collected.")
else:
    print("The coins cannot be collected.")
```

```
The coins cannot be collected.
```

[17]:
```
def collectAllButOne(x, n):
    """Tries to collect all coins except 1.

    Returns the forces and the index of the bad coin.
    """
    indexes = []
    forces = []
    coins = []
    for i in range(2*n):
        xMissing = np.delete(x, i)
        maybeFhat = isCollectible(xMissing, n, missingRowIndex=i)
        if maybeFhat is not None:
            indexes.append(i)
            forces.append(maybeFhat)
            coins.append((x[i], i+1))
    assert len(indexes) == 1
    assert len(forces) == 1
    assert len(coins) == 1
    return indexes[0], coins[0], forces[0],
```

[18]:
```
badCoinIndex, badCoin, forcesToCollectAllButOne = collectAllButOne(x, n)
```

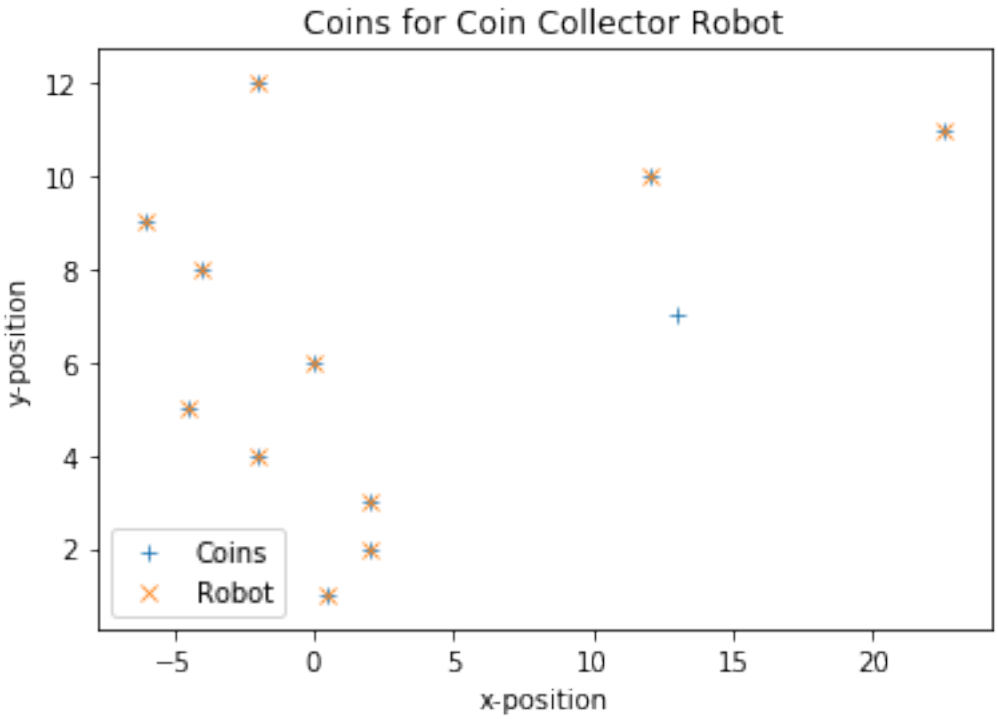
[19]:
```
print("The %s-th coin (1-indexed) is the coin which cannot be collected." %␣
 ↪(badCoinIndex + 1))
print("The bad coin is located at (%s,%s)." % (badCoin))
```

```
The 7-th coin (1-indexed) is the coin which cannot be collected.
The bad coin is located at (13.0,7).
```

[20]:
```
validA = generateAMatrix(n, missingRowIndex=badCoinIndex)
robotXPositions = np.dot(validA, forcesToCollectAllButOne)

# Generate the plot for everything (except the missing coin)
ax = sns.scatterplot(x=x, y=range(1,2*n + 1), marker="+", label="Coins")
ax = sns.scatterplot(x=robotXPositions,
                     y=list(range(1,badCoinIndex + 1)) +␣
 ↪list(range(badCoinIndex + 2, 2*n + 1)),
                     marker='x', label="Robot")
```

```
ax.set_title("Coins for Coin Collector Robot")
ax.set(xlabel='x-position', ylabel='y-position')
ax.get_figure().savefig("robot_collector")
```



2.4 Problem 6: Quadratic Extrapolation of Time Series using Least-Squares Fit

2.4.1 Part (a)

[23]:
```
def computeCMatrix(t):
    """
    For a given value of t, computes the C matrix
    as defined in the handout.
    """
    PAST_SAMPLES = 10
    F1 = [1] * PAST_SAMPLES
    F2 = [t - i for i in range(PAST_SAMPLES)]
    F3 = [(t - i)**2 for i in range(PAST_SAMPLES)]
    F = np.stack((F1, F2, F3), axis=1)
    t = np.array([1, t + 1, (t + 1)**2])
    C = np.dot(np.dot(t, np.linalg.inv(np.dot(F.T, F))), F.T)
    return C
```

[24]:
```
def verifiyCIsTimeInvariant(eps=1e-4):
    for t in range(1000):
```

```
        C1 = computeCMatrix(t + 1)
        C2 = computeCMatrix(t + 2)
        if not linalg.norm(C1 - C2) < eps:
            print(linalg.norm(C1 - C2))
    return computeCMatrix(1)
```

[25]:
```
computeCMatrix(1)
```

[25]:
```
array([ 0.9       ,  0.5       ,  0.18333333, -0.05      , -0.2       ,
       -0.26666667, -0.25      , -0.15      ,  0.03333333,  0.3       ])
```

[26]:
```
computeCMatrix(1000)
```

[26]:
```
array([ 0.90000038,  0.4999995 ,  0.18333217, -0.0500016 , -0.20000182,
       -0.26666848, -0.2500016 , -0.15000115,  0.03333284,  0.30000039])
```

2.4.2 Part (b)

[27]:
```
def generateTrueTimeSeries():
    # Generates the vector [1, ..., 1000]
    t = np.array(range(1, 1001))
    z = 5 * np.sin(t / 10 + 2) + 0.1 * np.sin(t) + 0.1 * np.sin(2*t - 5)
    return z
```

[28]:
```
def computeExtrapolationFromValues(values):
    assert len(values) == 10
    c = computeCMatrix(1)
    for _ in range(11, 1001):
        newValue = np.dot(c, np.flip(values[len(values)-10:]))
        values = np.append(values, newValue)
    return values

def computeEstimatesUsingTrueValues(values):
    c = computeCMatrix(1)
    estimates = values[:10].tolist()
    for i in range(10, 1000):
        estimates.append(np.dot(c, np.flip(values[i-10:i])))
    return np.array(estimates)
```

[29]:
```
def getRMSE(true, predictions):
    return np.sqrt(np.sum((true[10:] - predictions[10:])**2) / np.sum(true[10:
 ↪]**2))
```

[30]:
```
true = generateTrueTimeSeries()
predsUsingOnlyInitial = computeExtrapolationFromValues(true[:10])
predsUsingRealValues = computeEstimatesUsingTrueValues(true)
extrapolateRMSE = getRMSE(true, predsUsingOnlyInitial)
cheatRMSE = getRMSE(true, predsUsingRealValues)
```

[31]:
```
cheatRMSE
```

[31]:
```
0.050960852617736675
```

[32]:
```
MAX = 100
ax = sns.scatterplot(x=range(1,MAX+1), y=true[:MAX], color='blue')
ax = sns.scatterplot(x=range(1,MAX+1), y=predsUsingRealValues[:MAX],␣
 ↪color='red')
ax.set_title("Least Squares Fit on Ten Samples")
ax.set(xlabel='Time (t)', ylabel='Series Value (z)')
ax.get_figure().savefig('least_squares_fit_series')
```