Q1

The data was first extracted from the .dat file and seperated type. the "t" variable holds the discrete time stamps while the "y" variable holds the measured output at said timestamps. Note that the max parameters was increased from 5 to 6 to hihglight the behaviour of $\hat{\sigma}$ later on in part 3 of question 1.

In [68]:
```python
import pandas as pd
import numpy as np
param_max = 6 # Largest number of parameters

data = np.loadtxt('dataHw1.dat')
t = data[:,0].copy()
y = data[:,1].copy()
```

A look-up table containing the regressors was generated. The width of this table (# of columns) is determined by the value of the "param_max" variable declaired in the previous cell. The code code cell used for estimating phi will make use of this table by slicing the required section (e.g. only the first 2 columns will be used when estimating 2 parameters).

In [69]:
```python
phi = np.stack([(t - 1)**n for n in range(param_max)], axis=1) # generation of phi look-up table
pd.DataFrame(phi, columns=[f't^{i}' for i in range(param_max)]) # Prints look-up table bellow
```

Out[69]:

	t^0	t^1	t^2	t^3	t^4	t^5
0	1.0	0.0	0.0	0.0	0.0	0.0
1	1.0	1.0	1.0	1.0	1.0	1.0
2	1.0	2.0	4.0	8.0	16.0	32.0
3	1.0	3.0	9.0	27.0	81.0	243.0
4	1.0	4.0	16.0	64.0	256.0	1024.0
5	1.0	5.0	25.0	125.0	625.0	3125.0
6	1.0	6.0	36.0	216.0	1296.0	7776.0
7	1.0	7.0	49.0	343.0	2401.0	16807.0
8	1.0	8.0	64.0	512.0	4096.0	32768.0
9	1.0	9.0	81.0	729.0	6561.0	59049.0
10	1.0	10.0	100.0	1000.0	10000.0	100000.0
11	1.0	11.0	121.0	1331.0	14641.0	161051.0
12	1.0	12.0	144.0	1728.0	20736.0	248832.0
13	1.0	13.0	169.0	2197.0	28561.0	371293.0
14	1.0	14.0	196.0	2744.0	38416.0	537824.0

In the cell below, the least square estimates for parameters ranging from 1 to 5 (or param_max) are calculated with each iteration of the for loop

In [70]:
```python
import pandas as pd
from numpy.linalg import inv

theta_hat = [] # list for storing theta_hat
loss = [] # list for storing the loss functions
for i in range(0,param_max):
    phi_temp = phi[:, 0:i+1] # the "phi" look-up table is sliced as required for each iteration
    theta_temp = inv(phi_temp.T@phi_temp)@phi_temp.T@y # temporary storage of theta_hat estimate
    err = (y - phi_temp@theta_temp) # difference between measured output and estimated output
    theta_hat.append(np.append(theta_temp, [0]*(param_max - i - 1))) # estimated theta_hat for each
                                                                     # iteration are stored here
    loss.append(err@err/2) # Loss function for each iteration are stored here

# Theta_hats for parameter counts ranging from one to 5 (or value of "param_max") are
# packaged into a dataframe for presentation in table format
df = pd.DataFrame(np.vstack(theta_hat),
                  index=[i for i in range(1,param_max+1)],
                  columns=[f'Theta{i}' for i in range(1,param_max+1)])
df['Loss'] = loss # Loss column is added on far right side of table
df
```

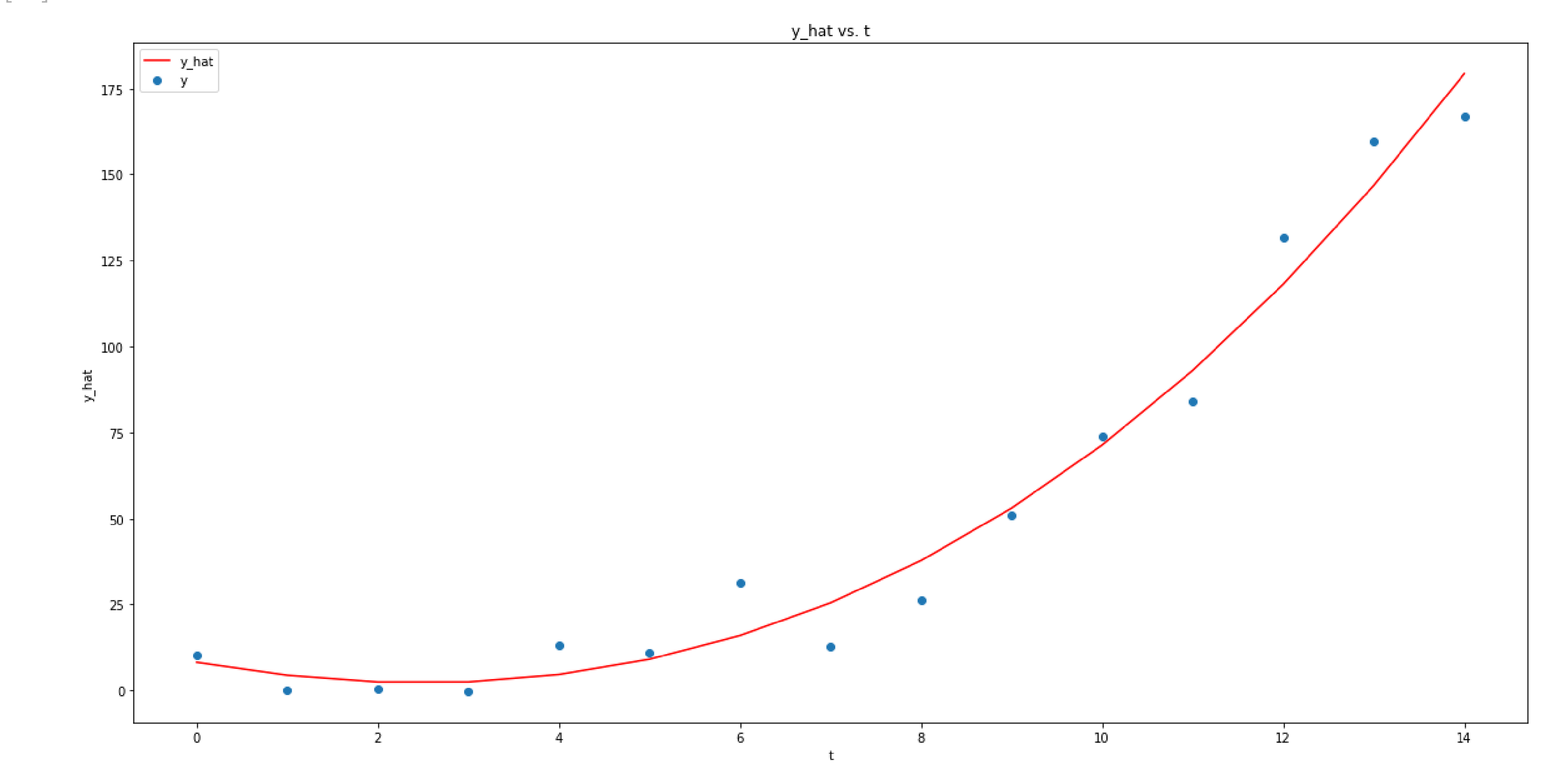
Out[70]:

	Theta1	Theta2	Theta3	Theta4	Theta5	Theta6	Loss
1	51.435013	0.000000	0.000000	0.000000	0.000000	0.000000	24103.092840
2	-31.106229	11.791606	0.000000	0.000000	0.000000	0.000000	4637.216740
3	11.150564	-7.711529	1.393081	0.000000	0.000000	0.000000	634.251379
4	8.137128	-4.576673	0.813574	0.027596	0.000000	0.000000	612.440363
5	4.234310	3.497340	-1.992023	0.346007	-0.011372	0.000000	563.290183
6	11.388026	-25.060590	14.203890	-2.876500	0.251265	-0.007504	293.073810

When observing the data in the table above, one can see that the loss function seems to stabilise around 3 parameters with relativley small differences for 4 parameters and up. Although the loss function seems to plateau, the value at which it levels out is still quite high. This is due to the large value of sigma (i.e. sigma = 11). The code in the cell below extracts the row coeresponding to 3 parameters (third row) and is post matrix multiplied by the "phi" look-up table. The row was padded with zeros for parameters 4 and 5 and therfore, can be directly multiplied by the entire look-up table when calculating the estimated y_hat vector.

In [71]:
```python
import matplotlib.pyplot as mpl
num_param_best = 4 # number of parameters to be used

# extraction of row with 3 parameters excluding the "loss" column
theta_hat_true = df.loc[num_param_best, df.columns != 'Loss']
y_hat = phi@theta_hat_true # calculation of estimated y_hat vector

mpl.scatter((t - 1), y) # plot measured output y
mpl.plot((t - 1), y_hat, 'r') # plot estimated output y_hat
mpl.xlabel('t')
mpl.ylabel('y_hat')
mpl.title('y_hat vs. t')
mpl.legend(labels=['y_hat', 'y'])
```

Out[71]: `<matplotlib.legend.Legend at 0x16b587232b0>`



Clearly, when analysing the graph above (y_hat vs. t), one can see that the curve representing the estimated outputs "y_hat" folows the trend of the measured values very closely.

In [72]:
```python
num_param_best = 3
sigma = np.sqrt(2*df['Loss']/(np.array([15]*param_max) - df.index))
pd.DataFrame(sigma, columns=['Sigma'])
```

Out[72]:

	Sigma
1	58.679630
2	26.709885
3	10.281467
4	10.552383
5	10.614049
6	8.070162

The equation for the estimation of the standard deviation is

$\hat{\sigma} = \sqrt{\frac{2V(\hat{\theta},t)}{t-n}}$

where

$V(\hat{\theta},t) = \frac{1}{2}\sum_{i=1}^{t}(y(i) - \phi^T\theta)^2$.

Since $y(i)$ is the measured output, it can be further exapnded to

$y(i) = y(i)_{actual} + e(i)$

where

$y(i)_{actual}$ is the actual output and $e(i)$ is the iid noise. If this is substituted into the loss function, the equation becomes

$V(\hat{\theta},t) = \frac{1}{2}\sum_{i=1}^{t}(y(i)_{actual} + e(i) - \phi^T\theta)^2$

If we assume that $\phi^T\theta$ ($y(i)_{estimate}$) is so close to $y(i)_{actual}$, that $y(i)_{actual} - \phi^T\theta \approx 0$, the loss function becomes

$V(\hat{\theta},t) = \frac{1}{2}\sum_{i=1}^{t}e(i)^2$

Plugging this result back into the equation for estimating $\hat{\sigma}$ we get

$\hat{\sigma} = \sqrt{\frac{\sum_{i=1}^{t}e(i)^2}{t-n}}$

Since the mean of the noise is zero, we can conclude that the estimation for $\hat{\sigma}$ becomes the estimation of the standard variation of the noise. Therfore, if $\hat{\sigma} \approx \sigma$, then $\phi^T\theta$ must be very close to y_{actual}. When observing the table above, one can see that $\hat{\sigma}$ is closes to σ ($\sigma = 11$) when 5 parameters are being estimated and has a sharp drop for 6 parameters. It should be taken into consideration that 15 data points is not very much and the total distribution might not have completly taken the form of a gaussian distribution. For this reason, it would be prudent to use 4 parameters since it is in the middle of the three different parameters closes to 11.

Q.2

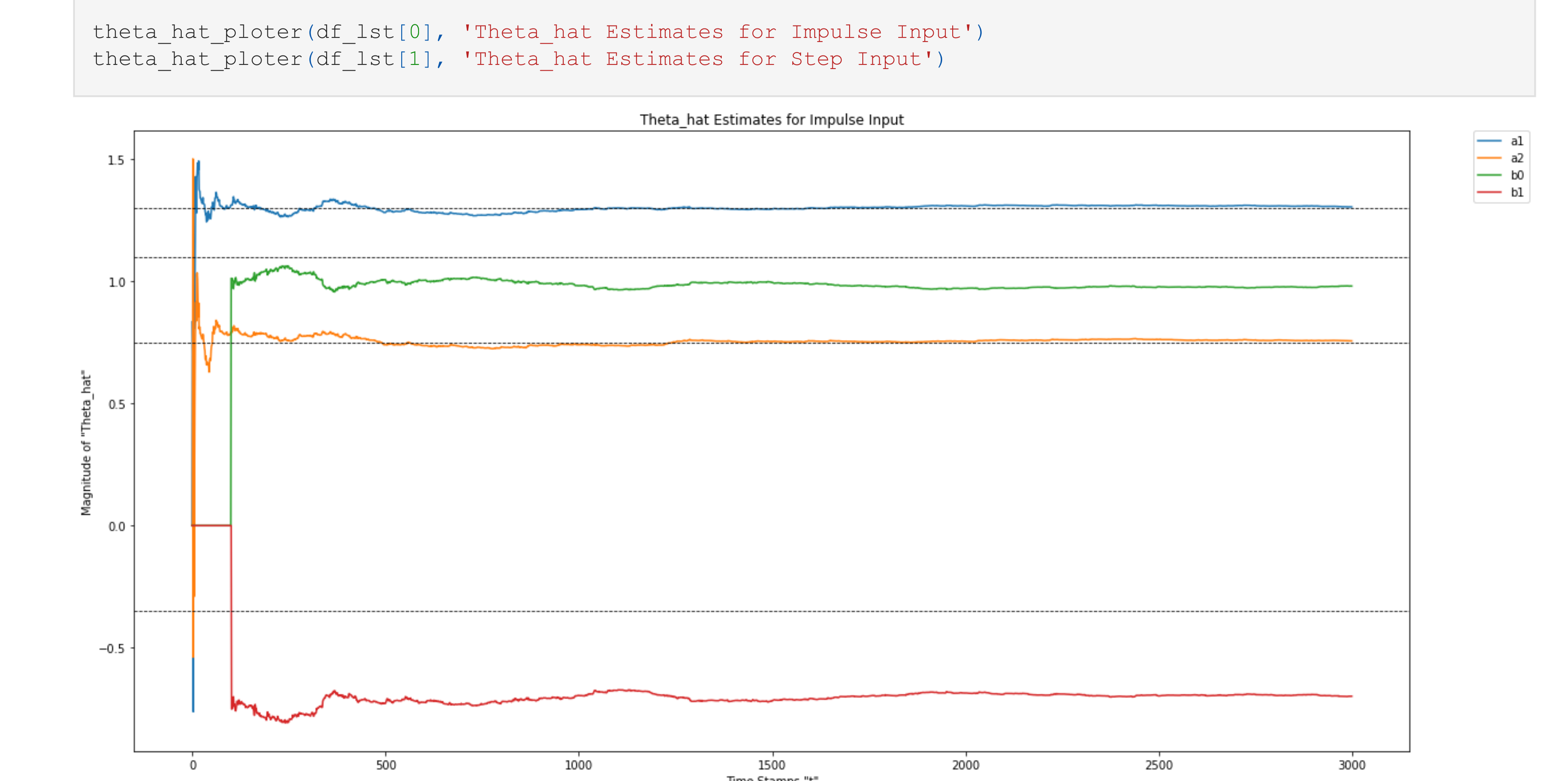
The cell below contains the initialisation of the experiment. All three inputs (i.e. $\delta(t-100)$, $u(t-100)$ and $sin(\frac{2\pi t}{5}) + cos(\frac{4\pi t}{5})$) are created and inserted into a list of three lists. Each individual list will be referenced during their respective iteration of the loops 2 cells below. Additionally, $\hat{\theta}$ and $y(t)$ are also initialised here.

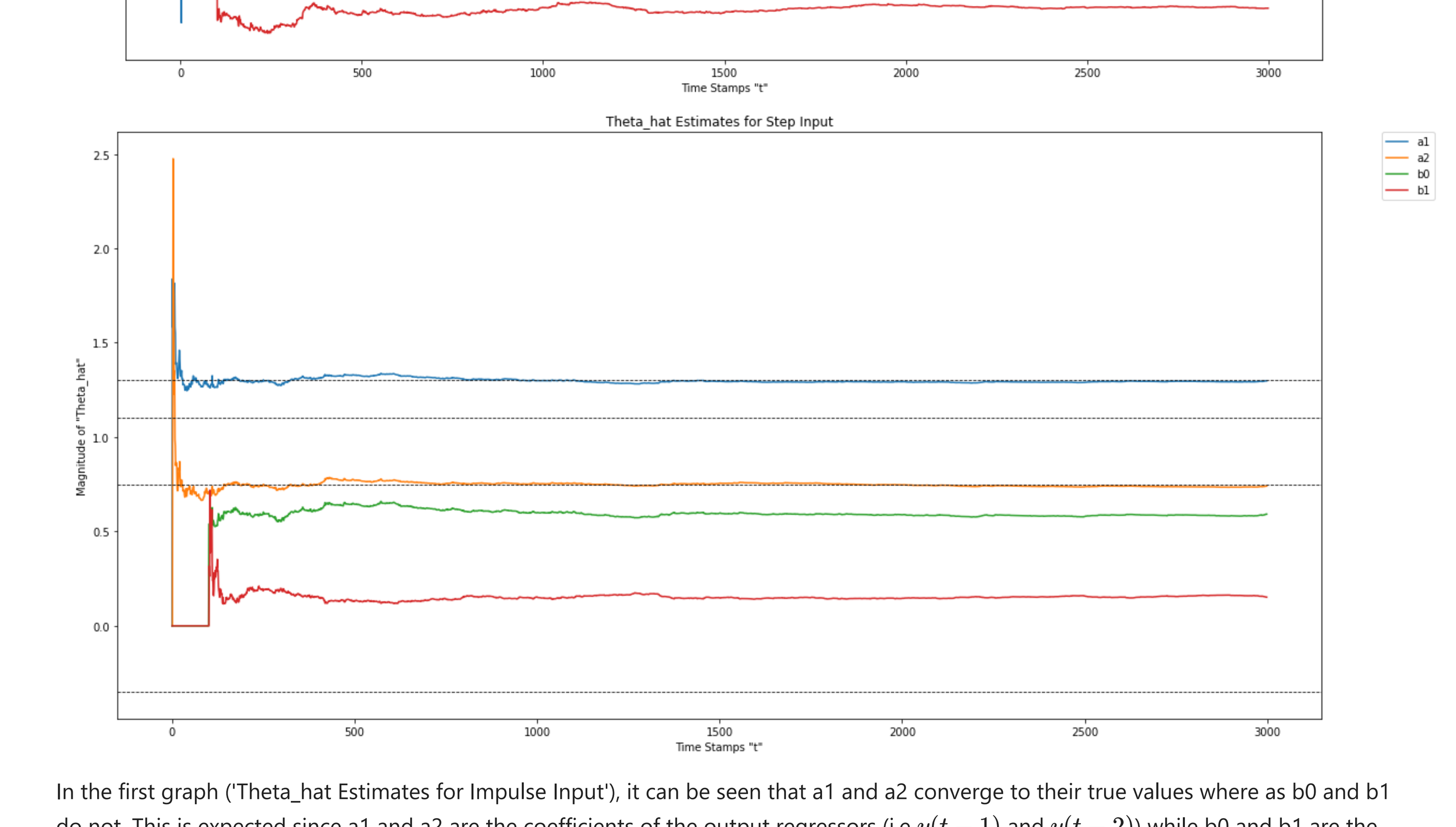
In [73]:
```python
import numpy as np
import pandas as pd
from math import cos, sin, pi
import scipy as spy
from numpy.linalg import inv
from numpy import sqrt
from scipy.signal import unit_impulse
import matplotlib.pyplot as mpl

sample_depth = 3000

a1 = 1.3
a2 = 0.75
b0 = 1.1
b1 = -0.35

theta0 = np.array([a1, a2, b0, b1])
p =100*np.identity(4) # starting P matrix

t = [i for i in range(sample_depth)]
u_t1 = unit_impulse(sample_depth, 100) # Creating impulse delta(t - 100)
u_t2 = np.zeros(sample_depth) # Creating unit step unit(t - 100)
u_t2[np.where(np.arange(0,sample_depth) >= 100)] = 1
u_t3 = np.array([sin(2*pi*t[i]/5) + cos(4*pi*t[i]/5) for i in t]) # Creating dual freq. sinusoid

u_t = np.stack([u_t1, u_t2, u_t3]) # impulse = u_t[0], step = u_t[1], dual freq. sinusoid = u_t3

sigma = 0.65
y0 = np.random.normal(0, sigma)
# creating a list of three lists which will be used to store the outputs of all three runs
y = [[y0] for i in range(len(u_t))]

theta_hat0 = np.reshape(np.array([0]*4), (-1,1)) # initial theta estimations will be 0
# creating a list of three lists which will be used to store the theta_hats of all three runs
theta_hat = [[theta_hat0] for i in range(len(u_t))]
```

The cell below contains two nested loops. The first loop determines which input vector will be used. The second loop, estimates the parameters recursively. During the first iteration, ϕ is set to $-y_{(j)(i-1)}, 0, u_{t(j)(i-1)}, 0$ since $-y_{(j)(i-2)}$ and $u_{t(j)(i-2)}$ do not exist yet. It should also be notted that the original equation given in the assingment was time shifted by 2 time units. This way, the equation relies on past values rather than future ones. Additionally, the final P matrix will be saved for each type of input. This will be used to calculate $\hat{\sigma_{b0}}$ and $\hat{\sigma_{b1}}$ in part 4 of this question.

In [74]:
```python
p_final = [] # used to derive sigma_hat_b0 and sigma_hat_b1 in part 4 of this question
for j in range(len(u_t)):
    p =100*np.identity(4) # starting P matrix
    for i in range(1,sample_depth):
        if (i == 1): # accounts for the lack of t-2 data on first iteration
            phi = np.array([-y[j][i-1], 0, u_t[j][i-1], 0])
        else:
            phi = np.array([-y[j][i-1], -y[j][i-2], u_t[j][i-1], u_t[j][i-2]])

        # changes phi's dimensions from (4,) to [4,1] enabling transpose operations
        phi = np.asarray(phi).reshape(-1,1)
        y[j].append(np.reshape(phi.T@theta0 + np.random.normal(0, sigma), ()))
        p = inv(inv(p) + phi@phi.T)
        k = p@phi
        theta_hat[j].append(theta_hat[j][i-1] + k*(y[j][i] - phi.T@theta_hat[j][i-1]))

        # The data collected with this "if" statement will be used in part 4 of Q2
        if (i == sample_depth - 1):
            p_final.append(p)

df_lst = [pd.DataFrame(np.asarray(theta_hat[i]).reshape(-1,4,),
                  columns=['a1', 'a2', 'b0', 'b1']) for i in range(len(u_t))]
```

In [75]:
```python
import matplotlib.pyplot as mpl
import seaborn as sns

mpl.rcParams['figure.figsize'] = [20, 10]

def theta_hat_ploter(df, title, line_width=0.8):
    graph = sns.lineplot(data=df, dashes=False)

    graph.axhline(y=a1, color='black', linestyle='--', linewidth=line_width, label='a1')
    graph.axhline(y=a2, color='black', linestyle='--', linewidth=line_width, label='a2')
    graph.axhline(y=b0, color='black', linestyle='--', linewidth=line_width, label='b0')
    graph.axhline(y=b1, color='black', linestyle='--', linewidth=line_width, label='b1')

    mpl.title(title)
    mpl.ylabel('Magnitude of "Theta_hat"')
    mpl.xlabel('Time Stamps "t"')
    mpl.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0, labels=df_lst[2].columns)
    mpl.show()

theta_hat_ploter(df_lst[0], 'Theta_hat Estimates for Impulse Input')
theta_hat_ploter(df_lst[1], 'Theta_hat Estimates for Step Input')
```





In the first graph ('Theta_hat Estimates for Impulse Input'), it can be seen that a1 and a2 converge to their true values where as b0 and b1 do not. This is expected since a1 and a2 are the coefficients of the output regressors (i.e.$y(t-1)$ and $y(t-2)$) while b0 and b1 are the coefficients of the inputs. Since the output is always stimulated by noise, the recursive least square estimates algorithum can pick-up and track changes in the $y(t-1)$ and $y(t-2)$ regressors and converge a1 and a2 to their true values. b0 and b1 on the other hand do not get stimulated until $t = 100$, the effects of which can be seen in the figure above where b0 and b1 are estimated to be zero (initial condition) until $t = 100$. The impulse dies immediatley after ($t = 101$). For this reason, b0 and b1 move but never converge onto their true values.

In the second graph ('Theta_hat Estimates for Step Input'), we can see a similar result where a1 and a2 converge to their true values while b0 and b1 do not. a1 and a2 converge for the same reasons they converged for an impulse input, however, b0 and b1 do not converge becasue a unit step is only sufficiently stimulating for one parameter. This can be explained qualitativley by thinking about the effects the input has on the output. At $t = 100$, the output will see a sudden jump, just like in the case for the impulse, however, unlike the impulse, the step continues to contribute to the output. Although the output feels the effects of the input for the rest of the experiment, there is no way of determining by how much a parameter regressor pair is contributing to the output. In other words, the solution is not unique.

Eq. 2.46 from the textbook states that parceval's theorem can be used to determine teh order of excitation of an input signal. The equation is

$\frac{1}{2\pi}\int_{-\pi}^{\pi} = |A(e^{j\omega})|^2\Phi(\omega)d\omega > 0$

In this equation, the amplitude $A(e^{j\omega})$ is squared and will therfore always be positive. The frequency content of the signal given by $\Phi(\omega)$ is the component that will determine at what frequency the solution is non-zero.

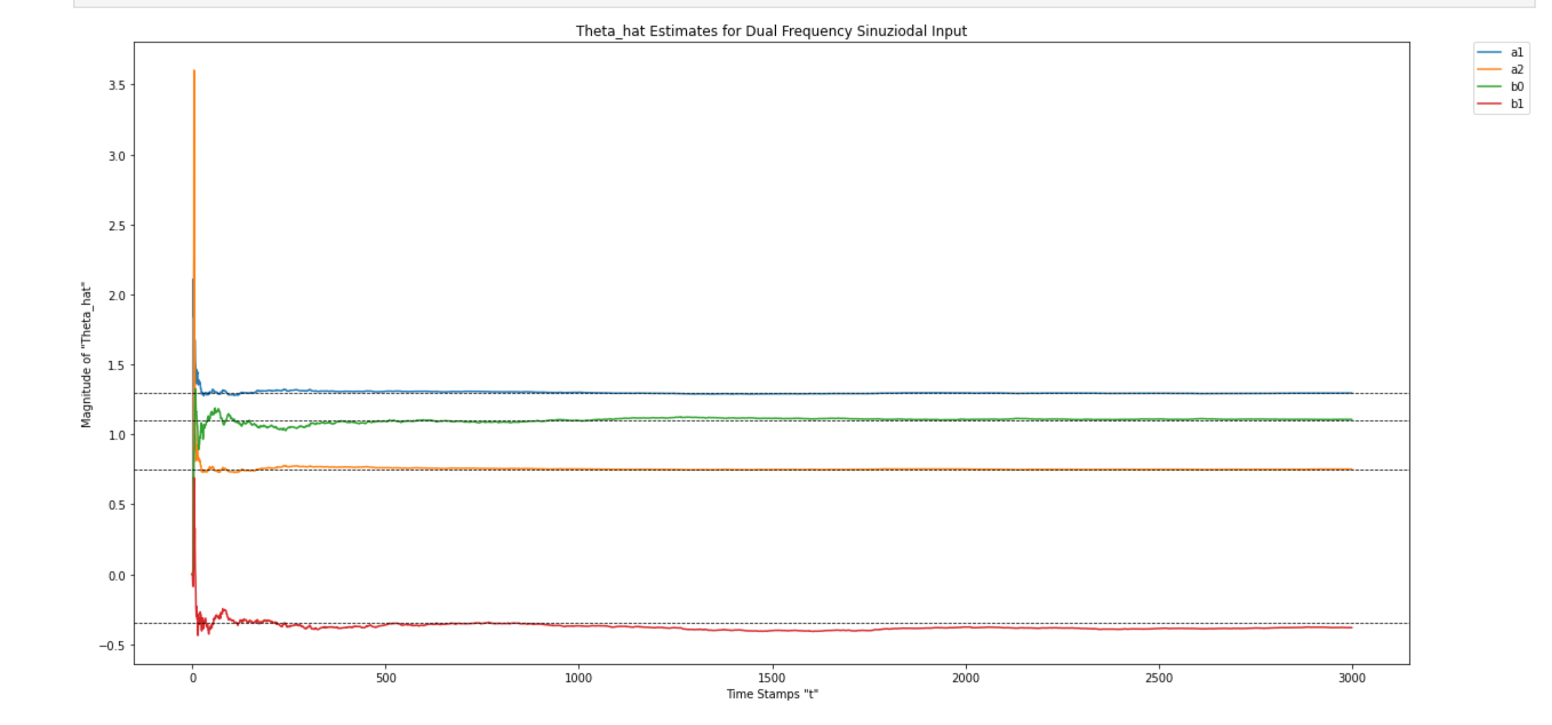
To determine the frequency content of the input signal, the fourier transform of the cosine and sin function will be derived.

F($cos(\omega t)$) = $\pi(\delta(\omega - \omega_0) + \delta(\omega + \omega_0))$

F($sin(\omega t)$) = $-j\pi(\delta(\omega - \omega_0) - \delta(\omega + \omega_0))$

Either of these functions contribute 2 degrees of excitation. Since $\omega_0 = \frac{4\pi}{5}$ for the cosine term and $\omega_0 = \frac{2\pi}{5}$ for the sine function, both components contribute to the order of excitation, that is, 4.

In [76]:
```python
theta_hat_ploter(df_lst[2], 'Theta_hat Estimates for Dual Frequency Sinuziodal Input')
```



In [77]:
```python
sigma_hat_b0 = [sigma*sqrt(p_final[i][2,2]) for i in range(len(u_t))]
sigma_hat_b1 = [sigma*sqrt(p_final[i][3,3]) for i in range(len(u_t))]

df = pd.DataFrame(np.stack([sigma_hat_b0, sigma_hat_b1]),
                  columns=['Impulse', 'Step', 'Dual Freq.'],
                  index=['sigma_hat_b0', 'sigma_hat_b1'])
df
```

Out[77]:

	Impulse	Step	Dual Freq.
sigma_hat_b0	0.646977	0.643915	0.014260
sigma_hat_b1	0.647113	0.643875	0.017144