1. Solve Exercise 4.9, parts (b)–(e) on page 247 in [Ioannou-Sun].

Solution. From part (a) we get
\[ m\ddot{y} + \beta \dot{y} + ky = u. \]

Taking Laplace transform of both sides gives
\[ \mathcal{L}\{y\}(s) = \frac{1/m}{s^2 + (\beta/m)s + k/m} \mathcal{L}\{u\}(s). \]

Filtering both sides with a second order stable filter
\[ \frac{1}{\Lambda(s)} := \frac{1}{(s + \lambda_1)(s + \lambda_2)}, \quad \lambda_1, \lambda_2 \in (-\infty, 0) \]
gives
\[ z = \theta^T \phi \]
where
\[ z(s) := \frac{s^2}{\Lambda(s)} \mathcal{L}\{y\}(s), \]
and
\[ \theta := \frac{1}{m} \begin{pmatrix} 1 \\ \beta \\ k \end{pmatrix}, \quad \phi(s) := \frac{1}{\Lambda(s)} \begin{pmatrix} \mathcal{L}\{u\}(s) \\ -s\mathcal{L}\{y\}(s) \\ -\mathcal{L}\{y\}(s) \end{pmatrix}. \]

(b) The estimate of \( z \) is defined by
\[ \hat{z} := \hat{\theta}^T \phi, \]
where \( \hat{\theta} \) is the estimate of \( \theta \). Normalizing the estimates gives
\[ \hat{\phi} := \frac{\phi}{n}, \quad \hat{z} := \theta^T \hat{\phi}, \quad \hat{\hat{z}} := \hat{\theta}^T \hat{\phi}, \]
where \( n \) is the normalizing signal defined by \( n := \sqrt{1 + \phi^T \phi} \) (so that \( \hat{\phi} = \phi/n \in L_\infty \)). Let \( \hat{\theta} := \hat{\theta} - \theta \) and \( \hat{e} := \hat{\hat{z}} - \hat{z} \) be the estimation errors. Then
\[ \hat{e} = \hat{\theta}^T \hat{\phi}. \]

Define the cost function
\[ J(\hat{\theta}) := \frac{\hat{e}^2}{2} = \frac{(\hat{\theta}^T \hat{\phi})^2}{2} \]
and calculate its gradient
\[ \nabla J(\hat{\theta}) = (\hat{\theta}^T \hat{\phi}) \hat{\phi} = \hat{e} \hat{\phi}. \]
Hence the gradient adaptive law is given as
\[ \dot{\hat{\theta}} = -\Gamma \hat{e} \hat{\phi}, \]
where \( \Gamma > 0 \) is a scaling matrix (adaptive gain).

(c) The continuous-time recursive least-squares algorithm is given as
\[ \dot{\hat{\theta}} = -P \hat{e} \hat{\phi}, \quad \dot{P} = -P \hat{\phi} \hat{\phi}^T P. \]

(d) For the gradient method, we set \( \Gamma = 10 I \). A single sinusoidal source is used with frequency \( \pi \) and amplitude 100. The Simulink diagram and the simulation result can be found in Fig. 1. We see from the simulation result that the estimate \( \hat{\theta} \) goes to \( \theta = (1, \beta, k)/m = (0.05, 0.005, 0.25) \).
For the least-square method, we set $P(0) = 100I$. The same sinusoidal source with frequency $\pi$ and amplitude 100 is used. The Simulink diagram and the simulation result can be found in Fig. 2.
We see from the simulation result that the estimate $\hat{\theta}$ goes to $\theta = (1, \beta, k)/m = (0.05, 0.005, 0.25)$.

(c) Since $m$ is time-varying, we need to substitute the dynamics of $y$ in both Simulink diagrams with the subsystem in Fig. 3a.

For the gradient method, we set $\Gamma = 10I$. The same sinusoidal source with frequency $\pi$ and amplitude 100 is used. The simulation result can be found in Fig. 3b.
We see from the simulation result that the estimate $\hat{\theta}$ goes to $\theta$ (dashed-line), which is slowly time-varying in this case.
Implement the indirect MRAC control design example given in class via computer simulation. Plot the behavior of the plant state as well as control gains over time. Does loss of stabilizability ($\hat{b} \to 0$) actually happen? Investigate this in the following cases:

a) $b > 0$ and $\hat{b}$ is initialized with the correct sign ($\hat{b}(0) > 0$).

b) $b > 0$ but $\hat{b}$ is initialized with the wrong sign ($\hat{b}(0) < 0$).

In each case, if you experience loss of stabilizability, modify the update law for $\hat{b}$ to get rid of the problem. Submit both sets of plots (with and without modification) and compare them.

**Solution.** The indirect MRAC model is given by
Plant:
\[ \dot{y} = ay + bu; \]

Reference model:
\[ \dot{y}_m = -a_m y_m + b_m r; \]

Controller:
\[ u = -\hat{k} y + \hat{l} r, \quad \hat{k} = \frac{\hat{a} + a_m}{\hat{b}}, \quad \hat{l} = \frac{b_m}{\hat{b}}; \]

Adaptation:
\[ e = y_m - y, \quad \hat{a} = -\gamma e y, \quad \hat{b} = -\gamma e u. \]

In the simulation, we set \((a, b) = (5, 3)\) and \((a_m, b_m) = (3, 1)\).

a) When \(\hat{b}\) is initialized with the correct sign, if the gain \(\gamma\) is relatively small (e.g. \(\hat{b}(0) = 1, \gamma = 0.1\)) then there is no loss of stabilizability. The simulation result can be found in Fig. 6. (Note that you need to assign the values of \(bHat0 = 1, \gamma = .1\) in MatLab before simulating the Simulink file on course website.)

![Fig. 6: Problem 2(a). Initialized with correct sign, no loss of stabilizability](image-url)

On the other hand, if the gain \(\gamma\) is relatively large (e.g. \(\hat{b}(0) = 5, \gamma = 10\)) then there is loss of stabilizability. The simulation result can be found in Fig. 7. (Note that you need to assign the values of \(bHat0 = 5, \gamma = 10\) in MatLab before simulating the Simulink file on course website.)

![Fig. 7: Problem 2(a). Initialized with correct sign, loss of stabilizability](image-url)

In this case, we prevent the loss of stabilizability using projection with \(b_0 = 0.2\). The simulation result can be found in Fig. 8.

b) When \(\hat{b}\) is initialized with the incorrect sign, there is always loss of stabilizability. The simulation result can be found in Fig. 9. In this case, we cannot prevent the loss of stabilizability using projection.

3. Consider the scalar system \(\dot{x} = f(x) + g(x) u\) where \(f(x) = -x \sin^2(x^2), g(x) = \cos(x^2)\).

a) Construct a feedback law \(u = k(x)\) which makes the closed-loop system GAS. (Justify this.)
b) Now suppose that the state measurements available to the feedback law are affected by an additive disturbance, resulting in the system \( \dot{x} = f(x) + g(x)k(x + d) \) where \( f, g \) are the same as before and \( k \) is the feedback you found in part a). Is this system ISS with respect to \( d \)? Prove or disprove.

**Solution.** The system is given by

\[ \dot{x} = -x \sin^2(x^2) + \cos(x^2)u. \]

a) Setting \( u = k(x) := -x \cos(x^2) \) gives the closed-loop system

\[ \dot{x} = -x(\sin^2(x^2) + \cos^2(x^2)) = -x. \]

Consider \( V : \mathbb{R} \rightarrow [0, \infty) \) defined by \( V(x) := x^2/2 \). Then \( \dot{V}(x) = x\dot{x} = -x^2 \leq 0 \) and \( \dot{V}(x) < 0 \) for all \( x \neq 0 \). Hence Theorem 2 implies that the closed-loop system is GAS.

b) The closed-loop system is given by

\[ \dot{x} = -x \sin^2(x^2) + \cos(x^2)k(x + d) = -x \sin^2(x^2) - (x + d) \cos(x^2) \cos((x + d)^2) =: h(x, d). \]

Notice that \( h \) is a continuous function in both arguments. Consider the sequences \( (a_n)_{n=1}^{\infty}, (d_n)_{n=1}^{\infty} \) defined by

\[ a_n := \sqrt{n\pi}, \quad d_n := \sqrt{(n + 1)\pi} - \sqrt{n\pi}. \]

It is straight forward to see that all \( d_n > 0 \), and \( a_n \rightarrow \infty, d_n \rightarrow 0 \) as \( n \rightarrow \infty \). Moreover,

\[ h(0, d_n) = -d_n \cos(d_n^2) < 0 \]

for all \( \sqrt{n + 1} - \sqrt{n} < 1/\sqrt{2} \), that is, all \( n \geq 1 \); and

\[ h(a_n, d_n) = -\sqrt{(n + 1)\pi} \cos(n\pi) \cos((n + 1)\pi) = \sqrt{(n + 1)\pi} > 0 \]

for all \( n \geq 1 \). By the continuity of \( h \), we get that for each \( n \geq 1 \), there exist a \( b_n \in (0, a_n) \) such that \( h(b_n, d_n) = 0 \) and \( h(x, d_n) > 0 \) for all \( x \in (b_n, a_n] \). Hence if the initial value \( x_n(0) \in (b_n, a_n] \), the state
satisfies \( x_n(t) \geq x_n(0) > b_n > 0 \) for all \( t \geq 0 \). Since \( a_n \to \infty, d_n \to 0 \) as \( n \to \infty \), this means that, when \( d_n \to 0 \), the state does not necessarily go to 0 as \( t \to \infty \). On the other hand, the system is ISS with respect to \( d \) only if there are \( \beta \in K\mathcal{L}, \gamma \in K_{\infty} \) such that
\[
|x(t)| \leq \beta(|x(0)|, t) + \gamma(|d|_t) \quad \forall \ t \geq 0,
\]
where \( |d|_t \) is the essential supremum of \( |d| \) on \([0,t] \). This implies that when \( d_n \to 0 \), we get \( x \to 0 \) as \( t \to \infty \). Therefore, the system is not ISS with respect to \( d \).

4. Consider again the system from the previous homework:
\[
\dot{x} = \theta x + \xi_1 \\
\dot{\xi}_1 = \xi_2 \\
\dot{\xi}_2 = u
\]
Use the adaptive ISS backstepping procedure given in Section 7.4.1 in the notes to design a feedback law that makes the closed-loop system ISS with respect to \((\tilde{\theta}, \dot{\tilde{\theta}})^T\) (for an arbitrary tuning law). You need to do the initialization step as well as the two recursion steps.

**Solution.**

**Step 0.** Consider the system
\[
\dot{x} = \theta x + u.
\]
By setting
\[
u := k_0(x, \tilde{\theta}) := -\tilde{\theta}x - x - x^3
\]
we get the closed-loop system
\[
\dot{x} = -\tilde{\theta}x - x - x^3,
\]
where \( \tilde{\theta} = \hat{\theta} - \theta \). Let \( V_0 : \mathbb{R} \to [0, \infty) \) be defined by
\[
V_0(x) := \frac{1}{2}x^2.
\]
Its derivative along the state trajectory is
\[
\dot{V}_0(x) = x(\theta x + k_0) \tag{1}
\]
\[
= -\tilde{\theta}x^2 - x^4 - x^2 \leq -x^2 + \tilde{\theta}^2 - \left(x^2 + \frac{1}{2} \tilde{\theta}^2\right)^2 \leq -x^2 + \tilde{\theta}^2. \tag{2}
\]

**Step 1.** Consider the system
\[
\dot{x} = \theta x + \xi_1, \quad \dot{\xi}_1 = u.
\]
Let \( V_1 : \mathbb{R}^3 \to [0, \infty) \) be defined by
\[
V_1(x, \tilde{\theta}, \xi_1) := V_0(x) + \frac{1}{2}(\xi_1 - k_0(x, \tilde{\theta}))^2.
\]
Its derivative along the state trajectory is
\[
\dot{V}_1(x, \tilde{\theta}, \xi_1) = x(\theta x + \xi_1) + (\xi_1 - k_0)\left(u - \frac{\partial k_0}{\partial x}(\theta x + \xi_1) - \frac{\partial k_0}{\partial \tilde{\theta}} \dot{\tilde{\theta}}\right) \tag{3}
\]
\[
= x(\theta x + k_0) + (\xi_1 - k_0)\left(u + x - \frac{\partial k_0}{\partial x}(\theta x + \xi_1) - \frac{\partial k_0}{\partial \tilde{\theta}} \dot{\tilde{\theta}}\right).
\]
By (1) and (2)

\[
\dot{V}_1(x, \dot{\theta}, \xi_1) \leq -x^2 + \dot{\theta}^2 + (\xi_1 - k_0) \left( u + x - \frac{\partial k_0}{\partial x}(\theta x + \xi_1) - \frac{\partial k_0}{\partial \theta} \dot{\theta} \right)
\]

\[
= -x^2 + \dot{\theta}^2 + (\xi_1 - k_0) \left( u + x - \frac{\partial k_0}{\partial x}(\dot{\theta} x + \xi_1) + \frac{\partial k_0}{\partial x}(\xi_1 - k_0) \dot{\theta} x - \frac{\partial k_0}{\partial \theta} (\xi_1 - k_0) \dot{\theta} \right)
\]

\[
\leq -x^2 + \dot{\theta}^2 - (\xi_1 - k_0)^2 + (\xi_1 - k_0) \left( u + x + (\xi_1 - k_0) - \frac{\partial k_0}{\partial x}(\dot{\theta} x + \xi_1) \right)
\]

\[
+ \left( \frac{\partial k_0}{\partial x} \right)^2 (\xi_1 - k_0)^2 x^2 + \frac{1}{4} \dot{\theta}^2 + \left( \frac{\partial k_0}{\partial \theta} \right)^2 (\xi_1 - k_0)^2 x^2 + \frac{1}{4} \dot{\theta}^2
\]

\[
= -x^2 - (\xi_1 - k_0)^2 + \frac{5}{4} \dot{\theta}^2 + \frac{1}{4} \dot{\theta}^2
\]

\[
+ (\xi_1 - k_0) \left( u + x + (\xi_1 - k_0) - \frac{\partial k_0}{\partial x}(\dot{\theta} x + \xi_1) + \left( \frac{\partial k_0}{\partial x} \right)^2 (\xi_1 - k_0) x^2 + \left( \frac{\partial k_0}{\partial \theta} \right)^2 (\xi_1 - k_0) \right).
\]

By setting

\[
u := k_1(x, \dot{\theta}, \xi_1) := -x - (\xi_1 - k_0) + \frac{\partial k_0}{\partial x}(\dot{\theta} x + \xi_1) - \left( \frac{\partial k_0}{\partial x} \right)^2 (\xi_1 - k_0) x^2 - \left( \frac{\partial k_0}{\partial \theta} \right)^2 (\xi_1 - k_0)
\]

we get

\[
\dot{V}_1(x, \dot{\theta}, \xi_1) \leq -x^2 - (\xi_1 - k_0)^2 + \frac{5}{4} \dot{\theta}^2 + \frac{1}{4} \dot{\theta}^2.
\]

(4)

**Step 2.** Consider the system

\[
\dot{x} = \theta x + \xi_1,
\]

\[
\dot{\xi}_1 = \xi_2
\]

\[
\dot{\xi}_2 = u.
\]

Let \( V_2 : \mathbb{R}^4 \to [0, \infty) \) be defined by

\[
V_2(x, \dot{\theta}, \xi_1, \xi_2) := V_1(x, \dot{\theta}, \xi_1) + \frac{1}{2} (\xi_2 - k_1(x, \dot{\theta}, \xi_1))^2.
\]

Its derivative along the state trajectory is

\[
\dot{V}_2(x, \dot{\theta}, \xi_1, \xi_2)
\]

\[
= x(\theta x + \xi_1) + (\xi_1 - k_0) \left( \xi_2 - \frac{\partial k_0}{\partial x}(\theta x + \xi_1) - \frac{\partial k_0}{\partial \theta} \dot{\theta} \right) + (\xi_2 - k_1) \left( u - \frac{\partial k_1}{\partial x}(\theta x + \xi_1) - \frac{\partial k_1}{\partial \theta} \dot{\theta} - \frac{\partial k_1}{\partial \xi_1} \xi_2 \right)
\]

\[
= x(\theta x + k_0) + (\xi_1 - k_0) \left( k_1 + x - \frac{\partial k_0}{\partial x}(\theta x + \xi_1) - \frac{\partial k_0}{\partial \theta} \dot{\theta} \right)
\]

\[
+ (\xi_2 - k_1) \left( u + (\xi_1 - k_0) - \frac{\partial k_1}{\partial x}(\theta x + \xi_1) - \frac{\partial k_1}{\partial \theta} \dot{\theta} - \frac{\partial k_1}{\partial \xi_1} \xi_2 \right)
\]

Then by (3) and (4),

\[
\dot{V}_2(x, \dot{\theta}, \xi_1, \xi_2) \leq -x^2 - (\xi_1 - k_0)^2 + \frac{5}{4} \dot{\theta}^2 + \frac{1}{4} \dot{\theta}^2 + (\xi_2 - k_1) \left( u + (\xi_1 - k_0) - \frac{\partial k_1}{\partial x}(\theta x + \xi_1) - \frac{\partial k_1}{\partial \theta} \dot{\theta} - \frac{\partial k_1}{\partial \xi_1} \xi_2 \right)
\]

\[
= -x^2 - (\xi_1 - k_0)^2 + \frac{5}{4} \dot{\theta}^2 + \frac{1}{4} \dot{\theta}^2 + (\xi_2 - k_1) \left( u + (\xi_1 - k_0) - \frac{\partial k_1}{\partial x}(\dot{\theta} x + \xi_1) - \frac{\partial k_1}{\partial \xi_1} \xi_2 \right)
\]

\[
+ \frac{\partial k_1}{\partial x}(\xi_2 - k_1) \dot{x} - \frac{\partial k_1}{\partial \theta}(\xi_2 - k_1) \dot{\theta}
\]
Therefore,
\[
\dot{V}_2(x, \dot{\theta}, \xi_1, \xi_2) \leq -x^2 - (\xi_1 - k_0)^2 + 5 \frac{1}{4} \dot{\theta}^2 + 1 \frac{1}{4} \dot{\theta}^2 - (\xi_2 - k_1)^2 \\
+ (\xi_2 - k_1) \left( u + (\xi_1 - k_0) + (\xi_2 - k_1) - \frac{\partial k_1}{\partial x} (\dot{\theta} x + \xi_1) - \frac{\partial k_1}{\partial \xi_1} \xi_2 \right) \\
+ \left( \frac{\partial k_1}{\partial x} \right)^2 (\xi_2 - k_1) x^2 + 1 \frac{3}{4} \dot{\theta}^2 + 1 \frac{1}{2} \dot{\theta}^2 - (\xi_2 - k_1)^2 + 1 \frac{1}{4} \dot{\theta}^2 \\
= -x^2 - (\xi_1 - k_0)^2 - (\xi_2 - k_1)^2 + 3 \frac{3}{2} \dot{\theta}^2 + 1 \frac{1}{2} \dot{\theta}^2 + (\xi_2 - k_1) \left( u + (\xi_1 - k_0) \\
+ (\xi_2 - k_1) - \frac{\partial k_1}{\partial x} (\dot{\theta} x + \xi_1) - \frac{\partial k_1}{\partial \xi_1} \xi_2 + \left( \frac{\partial k_1}{\partial x} \right)^2 (\xi_2 - k_1) x^2 + \left( \frac{\partial k_1}{\partial \theta} \right)^2 (\xi_2 - k_1) \right).
\]

By setting
\[
u := k_2(x, \dot{\theta}, \xi_1, \xi_2)
:= - (\xi_1 - k_0) - (\xi_2 - k_1) + \frac{\partial k_1}{\partial x} (\dot{\theta} x + \xi_1) + \frac{\partial k_1}{\partial \xi_1} \xi_2 - \left( \frac{\partial k_1}{\partial x} \right)^2 (\xi_2 - k_1) x^2 - \left( \frac{\partial k_1}{\partial \theta} \right)^2 (\xi_2 - k_1)
\]
we get
\[
\dot{V}_2(x, \dot{\theta}, \xi_1, \xi_2) \leq -x^2 - (\xi_1 - k_0)^2 - (\xi_2 - k_1)^2 + 3 \frac{3}{2} \dot{\theta}^2 + 1 \frac{1}{2} \dot{\theta}^2 \\
\leq -\alpha \left( \left\| \begin{array}{c} x \\ \xi_1 \\ \xi_2 \end{array} \right\| \right) + \gamma \left( \left\| \begin{array}{c} \dot{\theta} \\ \ddot{\theta} \end{array} \right\| \right)
\]
for suitable \( \alpha, \gamma \in K_\infty \). Hence the closed-loop system is ISS with respect to \((\dot{\theta}, \ddot{\theta})\).
1 Problem 1 - Mass Spring Damper

We implement a state-space approach. This is complementary to the transfer function approach. You should try to do both (see previous Simulink figures). See Chapter 3 of Adaptive Control Design & Analysis\(^1\) by Gang Tao [GT] for details which lays out a SS approach for both normalized gradient and least squares. Also we don’t want to write routines to handle transfer functions in Python.

Let us first define two mass functions since clearly there is a part of the question that requires the mass in the spring damper system to change with time.

```python
In [1]: from scipy.integrate import odeint # ODE integrator
    import matplotlib.pyplot as plt # Plotting routines
    from matplotlib import rcParams # Modify plot defaults
    import numpy as np # Numerics
    from functools import partial

def m1(t):
    return 20.0

def m2(t):
    if 0 <= t <= 20:
        return 20.0
    else:
        return 20*(2 - np.exp(-0.01*(t-20)))
```

Recall the state-space form of the mass-spring damper equation:

```python
In [2]: def spring_damper_dyn(x, t, m, beta, k, u):
    A = [[0,1],[-k/m(t), -beta/m(t)]
    B = [[0],[1/m(t)]
    return A @ x + np.squeeze(np.asarray(B)*u(t))
```

We choose the stable polynomial \( \Lambda(s) = s^2 + 3s + 2 \) in Eq. 3.4 in [GT]. Then Eq. 3.5 is for us:

\[
\theta^* = \begin{bmatrix}
1/m & 2 - k/m & 3 - \beta/m
\end{bmatrix}^T
\]

\(^1\)Available on campus VPN: [https://vufind.carli.illinois.edu/vf-uiu/Record/uiu_7773512](https://vufind.carli.illinois.edu/vf-uiu/Record/uiu_7773512)
Therefore our regressor dynamics (Eq. 3.13-14) are determined by the matrices:

\[
A_{\lambda} = \begin{bmatrix} 0 & 1 \\ -2 & -3 \end{bmatrix}, \quad b = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad C_{m} = \begin{bmatrix} 1 & 0 \end{bmatrix}
\]

We define these now. The \@ notation simply means matrix multiplication and is available from Python 3.7 onwards. Otherwise you can write np.matmul but that would be much more verbose in the following. \@ and the map() function (coming later) are the only new functions introduced this time. You have already seen all the others; so comments and explanations are minimal.

```
In [3]: def omega1_dyn(x, t, u):
    A = [[0,1],[-2, -3]]
    b = [[0],[1]]
    return A @ x + np.squeeze(np.asarray(b)*u(t))

def omega2_dyn(x, y):
    A = [[0,1],[-2, -3]]
    b = [[0],[1]]
    return A @ x + np.squeeze(np.asarray(b)*y)

def phi(w1, w2):
    C = [[1, 0]]
    return np.concatenate((C @ w1[:,None], w2[:,None]))

def theta_star(m, t, beta, k):
    return np.squeeze(np.asarray([[1/m(t)], [2-k/m(t)], [3-beta/m(t)]]))
```

Here \texttt{theta\_star} is a function representing the vector of true parameters which is possibly time varying. The normalized gradient update equation is given in Eq 3.23 of [GT] and we implement that.

```
In [4]: def epsilon(m, t, beta, k, theta, phi):
    return (theta - np.squeeze(np.asarray(theta_star(m, t, beta, k)))) @ phi

def m_square(fi, P=None):
    if P is None:
        return 1 + fi.T @ fi
    else:
        return 1 + fi.T @ P @ fi

def theta_dyn(x, t, gain, w1, w2, m, beta, k):
    fi = phi(w1, w2)
    numerator = -gain @ fi * epsilon(m, t, beta, k, x, fi)
    return np.squeeze(numerator/m_square(fi))
```
Looking ahead, we see Eq. 3.30-3.32 have similar form to the gradient algorithm. So the
dynamics that are common we write in a different function.

In [5]: def common_dynamics(t, x, w1, w2, theta, m, beta, k, u, gain):
   
   d_x = spring_damper_dyn(x, t, m, beta, k, u)
   d_w1 = omega1_dyn(w1, t, u)
   d_w2 = omega2_dyn(w2, x[0])
   d_theta = theta_dyn(theta, t, gain, w1, w2, m, beta, k)
   return d_x, d_w1, d_w2, d_theta

At this point we are ready to do the gradient algorithm. The ‘nga’ at the end is for normalized
gradient algorithm.

In [6]: def all_dynamics_nga(z, t, gain, m, beta, k, u):
   
   x, w1, w2, theta = extract(z)
   d_x, d_w1, d_w2, d_theta = common_dynamics(t,
   x, w1, w2, theta,
   m, beta, k, u, gain)
   return np.concatenate((d_x,d_w1, d_w2, d_theta))

Like last time we need an extractor function.

In [7]: def extract(x, lsr=False):
   
   spring_states, omega1_states = x[:2], x[2:4]
   omega2_states, theta_states = x[4:6], x[6:9]
   if lsr:
       P = np.apply_along_axis(np.reshape,
       0, arr=x[9:100], newshape=(3,3))
       return spring_states, omega1_states,
       omega2_states, theta_states, P
   else:
       return spring_states, omega1_states,
       omega2_states, theta_states

We need one last function to extract the parameters \(m, \beta, k\) from from the estimated \(\theta\).

In [8]: def undo(theta):
   
   theta[0] = 1 / theta[0]
   theta[1] = np.multiply(2 - theta[1], theta[0])
   theta[2] = np.multiply(3 - theta[2], theta[0])

1.1 Gradient algorithm with fixed mass

We will choose to simulate with a two sinusoidal inputs. Note that in all the definitions above
we have treated \(u\) like a place holder. The only restriction placed on it is that it must be able to
implement \(u(t)\) when the time comes.

In [9]: def u(t):
   
   return 100*np.sin(np.pi*t) - 100*np.cos(np.sqrt(5) *t)
k = 5
beta = 0.1
states = np.zeros(6)
params = np.asarray([1,1,1])  # For consistency
initial = np.concatenate((states, params))
t = np.linspace(0,50)
gain = np.diag([3, 10, 10])
sols = odeint(all_dynamics_nga, initial,
        t, args=(gain, m1, beta, k, u))
_, _, _, theta = extract(sols.T)

The _ means we don’t care about those variables. After playing around with the gains and the
initial conditions we get a nice plot without too many oscillations or large peaks that drown out
other data. Lets plot that.

In [10]: plt.figure(figsize=(12,6))
plt.subplot(1,2,1)
plt.plot(t, (theta - theta_star(m1, t, beta,k)[:,np.newaxis]).T)
plt.grid()
plt.title('Gradient Algorithm (mass fixed): $\theta^* - \theta$')
ax=plt.subplot(1,2,2)
undo(theta)
plt.plot(t, theta.T, label='Estimate')
plt.title('Grad. Alg. (mass fixed): System parameters')
ax.axhline(y=20, color='k', linestyle='--', linewidth=0.8)
ax.axhline(y=5, color='k', linestyle='--', linewidth=0.8)
ax.axhline(y=0.1, color='k', linestyle='--', linewidth=0.8)
plt.grid()
plt.show()

(see plot in next page)
We see that the system parameters are estimated pretty quickly pretty accurately, the blue line converges to \(~20\) kg, the yellow line converges to \(5\) which is the value of the spring constant and so on.

### 1.2 Gradient algorithm with changing mass.

Now we allow the mass to change. But this is almost no extra work. We just change the \(m_1\) to \(m_2\) in the call to \texttt{odeint} and using some partial function magic, we get the time-varying \texttt{theta_star_time} vector which computes the the real parameters as it varies in time. The comment about the gains from last time still applies

In \[11\]:

```ipython
sols = odeint(all_dynamics_nga, initial, t,
              args=(gain, m2, beta, k, u))
_, _, _, theta = extract(sols.T)
theta_star_time = np.asarray(list(map(partial(theta_star, m2,
                                            beta=beta, k=k), t)))
```

```ipython
plt.figure(figsize=(12,6))
plt.subplot(1,2,1)
plt.plot(t, (theta.T - theta_star_time))
plt.title('Gradient Algorithm (mass fixed): $\theta^* - \theta$
          )
plt.grid()
plt.subplot(1,2,2)
undo(theta_star_time.T)
do(theta)
plt.plot(t, theta_star_time, label='Real time')
plt.plot(t, theta.T, label='Estimate')
plt.legend()
```
The tracking looks good. `map` above is a Python built-in function that takes a list/iterable and applies a function to each item in it. But everything in Python is an object and sometimes Python is lazy\(^2\) and just makes a `map` object. The `list(map(...)` forces it to perform the map computation.

Finally you may notice we didn’t assign anything when we called `undo` that is because `undo` didn’t return anything. This shows that function calls in Python are a little different\(^3\).

Now let's do the least squares algorithm.

### 1.3 Least squares with fixed mass

Compared to everything we have set up, if we look at Eq 3.30 we really only need to define a function for the dynamics of \(P\).

In [12]:
```
def P_dynamics(P, w1, w2):
    fi = phi(w1, w2)
    to_return = -P @ fi @ fi.T @ P / m_square(fi, P)
    return np.squeeze(np.reshape(to_return, (9,)))
```

```
def all_dynamics_lsa(z, t, m, beta, k, u):
x, w1, w2, theta, P = extract(z, lsr=True)
d_x, d_w1, d_w2, d_theta = common_dynamics(t, x, w1, w2, theta,
```

\(^2\)https://en.wikipedia.org/wiki/Lazy_evaluation

\(^3\)https://robertheaton.com/2014/02/09/pythons-pass-by-object-reference-as-explained-by-philip-k-dick/
\[ m, \beta, k, u, P \]
\[
d_P = P_{\text{dynamics}}(P, w_1, w_2)
\]
\[
\text{return } \text{np.concatenate}((d_x, d_w_1, d_w_2, d_{\theta}, d_P))
\]

Now we do have one less argument because the gain itself is part of the states being passed to `odeint`. Let's integrate the ODE.

**In [13]:**
\[
k = 5
\]
\[
\beta = 0.1
\]
\[
\text{states} = \text{np.zeros}(2)
\]
\[
w_0 = \text{np.random.rand}(4)
\]
\[
\text{params} = \text{np.asarray}([1,1,1])
\]
\[
gain = \text{np.reshape}(\text{np.diag}([100, 100, 200]), (9,))
\]
\[
\text{initial} = \text{np.concatenate}((\text{states}, w_0, \text{params}, \text{gain}))
\]
\[
t = \text{np.linspace}(0,30,100)
\]
\[
sols = \text{odeint}(\text{all_dynamics_lsa}, \text{initial}, t,
\]
\[
\text{args}=(m1, \beta, k, u))
\]
\[
_, _, _, \theta, _ = \text{extract}(\text{sols.T}, \text{lsr=True})
\]

The noticeable difference is \( w_0 \), which we explicitly initialize to non-zero values because it is involved in the construction of \( \phi \) vector.

**In [14]:**
\[
\text{plt.figure}() \quad \text{(see plot in next page)}
\]
1.4 Least squares with changing mass

Implement yourself! Almost all the code pieces required are already defined.
Problem 2 (optional)

1 Problem 2

We start as usual by making the necessary imports

In [1]: from scipy.integrate import odeint # ODE integrator
   : import matplotlib.pyplot as plt # Plotting routines
   : import numpy as np # Numerics
   : from functools import partial

Now we implement the dynamical equations. We know in the case of loss of stabilizability due to high gain, we want to be able to implement projection, so we bake this functionality in.

In [2]: def plant(y, t, u):
   : return a*y + b*u

   def model(ym, t, r):
   : return -am*ym + bm*r

   def param_estimate(gamma, y, ym, u, bhat, projection, val=0.2):
   : d_ahat = -gamma *(ym - y)*y
   : d_bhat = -gamma * (ym - y) * u
   : if projection:
   :     if bhat >= val:
   :         pass
   :     else:
   :         d_bhat = 0
   : return [d_ahat, d_bhat]

Next we define the controller:

In [3]: def controller(ahat, am, bhat, bm, y):
   : khat = (ahat + am)/bhat
   : lhat = bm/b
   : return -khat *y + lhat * r

We concatenate all the dynamics into one function

In [4]: def all_dynamics(x, t, gamma, projection=False):
   : x_plant, x_model, params = extract(x)
   : x_ahat, x_bhat = params
\[ u = \text{controller}(x_{\hat{a}}, a_m, x_{\hat{b}}, b_m, x_{\text{plant}}) \]
\[ dx_{\text{plant}} = \text{plant}(x_{\text{plant}}, t, u) \]
\[ dx_{\text{model}} = \text{model}(x_{\text{model}}, t, r) \]
\[ d_{\hat{a}}, d_{\hat{b}} = \text{param\_estimate}(\gamma, x_{\text{plant}}, x_{\text{model}}, u, x_{\hat{b}}, \text{projection}) \]
\[ \text{return } dx_{\text{plant}}, dx_{\text{model}}, d_{\hat{a}}, d_{\hat{b}} \]

As before we have an exfil function to rescue variables of interest from the long vector.

\textbf{In [5]:} \texttt{def extract(x):}
\begin{verbatim}
      x_plant = x[0]
x_model = x[1]
      params = (x[2], x[3])
      return x_plant, x_model, params
\end{verbatim}

And we need a function to plot the results multiple times:

\textbf{In [6]:} \texttt{def make\_plots(sol, suptitle):}
\begin{verbatim}
x_plant, x_model, params = extract(sol.T)
x_{\hat{a}}, x_{\hat{b}} = params
fig, axs = plt.subplots(1,3, figsize=(12,4))
labels = [('x', 'x_m'), 'Error', ('\hat{a}', '\hat{b}')]
titles = ['States', 'Error', 'Control Gains']
items = [(x_plant, x_model), x_plant-x_model, (x_{\hat{a}}, x_{\hat{b}})]

for k, (ax, data, label, title) in enumerate(zip(axs,items, labels, titles)):  
if k == 1:
   ax.plot(t, data, label = label)
else:
   (d1, d2), (s1, s2) = data, label
   ax.plot(t, d1, label=s1)
   ax.plot(t, d2, label=s2)
   ax.legend()
   ax.set_title(title)
plt.suptitle(suptitle)
plt.show()
\end{verbatim}

A couple of Python’s default functions sneaked in there, but by now you should be able to figure out what is going on.

### 1.1 Part (a)

We see that if we have small gain and and \( \hat{b}(0) \) is initialized correctly then everything works out well.

\textbf{In [7]:} \texttt{a, b, r = 5, 3, 10}
\begin{verbatim}
am, bm = 3, 1
\end{verbatim}
init = np.random.rand(4)

t = np.linspace(0, 5, 100)
init[3] = 1  # Sets b(0) = 1
gamma = 0.1

sols = odeint(all_dynamics, init, t, args = (gamma, ))
make_plots(sols, 'Correct sign - small gain')

But if the gain is relatively large then we have loss of stabilizability and scipy.odeint complains at us:

In [8]: t = np.linspace(0, 1, 100)
init[3] = 5
gamma = 7
sols = odeint(all_dynamics, init, t, args = (gamma, ))
make_plots(sols, 'Correct sign - large gain')

Run with full_output = 1 to get quantitative information.
We can call on projection to see if it helps us.

In [9]: init = np.random.rand(4)
   ...: init[3] = 5
   ...: gamma = 10
   ...: t = np.linspace(0, 10, 100)
   ...: sols = odeint(partial(all_dynamics, projection=True), init, t, args = (gamma, ))
   ...: make_plots(sols, 'Correct sign - large gain with projection')

1.2 Part (b)

When \( \hat{b}(0) \) is initialized with the wrong sign, then there is always loss of stabilizability and projection does not help in this case:

In [10]: init = np.random.rand(4)
   ...: gamma = 10
\[ t = \text{np.linspace}(0, 1, 100) \]
\[ \text{sols} = \text{odeint(partial(all_dynamics, projection=True), init, t, args = (gamma, ))} \]
\[ \text{make_plots(sols, 'Correct sign - large gain with projection')} \]