ELG7113 Machine Learning for Adaptive and Intelligent Control Systems

Student: Derek Boase

Std Num: 300043860

e-mail: dboas065@uottawa.ca

assignment GitHub: git@github.com:derekboase/Adaptive_Control_Code.git

In [1]:
```
import matplotlib.pyplot as plt
import control as co
import numpy as np
import sympy as sp

from numpy import transpose as trans
from numpy import cos, sin, pi
from numpy.linalg import inv
```

Question 1

Question declarations:

In [2]:
```
r1, s0, s1, a01 = sp.symbols('r_1, s_0, s_1, a_0_1')
ao, a1, a2, b0 = sp.symbols('a_o,\hat{a}_1,\hat{a}_2,\hat{b}_0')
a, b, c, d, q, s, Ts = sp.symbols('a, b, c, d, q, s, T')
yddot, ydot = sp.symbols('\ddot{y},\dot{y}')
y, u, uc = sp.symbols('y, u, u_c')
a_true, b_true, c_true, d_true = (1, 1, 2, 1/2)
G1 = (b/(s + a)).subs(a, 1).subs(b, 1)
G2 = c/(s + d)
G_sym = b0/(s**2 + a1*s + a2)
G = sp.Mul(G1, G2)
Bm = 1
Am = s ** 2 + 2*s + 1
G = sp.collect(sp.expand(G), s)
B, A = sp.fraction(G)
B_plus = 1
B_minus = c
```

Part 1

Using the compatability conditions:

$$\mathrm{degA}_m = \mathrm{degA} = 2$$

$$\mathrm{degB}_m = \mathrm{degB} = 0$$

$$\mathrm{degA}_0 = \mathrm{degA} \text{ - } \mathrm{degB}^+ - 1 = 1$$

The causality condition gives,

$$\mathrm{degA}_m - \mathrm{degB}'_m \geq \mathrm{degA} - \mathrm{degB}^+$$

Given that

$$\mathrm{degA}_m = \mathrm{degA} = 2$$

and

$$\mathrm{degB}'_m = 0$$

then

$$\mathrm{degB}^+ = 0$$

With this we can find,

$$\mathrm{degA}_c = \mathrm{degA}_o + \mathrm{degA}_m + \mathrm{degB}^+ = 3$$

It is then found that,

$$\mathrm{degR} = \mathrm{degA}_c - \mathrm{degA} = 3 - 2 = 1$$

Using the minimum phase relationship,

$$\mathrm{degR} = \mathrm{degT} = \mathrm{degS} = 1$$

The control parameters may then be found using the Diophantine equation,

$$AR' + B^- S = A_0 A_m$$

$$(s^2 + (a+d)s + ad)(s + r_1) + bc(s_0 s + s_1) = (s + a_{o_1})(s^2 + 2s + 1)$$

In [3]:
```
R = s + r1
R_prime = R
S = s0*s + s1
Ao = (s + ao)
T = Ao*Bm/B_minus
```

$A_m = s^2 + 2s + 1$

$B_m = 1$

$R = r_1 + s$

$S = ss_0 + s_1$

$T = \frac{a_o + s}{c}$

In [4]:
```
dio_LHS = sp.collect(sp.expand(A*R_prime + B_minus*S), s)
dio_RHS = sp.collect(sp.expand(Ao*Am), s)
dio_poly = sp.Poly(sp.collect(sp.expand(dio_LHS - dio_RHS), s), s)

var = [r1, s0, s1]
control_vars = []
for idx in range(len(dio_poly.coeffs())):
    control_vars.append(sp.solve(dio_poly.coeffs()[idx], var[idx])[0])
r1c_sym = control_vars[0]
s0c_sym = control_vars[1]
s1c_sym = control_vars[2]
```

The above equation is described in the estimators by,

$r_1 = a_o - d + 1$

$s_0 = \frac{2a_o - dr_1 - d - r_1 + 1}{c}$

$s_1 = \frac{a_o - dr_1}{c}$

Part 2

The process is described by,

$p^2 y(t) + (d+1)py(t) + dy(t) = cu(t)$

Then we can define,

$y_f(s) = H_f(s)Y(s)$

and

$u_f(s) = H_f(s)U(s)$

where

$H_f(s) = \frac{1}{Am(s)} = \frac{1}{s^2 + 2s + 1}$

The model then becomes,

$\frac{p^2}{A_m(s)} y_f(t) = -(d+1)\frac{p}{A_m(s)} y_f(t) - \frac{d}{A_m(s)} y_f(t) + \frac{c}{A_m(s)} u_f(t)$

The RLS can be used on this model by defining,

$y_i(t) = \frac{p^i}{Am(p)} y(t)$

$u_i(t) = \frac{p^i}{Am(p)} u(t)$

With this choice of definition, we can reparameterize the model as,

$y_n(t) = \phi(t)^T \theta$

where,

$\phi(t) = \left[-y_{f1}(t) \;\; -y_{f0}(t) \;\; u_{f0}(t)\right]^T$

$\theta = \left[(d+1) \; d \; c\right]^T$

To begin to evaluate these parameters, the continuous time variables are estimated with the bilinear transformation. That is to say the following mapping is used,

$s = \frac{2(1-q^{-1})}{T_s(1+q^{-1})}$

where T_s is the sampling time

In [5]:
```
bilinear = 2*(1-q**(-1))/(Ts*(1+q**(-1)))


Hf1 = (s/Am).subs(s, bilinear)
Hf1 = sp.collect(sp.simplify(sp.expand(Hf1)), q)

Hf0 = (1/Am).subs(s, bilinear)
Hf0 = sp.collect(sp.simplify(sp.expand(Hf0)), q)

bilinear = 2*(1-q**(-1))/(Ts*(1+q**(-1)))

bf1, af1 = sp.fraction(Hf1)
af1_poly = sp.Poly(af1, q)
bf1_poly = sp.Poly(bf1, q)
af1_coeffs = []
bf1_coeffs = []
for n in range(len(af1_poly.coeffs())):
    af1_coeffs.append(sp.simplify(af1_poly.coeffs()[n]/af1_poly.coeffs()[0]))
for n in range(len(bf1_poly.coeffs())):
    bf1_coeffs.append(sp.simplify(bf1_poly.coeffs()[n]/af1_poly.coeffs()[0]))
bf1_coeffs.insert(1, 0)

Hf1 = (s/Am).subs(s, bilinear)
Hf1 = sp.collect(sp.simplify(sp.expand(Hf1)), q)

Hf0 = (1/Am).subs(s, bilinear)
Hf0 = sp.collect(sp.simplify(sp.expand(Hf0)), q)

bf0, af0 = sp.fraction(Hf0)
af0_poly = sp.Poly(af0, q)
bf0_poly = sp.Poly(bf0, q)
af0_coeffs = []
bf0_coeffs = []
for n in range(len(af0_poly.coeffs())):
    af0_coeffs.append(sp.simplify(af0_poly.coeffs()[n]/af0_poly.coeffs()[0]))
for n in range(len(bf0_poly.coeffs())):
    bf0_coeffs.append(sp.simplify(bf0_poly.coeffs()[n]/af0_poly.coeffs()[0]))
```

The coefficients for the denominator of the sampled filter H_{f1} are given as,

$\mathrm{den}\{H_{f1}\}\left[1, \frac{2(T-2)}{T+2}, \frac{T^2 - 4T + 4}{T^2 + 4T + 4}\right]$

The coefficients for the numerator of the sampled filter H_{f1} are given as,

$\mathrm{num}\{H_{f1}\}\left[\frac{2T}{T^2 + 4T + 4}, 0, -\frac{2T}{T^2 + 4T + 4}\right]$

The coefficients for the denominator of the sampled filtered H_{f0} are given as,

$\mathrm{den}\{H_{f0}\}\left[1, \frac{2(T-2)}{T+2}, \frac{T^2 - 4T + 4}{T^2 + 4T + 4}\right]$

The coefficients for the denominator of the sampled filtered H_{f0} are given as,

$\mathrm{num}\{H_{f0}\}\left[\frac{T^2}{T^2 + 4T + 4}, \frac{2T^2}{T^2 + 4T + 4}, \frac{T^2}{T^2 + 4T + 4}\right]$

Attention is taken to two results:

1. The denominators are both the same, as they come from the reference model parameters
2. The order of the coefficients goes from 0->2

The following difference equations are then found for y_{f_i} and u_{f_i}.

$y_{f1}(k) = \frac{2T}{T^2+4T+4} y(k) - \frac{2T}{T^2+4T+4} y(k-2) - \frac{2(T-2)}{T+2} y_{f1}(k-1) - \frac{T^2-4T+4}{T^2+4T+4} y_{f1}(k)$

$y_{f0}(k) = \frac{T^2}{T^2+4T+4} y(k) + \frac{2T^2}{T^2+4T+4} y(k-1) + \frac{T^2}{T^2+4T+4} y(k-2) - \frac{2(T-2)}{T+2} y_{f0}(k-1) - \frac{T^2-4T+4}{T^2+4T+4} y_0(k-2)$

$u_{f0}(k) = \frac{T^2}{T^2+4T+4} u(k) + \frac{2T^2}{T^2+4T+4} u(k-1) + \frac{T^2}{T^2+4T+4} u(k-2) - \frac{2(T-2)}{T+2} u_{f0}(k-1) - \frac{T^2-4T+4}{T^2+4T+4} u_0(k-2)$

Furthermore, rearranging the equation for $y_{f2}(k)$ we find,

$y(k) = \frac{T^2+4T+4}{4} y_{f2}(k) + \frac{T^2-4}{2} y_{f2}(k-1) + \frac{T^2-4T+4}{4} y_{f2}(k-2) + 2y(k-1) - y(k-2)$

To find the control action, $u(k)$, we look to the equation,

$u(t) = \frac{T}{R} u_c(t) - \frac{S}{R} y(t)$

The approach to find the discretized equation is the same as in the previous question so the details are neglected for readability,

$u(k) = -\frac{r_1 T - 2}{r_1 T + 2} u(k-1) - \frac{s_1 T + 2s_0}{r_1 T + 2} y(k) - \frac{s_1 T - 2s_0}{r_1 T + 2} y(k-1) + \frac{a_0 T + 2}{c(r_1 T + 2)} u_c(k) + \frac{a_0 - 2)}{c(r_1 T + 2)} u_c(k-1)$

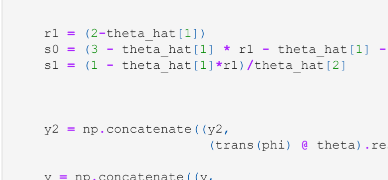
Part 3

In [6]:
```
def reference_signal(time=np.linspace(0, 1000, 1001)):
    sig = []
    rat = 2.0/40.0*pi
#     rat = 1.0/25.0*pi
    for t_lop in time:
        if sin(10*rat*t_lop) >= 0:
            sig.append(1)
        else:
            sig.append(0)
    return np.array(sig, dtype=float)
```

In [7]:
```
uc = reference_signal()
lam = 0.01
Ts = 0.1
end = 1000

# Assignmnent, works
am1 = -2*np.exp(-2)
am2 = np.exp(-4)
z_true, d_true, c_true = 1.5, 0.5, 2

# # In class example, works perfectly!
# am1 = -1.3205
# am2 = 0.4966
# a1_true, a2_true, b0_true, b1_true = -1.6065, 0.6065, 0.1065, 0.0902

theta = np.array([z_true, d_true, c_true ]).reshape(3,1)  # a1[0], a2[1], b0[2], b1[3]
```

In [8]:
```
theta_hat = np.array([0.1, 0.1, 0.1]).reshape(theta.shape)  # z d c
theta_lst = np.concatenate((theta_hat,
                            (theta_hat).reshape(3,1)), axis=1)

y = np.zeros(2)
u = np.zeros(2)


y2 = np.zeros(2)
y1 = np.zeros(2)
y0 = np.zeros(2)
u0 = np.zeros(2)

P = 10*np.eye(3)


for t in range(2, end):
    phi = np.array([-y1[t-1], -y0[t-1], u0[t-1]]).reshape(3,1)
    theta_hat = (theta_hat + Ts*(P @ phi)*(trans(phi) @ theta - trans(phi) @ theta_hat))
    theta_lst = np.concatenate((theta_lst,
                                theta_hat), axis=1)
    P = inv(inv(P) + phi@trans(phi))
    K = P@phi



    r1 = (2-theta_hat[1])
    s0 = (3 - theta_hat[1] * r1 - theta_hat[1] - r1)
    s1 = (1 - theta_hat[1]*r1)/theta_hat[2]



    y2 = np.concatenate((y2,
                         (trans(phi) @ theta).reshape(-1,)), axis=0)

    y = np.concatenate((y,
                        (1.1025*y2[t] - 1.995*y2[t-1] + 0.9025*y2[t-2] + 2*y[t-1] - y[t-2]).reshape(-1)), axis=
    u = np.concatenate((u,
                        (1/(Ts*r1 + 2)*(-(Ts*r1 - 2)*u[t-1] -(Ts*s1 + 2*s0)*y[t]) - (Ts*s1 -2*s0)* y[t-1] + (Ts

    y1 = np.concatenate((y1,
                         (0.04535*y[t] - 0.0435*y[t-2] + 1.8095*y1[t-1] - 0.8186*y1[t-2]).reshape(-1)), axis=0)
    y0 = np.concatenate((y0,
                         (0.00227*y[t] + 0.00454*y[t-1] + 0.00227*y[t-2] + 1.8095*y0[t-1] - 0.8186*y0[t-2]).resh
    u0 = np.concatenate((u0,
                         (0.00227*u[t] + 0.00454*u[t-1] + 0.00227*u[t-2] + 1.80952*u0[t-1] - 0.8186 * u0[t-2]).r
```

In [9]:
```
a, b = theta_lst.shape
colours = ['blue', 'orange', 'green']
for idx, th in enumerate(theta_lst):
    plt.plot(th)
    plt.hlines(theta[idx], 0, b, colours[idx], '--')
plt.legend(['$\hat{d+1}_0$', '$\hat{d}_1$', '$\hat{c}_0$'])
plt.title('Parameters')
plt.show()
```



Question 2

In [10]:
```
import matplotlib.pyplot as plt
import control as co
import numpy as np
import sympy as sp

from numpy import transpose as trans
from numpy import cos, sin, pi
from numpy.linalg import inv
```

Question Parameters

In [11]:
```
a1, a2, b0, b1 = sp.symbols('\hat{a}_1, \hat{a}_2, \hat{b}_0, \hat{b}_1')
r1, s0, s1, a0 = sp.symbols('r_1, s_0, s_1, a_0')
am1, am2 = sp.symbols('a_{m1}, a_{m2}')
y, u, uc = sp.symbols('y, u, u_c')
q = sp.symbols('q')
```

In [12]:
```
# A = (q**2 - 0.503215*q + 0.0497871)
# B = (1.59831*q + 0.587984)
A = (q**2 + a1*q + a2)
B = (b0*q + b1)
B_poly = sp.Poly(B, q)

H = B/A
B_plus = (q + B_poly.coeffs()[1]/B_poly.coeffs()[0])
B_minus = B_poly.coeffs()[0]
H
```

Out[12]: $\frac{\hat{b}_0 q + \hat{b}_1}{\hat{a}_1 q + \hat{a}_2 + q^2}$

In [13]:
```
# Am_test = co.sample_system(co.tf([1.], [1., 2., 1.]), Ts=2, method='zoh')
# Am = q**2 - 0.2707*q + 0.01832
Am = q**2 + am1*q + am2
Am
```

Out[13]: $a_{m1} q + a_{m2} + q^2$

2.1 Sampler (Bonus)

The original equations for the plany is given by $G = \frac{2}{s^2 + 1.5s + 0.5}$

In [14]:
```
co.sample_system(co.tf([2.], [1., 1.5, 0.5]), Ts=2, method='zoh')
```

Out[14]:

$$\frac{1.598z + 0.588}{z^2 - 0.5032z + 0.04979} \quad dt = 2$$

2.2 Indirect Linear Controller Design Using MDPP

We begin by factorizing B as follows,

$$B = B(q)^+ B(q)^- = (q + 0.36788)(1.59831)$$

To cancel the zeros, we require that they are stable and well-damped. In this case the zero is q = -0.36788 and is thus stable (inside the unit circle) and well-damped (sufficiently far from the unit circle). It is then clear that we may cancel the zeros.

Using the compatability conditions:

$$\mathrm{deg}A_m = \mathrm{deg}A = 2$$

$$\mathrm{deg}B_m = \mathrm{deg}B = \mathrm{degB}^+ = 1$$

$$\mathrm{deg}A_0 = \mathrm{deg}A - \mathrm{deg}B^+ - 1 = 0$$

With this we can find,

$$\mathrm{deg}A_c = \mathrm{deg}A_o + \mathrm{deg}A_m + \mathrm{deg}B^+ = 3$$

From the minimum phase conditions we then get that,

$$\mathrm{deg}R = \mathrm{deg}S = \mathrm{deg}T = \mathrm{deg}A_c - \mathrm{deg}A = 1$$

Noting that $\mathrm{deg}R = 1$ and $\mathrm{deg}B^+ = 1$ then it follows that

$$\mathrm{deg}R' = 0$$

Let $R' = 1$, then the left hand side (LHS) of the Diophatine equation becomes,

$$\mathrm{LHS} = AR' + B^- S$$

In [15]:
```
LHS = A + B_minus*(s0*q + s1)
```

From the compatability conditions, we get the $\mathrm{deg}A_o = 0$, thus we choose it to be,

$$A_o = 1$$

The right hand side (RHS) of the Diophantine equation becomes,

$$\mathrm{RHS} = A_o A_m$$

In [16]:
```
RHS = Am
```

In [17]:
```
dio_poly = sp.Poly(sp.collect(LHS - RHS, q), q)
```

To find s_0 we consider the first degree terms,

In [18]:
```
s0_sym = sp.solve(dio_poly.coeffs()[0], s0)[0]
s0_sym
```

Out[18]: $\frac{-\hat{a}_1 + a_{m1}}{\hat{b}_0}$

To find s_1 we consider the equations that are constant in q,

In [19]:
```
s1_sym = sp.solve(dio_poly.coeffs()[1], s1)[0]
S = (s0_sym*q + s1_sym)
R = B_plus
s1_sym
```

Out[19]: $\frac{-\hat{a}_2 + a_{m2}}{\hat{b}_0}$

With this the S polynomial is,

$S = \frac{q(-\hat{a}_1 + a_{m1})}{\hat{b}_0} + \frac{-\hat{a}_2 + a_{m2}}{\hat{b}_0}$

In [20]:
```
r1_sym = b1/b0
r1_sym
```

Out[20]: $\frac{\hat{b}_1}{\hat{b}_0}$

Additionally, we know that, $R = R'B^+$ and that $R' = 1$ thus,

$R = B^+ = q + \frac{\hat{b}_1}{\hat{b}_0}$

The R polynomial is given as,

$R = q + \frac{\hat{b}_1}{\hat{b}_0}$

The T polynomial is found using,

$$T = A_o B'_m$$

where $B_m = B^- B'_m = A_m(1) q^{n-d_0}$

In [21]:
```
Bm = Am.subs(q, 1)*q
t0_sym = Bm/B_minus
T = t0_sym
```

Then T is,

$T = \frac{q(a_{m1} + a_{m2} + 1)}{\hat{b}_0}$

As a summary, the control polynomials are given by,

$R = q + r_1 = q + \frac{\hat{b}_1}{\hat{b}_0}$

$S = s_0 q + s_1 = \frac{q(-\hat{a}_1 + a_{m1})}{\hat{b}_0} + \frac{-\hat{a}_2 + a_{m2}}{\hat{b}_0}$

$T = t_0 q = \frac{q(a_{m1} + a_{m2} + 1)}{\hat{b}_0}$

These values are then used in the linear controller algorithm,

$Ru(t) = Tu_c(t) - Sy(t)$

$u(t) = -r_1 u(t-1) + t_0 u_c(t) - s_0 y(t) - s_1 y(t-1)$

The output equation is given by,

$y(t) = \phi(t-1)^T \theta$

where,

$\phi(t-1)^T = \left[-y(t-1) \;\; -y(t-2) \;\; u(t-1) \;\; u(t-2)\right]$

$\hat{\theta} = \left[\hat{a}_1 \; \hat{a}_2 \; \hat{b}_0 \; \hat{b}_1\right]^T$

2.3-4 Minimal Degree Pole Placement Algorithm for Indirect Controller and Simulation

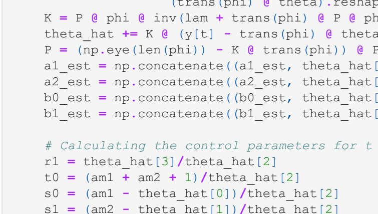
In [22]:
```
def reference_signal(time=np.linspace(0, 100, 101)):
    sig = []
    rat = 2.0/40.0*pi
#     rat = 1.0/25.0*pi
    for t_lop in time:
        if sin(rat*t_lop) >= 0:
            sig.append(1)
        else:
            sig.append(0)
    return np.array(sig, dtype=float)

def noise_generator(time=np.linspace(0, 100, 101)):
    noise = []
    for t_lop in time:
        if t_lop < 35:
            noise.append(0)
        else:
            noise.append(0.25)
    return np.array(noise)
```

We start by initializing the values of the variables,

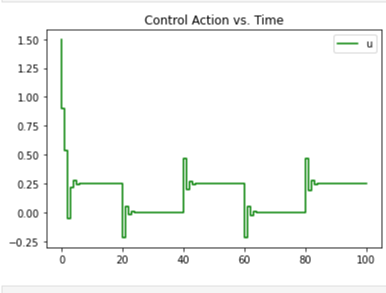
In [23]:
```
uc = reference_signal()
lam = 1

# Assignmnent, works
am1 = -2*np.exp(-2)
am2 = np.exp(-4)
a1_true, a2_true, b0_true, b1_true = -0.503215, 0.0497871, 1.59831, 0.587984

# # In class example, works perfectly!
# am1 = -1.3205
# am2 = 0.4966
# a1_true, a2_true, b0_true, b1_true = -1.6065, 0.6065, 0.1065, 0.0902

theta = np.array([a1_true, a2_true, b0_true, b1_true]).reshape(4,1)  # a1[0], a2[1], b0[2], b1[3]
```

In [24]:
```
# Initializing the coefficients for A, B for t = 0
theta_hat = np.array([0., 0., 0.5, 0.2]).reshape(4,1)  # a1[0], a2[1], b0[2], b1[3]  ** SET b0 TO 0.01 FOR CONV
a1_est, a2_est, b0_est, b1_est = theta_hat[0], theta_hat[1], theta_hat[2], theta_hat[3]
P = np.diag([10, 1, 100, 10])  # WORKS
# P = np.diag([100, 100, 100, 100])  # WORKS

# Calculating the control parameters for t = 0
y = np.array([0])
y_hat = y
t0 = (am1 + am2 + 1)/theta_hat[2]
s0 = (am1 - theta_hat[0])/theta_hat[2]
u = t0*uc[0] - s0*y[0]

# Calculating the coefficients for A, B for t = 1
# phi = np.array([-y[0], 0, u[0], 0]).reshape(4,1)
phi = np.array([0., 0., 0., 0.]).reshape(4,1)
y = np.concatenate((y,
                    (trans(phi) @ theta).reshape(-1)), axis=0)

K = P @ phi @ inv(lam + trans(phi) @ P @ phi)
theta_hat = theta_hat + K @ (y[1] - trans(phi) @ theta_hat)
P = (np.eye(len(phi)) - K @ trans(phi)) @ P /lam

a1_est = np.concatenate((a1_est, theta_hat[0]), axis=0)
a2_est = np.concatenate((a2_est, theta_hat[1]), axis=0)
b0_est = np.concatenate((b0_est, theta_hat[2]), axis=0)
b1_est = np.concatenate((b1_est, theta_hat[3]), axis=0)

# Calculating the control parameters for t = 1
r1 = theta_hat[3]/theta_hat[2]
t0 = (am1 + am2 + 1)/theta_hat[2]
s0 = (am1 - theta_hat[0])/theta_hat[2]
s1 = (am2 - theta_hat[1])/theta_hat[2]
u = np.concatenate((u,
                    -r1*u[0] + t0*uc[1] - s0*y[1] - s1*y[0]))

for t in np.arange(2, 101):
    # Calculating the coefficients for A, B for t
    phi = np.array([-y[t-1], -y[t-2], u[t-1], u[t-2]]).reshape(4,1)
    y = np.concatenate((y,
                        (trans(phi) @ theta).reshape(-1)), axis=0)
    K = P @ phi @ inv(lam + trans(phi) @ P @ phi)
    theta_hat += K @ (y[t] - trans(phi) @ theta_hat)
    P = (np.eye(len(phi)) - K @ trans(phi)) @ P /lam
    a1_est = np.concatenate((a1_est, theta_hat[0]), axis=0)
    a2_est = np.concatenate((a2_est, theta_hat[1]), axis=0)
    b0_est = np.concatenate((b0_est, theta_hat[2]), axis=0)
    b1_est = np.concatenate((b1_est, theta_hat[3]), axis=0)

    # Calculating the control parameters for t
    r1 = theta_hat[3]/theta_hat[2]
    t0 = (am1 + am2 + 1)/theta_hat[2]
    s0 = (am1 - theta_hat[0])/theta_hat[2]
    s1 = (am2 - theta_hat[1])/theta_hat[2]
    u = np.concatenate((u,
                        -r1*u[t-1] + t0*uc[t] - s0*y[t] - s1*y[t-1]))
```

In [25]:
```
plt.plot(np.linspace(0, 100, 101), y)
plt.plot(np.linspace(0, 100, 101), uc)
# plt.plot(y[45:60])
# plt.plot(uc[45:60])
plt.legend(['y',
            'u_c'])
plt.title('System Output and Reference')
plt.show()
```



In [26]:
```
plt.step(np.linspace(0, 100, 101), u, 'g')
# plt.step(u[5:], 'g')
plt.legend(['u'])
plt.title('Control Action vs. Time')
plt.show()
```



In [27]:
```
plt.plot(a1_est)
plt.hlines(a1_true, 0, 100, 'blue', '--')

plt.plot(a2_est)
plt.hlines(a2_true, 0, 100, 'orange', '--')

plt.plot(b0_est)
plt.hlines(b0_true, 0, 100, 'green', '--')

plt.plot(b1_est)
plt.hlines(b1_true, 0, 100, 'red', '--')
plt.title('Parameters')
plt.legend(['$\hat{a}_1$', '$\hat{a}_2$', '$\hat{b}_0$', '$\hat{b}_1$'])
plt.show()
```

2.5 Disturbance Rejection

In [28]:
```
R0 = R
S0 = S
T0 = T
ro1, so0, so1 = sp.symbols('r_{o1},s_{o0},s_{o1}]')
x0, y0 = sp.symbols('x_0,y_0')
```

In [29]:
```
X = q
Y = -sp.simplify(R0.subs(q, 1)/B.subs(q, 1))
```

Placing the integrator pole at the origin gives,

$X = q + x_0 = q$

and we define Y to be,

$Y = y_0 = -\frac{R^0(1)}{B(1)} = -\frac{1}{\hat{b}_0}$

We find the new polynomial $R = XR^0 + YB$

In [30]:
```
Rdist = sp.simplify(X*(q + ro1) + y0*B)
Rdist = sp.collect(sp.expand(Rdist.subs(ro1, r1_sym).subs(y0, Y)), q)
```

$R = q^2 + q\left(-1 + \frac{\hat{b}_1}{\hat{b}_0}\right) - \frac{\hat{b}_1}{\hat{b}_0}$

Following the same method, we find $S = XS^0 - YA$

In [31]:
```
Sdist = sp.simplify(X*(so0*q + so1) - y0*A)
Sdist = sp.collect(sp.expand(Sdist).subs(so0, s0_sym).subs(so1, s1_sym).subs(y0, Y), q)
Sdist_poly = sp.Poly(Sdist, q)
```

$S = \frac{\hat{a}_2}{\hat{b}_0} + q^2\left(\frac{-\hat{a}_1 + a_{m1}}{\hat{b}_0} + \frac{1}{\hat{b}_0}\right) + q\left(\frac{\hat{a}_1}{\hat{b}_0} + \frac{-\hat{a}_2 + a_{m2}}{\hat{b}_0}\right)$

$T = \frac{q(a_{m1} + a_{m2} + 1)}{\hat{b}_0}$

In [ ]:

Noting that the T polynomial doesn't change we get the control signal as,

$u(t) = -r_{11} u(t-1) - r_{12} u(t-2) + t_0 u_c(t-1) - s_{10} y(t) - s_{11} y(t-1) - s_{12} y(t-2)$

We now model the system as,

$A_d A y(t) = A_d B\big(u_(t) + v(t)\big)$

we define $A_d = (q-1)$

Factoring and cancelling a q term on each side,

$(1 - q^{-1}) A y(t) = (1 - q^{-1}) B\big(u_(t) + v(t)\big)$

Note that the addition of the A_d term to compensate for the noise changes the estimator, and thus changes the regressor. The RLS is now learning the filtered output, $y_f(t) = A_d y(t)$, but the paramters stay the same. To fix the model, we note that,

$y_f(t) = (1 - q^{-1}) y(t)$

$y(t) = y_f(t) + y(t-1)$

The same argument is made for $u_f(t) = u(t) - u(t-1)$

2.6 Plotting

Uncompensated Noise

The following blocks find the uncompensated response with disturbance

In [32]:
```
uc = reference_signal()
v = noise_generator()
lam = 1

# Assignmnent, works
am1 = -2*np.exp(-2)
am2 = np.exp(-4)
a1_true, a2_true, b0_true, b1_true = -0.503215, 0.0497871, 1.59831, 0.587984

# # In class example, works perfectly!
# am1 = -1.3205
# am2 = 0.4966
# a1_true, a2_true, b0_true, b1_true = -1.6065, 0.6065, 0.1065, 0.0902

theta = np.array([a1_true, a2_true, b0_true, b1_true]).reshape(4,1)  # a1[0], a2[1], b0[2], b1[3]
```

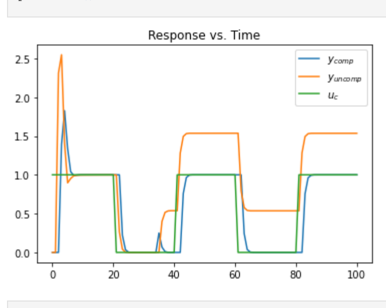
In [33]:
```python
# Initializing the coefficients for A, B for t = 0
theta_hat = np.array([0., 0., 0.5, 0.2]).reshape(4,1)  # a1[0], a2[1], b0[2], b1[3]  ** SET b0 TO 0.01 FOR CONV
a1_est, a2_est, b0_est, b1_est = theta_hat[0], theta_hat[1], theta_hat[2], theta_hat[3]
P = np.diag([10, 1, 100, 10])  # WORKS
# P = np.diag([100, 100, 100, 100])  # WORKS

# Calculating the control parameters for t = 0
y = np.array([0])
y_hat = y
t0 = (am1 + am2 + 1)/theta_hat[2]
s0 = (am1 - theta_hat[0])/theta_hat[2]
u = t0*uc[0] - s0*y[0]

e_uncomp = np.array([y[0] - uc[0]])

# Calculating the coefficients for A, B for t = 1
# phi = np.array([-y[0], 0, u[0], 0]).reshape(4,1)
phi = np.array([0., 0., 0., 0.]).reshape(4,1)
y = np.concatenate((y,
                    (trans(phi) @ theta).reshape(-1)), axis=0)
e_uncomp = np.concatenate((e_uncomp, np.array([y[1] - uc[1]])))

K = P @ phi @ inv(lam + trans(phi) @ P @ phi)
theta_hat = theta_hat + K @ (y[1] - trans(phi) @ theta_hat)
P = (np.eye(len(phi)) - K @ trans(phi)) @ P /lam

a1_est = np.concatenate((a1_est, theta_hat[0]), axis=0)
a2_est = np.concatenate((a2_est, theta_hat[1]), axis=0)
b0_est = np.concatenate((b0_est, theta_hat[2]), axis=0)
b1_est = np.concatenate((b1_est, theta_hat[3]), axis=0)

# Calculating the control parameters for t = 1
r1 = theta_hat[3]/theta_hat[2]
t0 = (am1 + am2 + 1)/theta_hat[2]
s0 = (am1 - theta_hat[0])/theta_hat[2]
s1 = (am2 - theta_hat[1])/theta_hat[2]
u = np.concatenate((u,
                    -r1*u[0] + t0*uc[1] - s0*y[1] - s1*y[0]))

for t in np.arange(2, 101):
    # Calculating the coefficients for A, B for t
    phi = np.array([-y[t-1], -y[t-2], u[t-1], u[t-2]]).reshape(4,1)
    y = np.concatenate((y,
                        (trans(phi) @ theta).reshape(-1)), axis=0)
    e_uncomp = np.concatenate((e_uncomp, np.array([y[t] - uc[t]])))
    K = P @ phi @ inv(lam + trans(phi) @ P @ phi)
    theta_hat += K @ (y[t] - trans(phi) @ theta_hat)
    P = (np.eye(len(phi)) - K @ trans(phi)) @ P /lam
    a1_est = np.concatenate((a1_est, theta_hat[0]), axis=0)
    a2_est = np.concatenate((a2_est, theta_hat[1]), axis=0)
    b0_est = np.concatenate((b0_est, theta_hat[2]), axis=0)
    b1_est = np.concatenate((b1_est, theta_hat[3]), axis=0)

    # Calculating the control parameters for t
    r1 = theta_hat[3]/theta_hat[2]
    t0 = (am1 + am2 + 1)/theta_hat[2]
    s0 = (am1 - theta_hat[0])/theta_hat[2]
    s1 = (am2 - theta_hat[1])/theta_hat[2]
    u = np.concatenate((u,
                        -r1*u[t-1] + t0*uc[t] - s0*y[t] - s1*y[t-1] + v[t]))

y_uncomp = y
```

In [34]:
```python
def params():
    global theta_hat, am1, am2
    rdist1 = (theta_hat[3]/theta_hat[2] - 1)
    rdist2 = -theta_hat[3]/theta_hat[2]
    sdist0 = (am1 - theta_hat[0] + 1)/(theta_hat[2])
    sdist1 = (theta_hat[0] - theta_hat[1] + am2)/theta_hat[2]
    sdist2 = theta_hat[1]/theta_hat[2]
    tdist0 = (am1 + am2 + 1)/theta_hat[2]
#     print(np.array([rdist1, rdist2, sdist0, sdist1, sdist2, tdist0]).reshape(1,-1))
    return np.array([rdist1, rdist2, sdist0, sdist1, sdist2, tdist0]).reshape(1,-1)
```

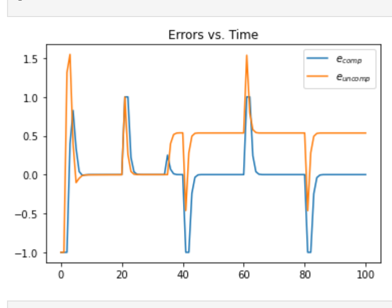
In [35]:
```python
uc = reference_signal()
v = noise_generator()
lam = 1
# Assignmnent, works
am1 = -2*np.exp(-2)
am2 = np.exp(-4)
a1_true, a2_true, b0_true, b1_true = -0.503215, 0.0497871, 1.59831, 0.587984


theta = np.array([a1_true, a2_true, b0_true, b1_true]).reshape(4,1)  # a1[0], a2[1], b0[2], b1[3]
```

In [36]:
```python
# Initializing the coefficients for A, B for t = 0
theta_hat = np.array([0., 0., 1., 0.2]).reshape(4,1)  # a1[0], a2[1], b0[2], b1[3]  ** SET b0 TO 0.01 FOR CONVE
theta_lst = theta_hat
a1_est, a2_est, b0_est, b1_est = theta_hat[0], theta_hat[1], theta_hat[2], theta_hat[3]
P = np.diag([10, 10, 100, 10])  # WORKS
# P = 100*np.eye(4)

# Calculating the control parameters for t = 0
yf = np.array([0] + v[0])
y = np.array([0])
vec = np.array([0., 0., -y[0], 0., 0., 0.]).reshape(-1,1)
u = (params()@vec).reshape(-1)
uf = np.array([u[0]])

e_comp = np.array([y[0] - uc[0]])


# Calculating the coefficients for A, B for t = 1
phi = np.array([0., 0., 0., 0.]).reshape(4,1)
yf = np.concatenate((yf,
                     (trans(phi) @ theta + v[1] - v[0]).reshape(-1)), axis=0)
y = np.concatenate((y,
                    (yf[1] + y[0]).reshape(-1)), axis=0)
e_comp = np.concatenate((e_comp, np.array([y[1] - uc[1]])))
vec = np.array([-u[0], 0., -y[1], -y[0], 0., uc[0]]).reshape(-1,1)
u = np.concatenate((u,
                    (params()@vec).reshape(-1)), axis=0)
uf = np.concatenate((uf,
                     (u[1] - u[0]).reshape(-1)), axis=0)
K = P @ phi @ inv(lam + trans(phi) @ P @ phi)
theta_hat = theta_hat + K @ (yf[1] - trans(phi) @ theta_hat)
theta_lst = np.concatenate((theta_lst, theta_hat), axis=1)
P = (np.eye(len(phi)) - K @ trans(phi)) @ P /lam

# Calculating the coefficients for A, B for t = 2
phi = np.array([-yf[0], 0., uf[0], 0.]).reshape(4,1)
yf = np.concatenate((yf,
                     (trans(phi) @ theta + v[2] - v[1]).reshape(-1)), axis=0)
y = np.concatenate((y,
                    (yf[2] + y[1]).reshape(-1)), axis=0)
e_comp = np.concatenate((e_comp, np.array([y[2] - uc[2]])))
vec = np.array([-u[1], -u[0], -y[2], -y[1], -y[0], uc[1]]).reshape(-1,1)
u = np.concatenate((u,
                    (params()@vec).reshape(-1)), axis=-1)
uf = np.concatenate((uf,
                     (u[2] - u[1]).reshape(-1)), axis=0)
K = P @ phi @ inv(lam + trans(phi) @ P @ phi)
theta_hat = theta_hat + K @ (yf[2] - trans(phi) @ theta_hat)
theta_lst = np.concatenate((theta_lst, theta_hat), axis=1)
P = (np.eye(len(phi)) - K @ trans(phi)) @ P /lam

for t in np.arange(3, 101):
    # Calculating the coefficients for A, B for t = 2
    phi = np.array([-yf[t-1], -yf[t-2], uf[t-1], uf[t-2]]).reshape(4,1)
    yf = np.concatenate((yf,
                         (trans(phi) @ theta + v[t] - v[t-1]).reshape(-1)), axis=0)
    y = np.concatenate((y,
                        (yf[t] + y[t-1]).reshape(-1)), axis=0)
    e_comp = np.concatenate((e_comp, np.array([y[t] - uc[t]])))
    vec = np.array([-u[t-1], -u[t-2], -y[t], -y[t-1], -y[t-2], uc[t-1]]).reshape(-1,1)
    u = np.concatenate((u,
                        (params()@vec).reshape(-1)), axis=0)
    uf = np.concatenate((uf,
                         (u[t] - u[t-1]).reshape(-1)), axis=0)
    K = P @ phi @ inv(lam + trans(phi) @ P @ phi)
    theta_hat = theta_hat + K @ (yf[t] - trans(phi) @ theta_hat)
    theta_lst = np.concatenate((theta_lst, theta_hat), axis=1)
    P = (np.eye(len(phi)) - K @ trans(phi)) @ P /lam

y_comp = y
```

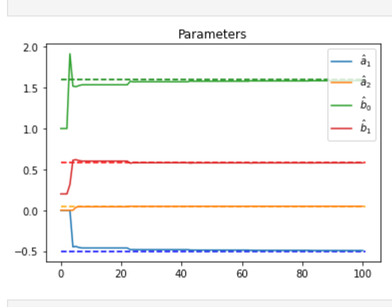
In [37]:
```python
plt.plot(y_comp)
plt.plot(y_uncomp)
plt.plot(uc)
plt.legend(['$y_{comp}$', '$y_{uncomp}$', '$u_c$'])
plt.title('Response vs. Time')
plt.show()
```



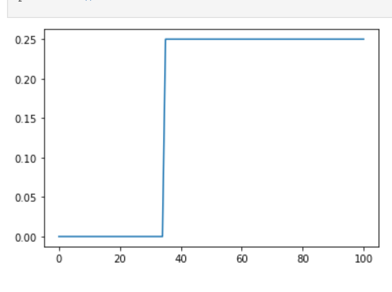
In [38]:
```python
plt.plot(e_comp)
plt.plot(e_uncomp)
plt.legend(['$e_{comp}$', '$e_{uncomp}$'])
plt.title('Errors vs. Time')
plt.show()
```



In [39]:
```python
a, b = theta_lst.shape
colours = ['blue', 'orange', 'green', 'red']
for idx, th in enumerate(theta_lst):
    plt.plot(th)
    plt.hlines(theta[idx], 0, b, colours[idx], '--')
plt.title('Parameters')
plt.legend(['$\hat{a}_1$', '$\hat{a}_2$', '$\hat{b}_0$', '$\hat{b}_1$'])
plt.show()
```



In [40]:
```python
plt.plot(v)
plt.show()
```



Question 3

Q3.1 Derivation of Control Parameters Direct Self-Tuning Regulator

Given that the system is the same as in the previous question, the control parameters for the direct self-tuning regulator will also be the same. To find the reparameterized model, we look into the equation,

$$A_o A_m y(t) = B^-\left(Ru(t) + Sy(t)\right)$$

From the previous example, we know the following,

$A_0 = 1$

$A_m = a_{m1}q + a_{m2} + q^2$

$B^- = \hat{b}_0$

$R = q + r_1 = q + \frac{\hat{b}_1}{\hat{b}_0}$

$S = s_0 q + s_1 = \frac{q\left(-\hat{a}_1 + a_{m1}\right)}{\hat{b}_0} + \frac{-\hat{a}_2 + a_{m2}}{\hat{b}_0}$

$T = t_0 q = \frac{q\left(a_{m1} + a_{m2} + 1\right)}{\hat{b}_0}$

Then the equation above becomes,

$$y(t) = \frac{1}{q^2 + a_{m_1}q + a_{m_2}}\hat{b}_0\Big(\big(q + r_1\big)u(t) + \big(s_0 q + s_1\big)y(t)\Big)$$

Commented out vv ^^

To ensure that we are working in the backwords shift operator, we multiply the top and bottom of the equation by q^{-2}. Then,

$$y(t) = \frac{1}{1 + a_{m_1}q^{-1} + a_{m_2}q^{-2}}\Big(\big(\hat{b}_0 q^{-1} + \hat{b}_0 r_1 q^{-2}\big)u(t) + \big(\hat{b}_0 s_0 q^{-1} + \hat{b}_0 s_1 q^{-2}\big)y(t)\Big)$$

Then by distributing the outermost bracket into the inputs and outputs, $u(t)$ and $y(t)$ respectively, the output may be written as,

$$y(t) = \Big(\big(\hat{b}_0 q^{-1} + \hat{b}_0 r_1 q^{-2}\big)u_f(t) + \big(\hat{b}_0 s_0 q^{-1} + \hat{b}_0 s_1 q^{-2}\big)y_f(t)\Big)$$

where,

$$u_f(t) = \frac{1}{1 + a_{m_1}q^{-1} + a_{m_2}q^{-2}}u(t)$$

$$y_f(t) = \frac{1}{1 + a_{m_1}q^{-1} + a_{m_2}q^{-2}}y(t)$$

Expanding these equations we get,

$$u_f(t) = u(t) - a_{m_1}u_f(t-1) - a_{m_2}u_f(t-2)$$

and

$$y_f(t) = u(t) - a_{m_1}y_f(t-1) - a_{m_2}y_f(t-2)$$

Recalling the output of the system we can redefine it to be,

$$y(t) = \hat{b}_0 u_f(t-1) + \hat{b}_0 r_1 u_f(t-2) + \hat{b}_0 s_0 y_f(t-1) + \hat{b}_0 s_1 y_f(t-2) = \phi(t-1)^T\theta$$

with,

$\phi(t) = \left[u_f(t)\ u_f(t-1)\ y_f(t)\ y_f(t-1)\right]^T$

and

$\theta = \left[\hat{r}_0\ \hat{r}_1\ \hat{s}_0\ \hat{s}_1\right]^T$

With these known, we can use the recursive linear regressor to approximate the parameters for θ

Noting that R is no longer monic due to the factorization of B^- we get,

$S = \hat{s}_0 s + \hat{s}_1$

$R = \hat{r}_0 s + \hat{r}_1$

$T = t_0 q$

The control action is then found using,

$Ru(t) = Tu_c(t) - Sy(t)$

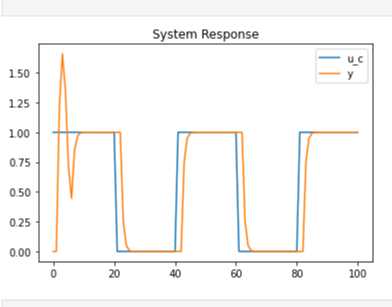
$u(t) = \frac{1}{\hat{r}_0}\big[-\hat{r}_1 u(t-1) + t_0 u_c(t) - s_0 y(t) - s_1 y(t-1)\big]$

Q3.2 Implementation of Direct Self-Tuning Regulator

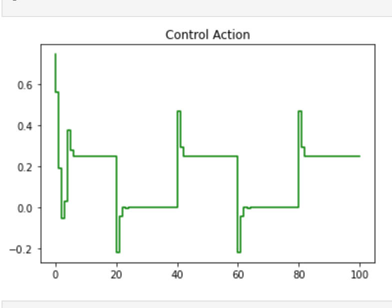
In [41]:
```python
uc = reference_signal()
lam = 1

# Assignmnent, works
am1 = -2*np.exp(-2)
am2 = np.exp(-4)
r0_true, r1_true, s0_true, s1_true = 1.59831, 0.587984, 0.23254443, -0.03147146
t0_true = 1 + am1 + am2
t0 = t0_true

theta = np.array([r0_true, r1_true, s0_true, s1_true]).reshape(4,-1)
```

In [42]:
```python
# Initializing the coefficients for R, S for t = 0
y = np.array([0])
theta_hat = np.array([1., 0.25, 0.25, 0.25]).reshape(4,-1)
theta_lst = np.array(theta_hat)
# P = np.diag([1000, 100, 1000, 1000])  # WORKS
P = 100*np.eye(4)

# Calculating the control parameters for t = 0
u = 1/theta_hat[0]*(t0*uc[0] - theta_hat[2]*y[0])
uf = np.array([u[0]])
yf = np.array([y[0]])



# Initializing the coefficients for A, B for t = 1
phi = np.array([0., 0., 0., 0.]).reshape(4,-1)
y = np.concatenate((y,
                    (trans(phi)@theta).reshape(-1,)), axis=0)
K = P @ phi @ inv(lam + trans(phi) @ P @ phi)
theta_hat = theta_hat + K @ (y[1] - trans(phi) @ theta_hat)
theta_lst = np.concatenate((theta_lst, theta_hat), axis=1)
P = (np.eye(len(phi)) - K @ trans(phi)) @ P /lam

# Calculating the control parameters for t = 1
u = np.concatenate((u,
                    (1/theta_hat[0]*(-theta_hat[1]*u[0] + t0*uc[1] - theta_hat[2]*y[1] - theta_hat[3]*y[0])).res
                     axis=0)
uf = np.concatenate((uf,
                     (u[1] - am1*uf[0]).reshape(-1,)), axis=0)
yf = np.concatenate((yf,
                     (y[1] - am1*yf[0]).reshape(-1,)), axis=0)

phi = np.array([uf[1], 0., yf[1], 0.]).reshape(4,-1)
for t in np.arange(2, 101):
    # Initializing the coefficients for A, B for t = t
    y = np.concatenate((y,
                        (trans(phi)@theta).reshape(-1,)), axis=0)
    K = P @ phi @ inv(lam + trans(phi) @ P @ phi)
    theta_hat = theta_hat + K @ (y[t] - trans(phi) @ theta_hat)
    theta_lst = np.concatenate((theta_lst, theta_hat), axis=1)
    P = (np.eye(len(phi)) - K @ trans(phi)) @ P /lam

    # Calculating the control parameters for t = t
    u = np.concatenate((u,
                        (1/theta_hat[0]*(-theta_hat[1]*u[t-1] + t0*uc[t] - theta_hat[2]*y[t] - theta_hat[3]*y[t-
                        axis=0)
    uf = np.concatenate((uf,
                         (u[t] - am1*uf[t-1] - am2*uf[t-2]).reshape(-1,)), axis=0)
    yf = np.concatenate((yf,
                         (y[t] - am1*yf[t-1] - am2*yf[t-2]).reshape(-1,)), axis=0)
    phi = np.array([uf[t-1], uf[t-2], yf[t-1], yf[t-2]]).reshape(4,-1)
```

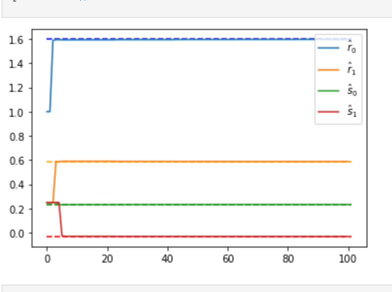
In [43]:
```python
plt.plot(uc)
plt.plot(y)
plt.legend(['u_c', 'y'])
plt.title('System Response')
plt.show()
```



In [44]:
```python
plt.step(np.linspace(0, 100, 101), u, 'g')
plt.title('Control Action')
plt.show()
```



In [45]:
```python
a, b = theta_lst.shape
colours = ['blue', 'orange', 'green', 'red']
for idx, th in enumerate(theta_lst):
    plt.plot(th)
    plt.hlines(theta[idx], 0, b, colours[idx], '--')
plt.legend(['$\hat{r}_0$', '$\hat{r}_1$', '$\hat{s}_0$', '$\hat{s}_1$'])
plt.show()
```



In [ ]:

In [ ]: