Numerical Analysis **Spring 2025**

Term Project

Instructor: **Kwangki Kim**

Class :

Name :

Student ID# :

Autonomous Systems Laboratory
Dept. of EE, Inha University

Problem 1. [*(Control)*] Let us consider this section with a tiny example from the optimal control theory. Optimal control deals with the problem of finding a control law for a given system such that a certain optimality criterion is achieved. This phrase is too unspecific, let us illustrate it. Imagine that we have a car that advances with some speed, say 0.5m/s. The goal is to accelerate and reach, say, 2.3m/s. We cannot control the speed directly, but we can act on the acceleration via the gas pedal. We can model the system with a very simple equation:

$$v_{i+1} = v_i + u_i,$$

where the signals are sampled every 1 second, v_i is the car speed and u_i is the acceleration of the car. Let us say that we have half a minute to reach the given speed, i.e, $v_0 = 0.5$m/s, $v_n = 2.3$m/s, $n = 30$s. So, we need to find $\{u_i\}_{i=0}^{n-1}$ that optimizes some quality criterion $J(\vec{v}, \vec{u})$:

$$\min J(\vec{v}, \vec{u}) \quad s.t.\ v_{i+1} = v_i + u_i = v_0 + \sum_{j=0}^{i-1} u_j \ \ \forall i \in 0..n-1$$

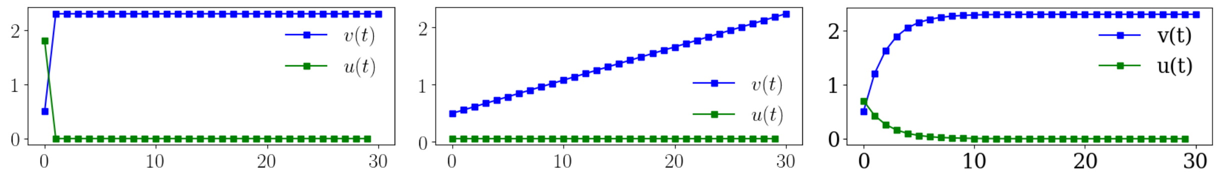
The case where the system dynamics are described by a set of differential equations and the cost is described by a quadratic functional is called a linear quadratic problem. Let us test few different quality criteria. What happens if we ask for the car to reach the final speed as quickly as possible? It can be written as follows:

$$J(\vec{v}, \vec{u}) := \sum_{i=1}^{n} (v_i - v_n)^2 = \sum_{i=1}^{n} \left(\sum_{j=0}^{i-1} u_j - v_n + v_0 \right)^2$$

To minimize this criterion, we can solve the following system in the least squares sense:

$$\begin{cases} u_0 & & & & = v_n - v_0 \\ u_0 & +u_1 & & & = v_n - v_0 \\ \vdots & & \ddots & & \vdots \\ u_0 & +u_1 & \dots & +u_{n-1} & = v_n - v_0 \end{cases}$$

Figure: 1D optimal control problem. **Left:** lowest settling time goal; **middle:** lowest control signal goal; **right:** a trade-off between the control signal amplitude and the settling time.



Following listing solves the system:

```python
import numpy as np
n,v0,vn = 30,0.5,2.3
A = np.matrix(np.tril(np.ones((n,n))))
b = np.matrix([[vn-v0]]*n)
u = np.linalg.inv(A.T*A)*A.T*b
v = [v0 + np.sum(u[:i]) for i in range(0,n+1)]
```

The resulting arrays $\{u_i\}_{i=0}^{n-1}$ and $\{v_i\}_{i=0}^{n}$ are shown in the leftmost image of Figure 1. The solution is obvious: $u_0 = v_n - v_0$, $u_i = 0\ \forall i > 0$, so in this case the system reaches the final state in one time-step, and it is clearly physically impossible for a car to produce such an acceleration.

Okay, no problem, let us try to penalize large accelerations:

$$J(\vec{v},\vec{u}) := \sum_{i=0}^{n-1} {u_i}^2 + \left(\sum_{i=0}^{n-1} u_i - v_n + v_0\right)^2$$

Minimization of this criterion is equivalent to solving the following system in the least squares sense:

$$\begin{cases} u_0 & & & & = 0 \\ & u_1 & & & = 0 \\ & & \ddots & & \vdots \\ & & & u_{n-1} & = 0 \\ u_0 & +u_1 & \dots & +u_{n-1} & = v_n - v_0 \end{cases}$$

Following listing solves this system, and the resulting arrays are shown in the middle image of Figure 1.

```python
import numpy as np
n,v0,vn = 30,0.5,2.3
A = np.matrix(np.vstack((np.diag([1]*n), [1]*n)))
b = np.matrix([[0]]*n + [[vn-v0]])
u = np.linalg.inv(A.T*A)*A.T*b
v = [v0 + np.sum(u[:i]) for i in range(0,n+1)]
```

This criterion indeed produces low acceleration, however the transient time becomes unacceptable.

Minimization of the transient time and low acceleration are competing goals, but we can find a trade-off by mixing both goals:

$$J(\vec{v},\vec{u}) := \sum_{i=1}^{n}(v_i - v_n)^2 + 4\sum_{i=0}^{n-1} u_i^2 = \sum_{i=1}^{n}\left(\sum_{j=0}^{i-1} u_j - v_n + v_0\right)^2 + 4\sum_{i=0}^{n-1} u_i^2$$

This criterion asks to reach the goal as quickly as possible, while penalizing large accelerations. It can be minimized by solving the following system:

$$\begin{cases} u_0 & & & & = v_n - v_0 \\ u_0 & +u_1 & & & = v_n - v_0 \\ \vdots & & \ddots & & \vdots \\ u_0 & +u_1 & \dots & +u_{n-1} & = v_n - v_0 \\ 2\,u_0 & & & & = 0 \\ & 2\,u_1 & & & = 0 \\ & & \ddots & & \vdots \\ & & & 2\,u_{n-1} & = 0 \end{cases}$$

Note the coefficient 2 in the equations $2u_i = 0$ and recall that we solve the system in the least squares sense. By changing this coefficient, we can attach more importance to one of the competing goals.

Following listing solves this system, and the resulting arrays are shown in the right image of Figure 1.

```
import numpy as np
n,v0,vn = 30,0.5,2.3
A = np.matrix(np.vstack((np.tril(np.ones((n,n))), np.diag([2]*n))))
b = np.matrix([[vn-v0]]*n + [[0]]*n)
u = np.linalg.inv(A.T*A)*A.T*b
v = [v0 + np.sum(u[:i]) for i in range(0,n+1)]
```

Note that the signal $u(t)$ is equal to the signal $v(t)$ up to a multiplication by a constant gain:

$$u(t) = -F(v(t) - v_{\text{goal}}),$$

This gain is necessary to know in order to build a closed-loop regulator, and it can be computed from the cost function J, defined previously as

$$J(\vec{v}, \vec{u}) := \sum_{i=0}^{n-1} {u_i}^2 + \left(\sum_{i=0}^{n-1} u_i - v_n + v_0 \right)^2$$

In practice, just like we did in this section, engineers try different combinations of competing goals until they obtain a satisfactory transient time while not exceeding regulation capabilities.

(a) Rewrite all the codes for this problem by yourself.

(b) Analyze the problem and solution by yourself.

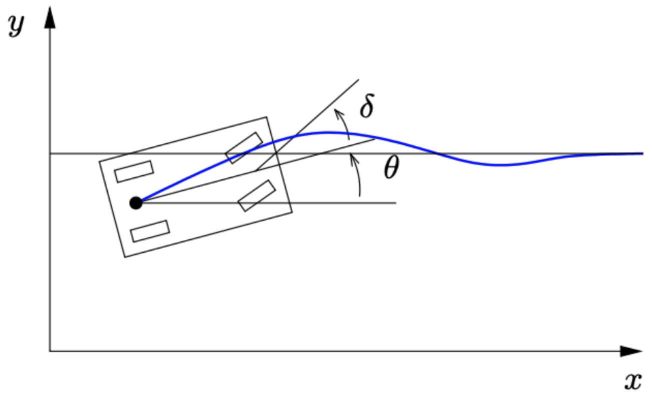
Problem 2. [*(Control)*]

Vehicle steering dynamics The vehicle dynamics are given by a simple bicycle model:

$$\begin{aligned} \dot{x} &= \cos\theta v \\ \dot{y} &= \sin\theta v \qquad \Leftrightarrow \qquad \dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}) \\ \dot{\theta} &= \frac{v}{l}\tan\delta \end{aligned}$$

We take the state of the system as $\boldsymbol{x} = (x, y, \theta)$ where (x, y) is the position of the vehicle in the plane and θ is the angle of the vehicle with respect to horizontal. The vehicle input is given by $\boldsymbol{u} = (v, \delta)$ where v is the forward velocity of the vehicle and δ is the angle of the steering wheel. The model does not include saturation of the vehicle steering angle.

Figure: Vehicle steering dynamics.



- Linearize:
$$\dot{\delta\boldsymbol{x}} = \boldsymbol{A}\delta\boldsymbol{x} + \boldsymbol{B}\delta\boldsymbol{u} + \boldsymbol{c}$$
where the Jacobians are given as
$$\boldsymbol{A} = \left.\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right|_{(\bar{\boldsymbol{x}},\bar{\boldsymbol{u}})}, \quad \boldsymbol{B} = \left.\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}}\right|_{(\bar{\boldsymbol{x}},\bar{\boldsymbol{u}})}, \quad \boldsymbol{c} = \boldsymbol{f}(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})$$
with the nominal state-input pair $(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})$.
- Time-discretization (Euler-forward)[1]:
$$\begin{aligned} \delta\boldsymbol{x}_{t+1} &= (\boldsymbol{I} + \delta t\boldsymbol{A})\delta\boldsymbol{x}_t + \delta t\boldsymbol{B}\delta\boldsymbol{u} + \delta t\boldsymbol{c} \\ &= \boldsymbol{A}_d\delta\boldsymbol{x}_t + \boldsymbol{B}_d\delta\boldsymbol{u}_t + \boldsymbol{c}_d \end{aligned}$$
with which
$$\boldsymbol{x}_t = \bar{\boldsymbol{x}} + \delta\boldsymbol{x}_t \quad \text{and} \quad \boldsymbol{u}_t = \bar{\boldsymbol{u}} + \delta\boldsymbol{u}_t\,.$$

[1]Well, I know this is not good, but let's focus on the main idea, not on the technical details of numerical differentiation for now.

(a) Write a Python code to represent

$$\boldsymbol{x}_T = \boldsymbol{F}\boldsymbol{u}_{0:T-1} + \boldsymbol{G}\boldsymbol{x}_0 + \boldsymbol{H}\boldsymbol{c}_d$$

where

$$\boldsymbol{u}_{0:T-1} = \begin{bmatrix} \boldsymbol{u}_0 \\ \boldsymbol{u}_1 \\ \vdots \\ \boldsymbol{u}_{T-1} \end{bmatrix}$$

for $T > 0$ and

$$\bar{\boldsymbol{x}} = \begin{bmatrix} 0 \\ -2 \\ 0 \end{bmatrix}, \quad \bar{\boldsymbol{u}} = \begin{bmatrix} 10 \\ 0 \end{bmatrix}$$

Consider a goal pose

$$\boldsymbol{x}_{\text{goal}} = \begin{bmatrix} 100 \\ 2 \\ 0 \end{bmatrix}$$

with the terminal input

$$\boldsymbol{u}_T = \begin{bmatrix} 10 \\ 0 \end{bmatrix}$$

where $T = 10$. Consider the cost function

$$J(\boldsymbol{x}_0, \boldsymbol{u}_{0:T-1}) = \sum_{t=1}^{T} \|\boldsymbol{x}_t - \boldsymbol{x}_{\text{goal}}\|^2 + \lambda \sum_{t=0}^{T-1} \|\boldsymbol{u}_t\|^2$$

where $\lambda > 0$ refers to a weight factor.

(b) Write a linear least-squares problem for

$$\underset{\boldsymbol{u}_{0:T-1}}{\text{minimize}} \; J(\boldsymbol{x}_0, \boldsymbol{u}_{0:T-1}) .$$

(c) Using Python, solve the least-squares problem you defined in Part (b) with different values of the weight factor $\lambda > 0$.

Problem 3. [*Least-squares Methods (programming part)*] Auto-regressive time series model. Suppose that $z_1, z_2, \ldots$ is a time series. An auto-regressive model (also called AR model) for the time series has the form

$$\hat{z}_{t+1} = \theta_1 z_t + \cdots + \theta_M z_{t-M+1}, \quad t = M, M+1, \ldots$$

where M is the memory or lag of the model. Here $\hat{z}_{t+1}$ is the prediction of z_{t+1} made at time t (when $z_t, \ldots, z_{t-M+1}$ are known). This prediction is a linear function of the previous M values of the time series. With good choice of model parameters, the AR model can be used to predict the next value in a time series, given the current and previous M values. This has many practical uses.

We can use least squares (or regression) to choose the AR model parameters, based on the observed data $z_1, \ldots, z_T$, by minimizing the sum of squares of the prediction errors $z_t - \hat{z}_t$ over $t = M+1, \ldots, T$, i.e.,

$$(z_{M+1} - \hat{z}_{M+1})^2 + \cdots + (z_T - \hat{z}_T)^2$$

(We must start the predictions at $t = M+1$, since each prediction involves the previous M time series values, and we do not know $z_0, z_{-1}, \ldots$.)

The AR model can be put into the general linear in the parameters model form by taking

$$y^{(i)} = z_{M+i}, \quad x^{(i)} = (z_{M+i-1}, \ldots, z_i), \quad i = 1, \ldots, T-M$$

We have $N = T - M$ examples, and $n = M$ features.

Consider the time series of hourly temperature data given in the file

`temperature_data.ipynb`

with length $31 \times 24 = 744$.

(a) Fit an AR model with memory $M = 8$ using least squares, with $N = 31 \times 24 - 8 = 736$ samples.

(b) What is the RMS error of this predictor?

$$\text{RMS}(e) = \sqrt{\frac{\sum_{t=9}^{744} (z_t - \hat{z}_t)^2}{N}}$$

(c) Repeat the computations of Parts (a) and (b), i.e., (a) Model fitting and (b) Computation of the RMS error, for $M = 4, 8, 12, 24$. Plot M vs. RMS.

Problem 4. [*Least-squares (programming)*] Consider the power conversion relation

$$P_{\text{out}} = P_{\text{in}} + P_{\text{loss}} = \eta_{\text{effc}} P_{\text{in}}$$

where $P_{\text{out}} = P_{\text{mech}}$ is the mechanical power delivered (for $P_{\text{mech}} > 0$) by a motor or the mechanical power supplied (for $P_{\text{mech}} < 0$) to a motor, and $P_{\text{in}} = P_{\text{elec}}$ is the electrical power supplied to a motor from a battery (for $P_{\text{elec}} > 0$) or generated by a motor and supplied to a battery (for $P_{\text{elec}} < 0$).

The power conversion efficiency, from electrical power to mechanical power when $P_{\text{elec}} > 0$ (traction mode) and from mechanical power to electrical power when $P_{\text{elec}} < 0$ (generator mode), $\eta_{\text{effc}} \in (0,1)$ is considered to be a function of motor speed ω_m [rad/sec] and T_m [N·m]. In other words,

$$\eta_{\text{effc}} = g(\omega_m, T_m)$$

for some function $g : \mathbb{R} \times \mathbb{R} \to (0,1)$.

Consider the data file

`motor_efficiency_data.mat`

There are two variables,

- `Data_regen_braking`$\in \mathbb{R}^{170\times 3}$ and
- `Data_traction`$\in \mathbb{R}^{180\times 3}$.

In the variables `Data_regen_braking` and `Data_traction`, the first column is the power conversion efficiency η_{effc}, the second column is the motor speed ω_m [rad/sec], and the third column is the motor torque T_m [N·m].

In this problem, we want to determine a (regression) model for P_{elec} as a function of (ω_m, T_m). In other words,

$$P_{\text{elec}} = f(\omega_m, T_m) = \begin{cases} \dfrac{T_m \omega_m}{\eta_{\text{effc}}} & \text{for } T_m > 0\,, \\ T_m \omega_m \eta_{\text{effc}} & \text{for } T_m < 0\,, \end{cases}$$

where $P_{\text{mech}} = T_m \omega_m$.

(a) Consider a parameterized linear regression with the following set of basis functions $\{b_k(\omega_m, T_m)\}_{k=1}^{7}$:

$$\{1, \omega_m, \omega_m^2, \omega_m T_m, T_m^2, \omega_m T_m^2, \omega_m^2 T_m^2\}\,.$$

Determine the parameters of approximation models

$$\hat{f}^{+}(\omega_m, T_m) = \sum_{k=1}^{7} c_k^{+} b_k(\omega_m, T_m) \approx \frac{T_m \omega_m}{\eta_{\text{effc}}(\omega_m, T_m)} \quad \text{for } T_m > 0$$

and

$$\hat{f}^{-}(\omega_m, T_m) = \sum_{k=1}^{7} c_k^{-} b_k(\omega_m, T_m) \approx T_m \omega_m \eta_{\text{effc}}(\omega_m, T_m) \quad \text{for } T_m < 0\,,$$

via the least-squares method:

$$\hat{c}^{+} := \arg\min \sum_{i=1}^{N_s^{+}} \left| \hat{f}^{+}(\omega_m^{(i)}, T_m^{(i)}) - \frac{T_m^{(i)} \omega_m^{(i)}}{\eta_{\text{effc}}(\omega_m^{(i)}, T_m^{(i)})} \right|^2$$

and

$$\hat{c}^{-} := \arg\min \sum_{i=1}^{N_s^{-}} \left| \hat{f}^{-}(\omega_m^{(i)}, T_m^{(i)}) - T_m^{(i)} \omega_m^{(i)} \eta_{\text{effc}}(\omega_m^{(i)}, T_m^{(i)}) \right|^2 ,$$

where $N_s^{+} = 180$ and $N_s^{-} = 170$ are the number of experimental data obtained for the traction mode and the regenerative braking mode, respectively. (They are indeed obtained from a real electric motor of a commercial PHEV, Hyundai Ioniq PHEV.)

(b) Apply the regularized least-squares method for the problem stated in part (a). Compare the results with varying weights for the least-norm criterion $\lambda \|c\|_2^2$[2].

(c) Apply the kernel methods (non-parametric linear regression). Compare the results with the ones obtained in Parts (a) and (b).

[2]Since the elements of the coefficient vector c have different units and scales, you might apply the following normalized least-norm criterion: For example, let $\hat{c}(0)$ be the optimal solution for the unregularized (i.e., $\lambda = 0$) least-squares problem and use the least-norm criterion $\lambda \|(c - \hat{c}(0))/\hat{c}(0)\|_2^2$ where the operator / implies the elementwise division.

Appendix: scipy.io를 사용하여 python에서 mat 데이터 파일을 읽어서 사용하는 방법
MATLAB 데이터 파일(.mat)을 Python에서 읽어오는 방법은 다음과 같다.

```
>>> from scipy import io
>>> data1 = io.loadmat('engine_efficiency_data') # 엔진 효율 데이터 → data1
>>> data2 = io.loadmat('motor_efficiency_data') # 모터 효율 데이터 → data2
```

변수 이름을 사용하여 각 변수 데이터를 array로 사용할 수 있도록 하기 위해서는 다음의 과정이 필요하다.

```
>>> data1_var1 = data1['변수명11']
>>> data1_var2 = data1['변수명12']
>>> data2_var1 = data2['변수명21']
>>> data2_var2 = data2['변수명22']
```

위의 command를 통해서 array 타입 변수인 data1_var1, ..., data2_var2 이 생성되었음을 확인하고, 이 변수들을 사용하여 원하는 수치연산을 수행하면 된다.

Problem 5. [*Linear system (programming)*]
Consider the so-called Lyapunov equation

$$AX + XB = C$$

where $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{m\times m}$, $C \in \mathbb{R}^{n\times m}$ are given matrices and $X \in \mathbb{R}^{n\times m}$ is the unknown to be found[3].

(a) Make your own **python** function to solve the Lyapunov equation for X when the matrices A, B, and C are given. You might first want to convert the matrix equations into a linear system (see the next page for an example) and solve the linear system to find the elements of the matrix X. You are asked to write a python code in which the input is (A, B, C) and the output (or return) is X of a matrix form with a compatible dimension.

(b) Test your python code with numerical case studies. (If you are taking my class, “Control Systems Design", then use the pole placement examples of state feedback controller design and state observer design in the lecture note.) Show the test results you performed.

(c) Test your code with $A = \begin{bmatrix} 1 & 2 \\ -3 & -4 \end{bmatrix}$, $B = A^\top$, $C = \begin{bmatrix} 3 & 1 \\ 1 & 1 \end{bmatrix}$.

[3]There already exist some numerical solvers for this form of matrix equations:

- For MATLAB, there is a built-in function `lyap` to solve the special and general forms of the Lyapunov equation. You can find more details of the MATLAB built-in function `lyap` in the link https://www.mathworks.com/help/control/ref/lyap.html.
- For Python, a similar function `control.matlab.lyap` from *Python Control Systems Library* can be used to solve the special and general forms of the Lyapunov equation. You can find more details of the Python function `control.matlab.lyap` in the link https://python-control.readthedocs.io/en/0.8.3/generated/control.matlab.lyap.html.

Background on the use of Lyapunov equation for pole-placement

• Lyapunov Equation

$$AX + XB = C$$

where $A \in \mathbb{R}^{n\times n},\ B \in \mathbb{R}^{m\times m},\ C \in \mathbb{R}^{n\times m},\ X \in \mathbb{R}^{n\times m}$
(A, B, C) given $\rightarrow$ solve it for $X \in \mathbb{R}^{n\times m}$
The equation is called the Lyapunov equation for X.

(Example) Consider a case with n = 3 and m = 2 :

$$AX + XB = C$$

$$\Leftrightarrow \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \end{bmatrix} + \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \end{bmatrix}$$

$$\Leftrightarrow \begin{bmatrix} a_{11}+b_{11} & a_{12} & a_{13} & b_{21} & 0 & 0 \\ a_{21} & a_{22}+b_{11} & a_{23} & 0 & b_{21} & 0 \\ a_{31} & a_{32} & a_{33}+b_{11} & 0 & 0 & b_{21} \\ b_{12} & 0 & 0 & a_{11}+b_{22} & a_{12} & a_{13} \\ 0 & b_{12} & 0 & a_{21} & a_{22}+b_{22} & a_{23} \\ 0 & 0 & b_{12} & a_{31} & a_{32} & a_{33}+b_{22} \end{bmatrix} \begin{bmatrix} x_{11} \\ x_{21} \\ x_{31} \\ x_{12} \\ x_{22} \\ x_{32} \end{bmatrix} = \begin{bmatrix} c_{11} \\ c_{21} \\ c_{31} \\ c_{12} \\ c_{22} \\ c_{32} \end{bmatrix}$$

$$\Leftrightarrow \mathcal{A}\mathbf{x} = \mathbf{c}$$

This is indeed a standard linear system (i.e., a system of linear algebraic equations). There are $n \times m = 6$ equations for $n \times m = 6$ unknowns $(x_{11}, x_{21}, x_{31}, x_{12}, x_{22}, x_{32})$.

An application: Pole-placement via solving Lyapunov equation We discuss a method of computing state feedback gain for eigenvalue assignment (i.e. pole placement).

Procedure : Consider controllable (A, B) where $A \in \mathbb{R}^{n\times n},\ B \in \mathbb{R}^{n\times 1}$. We want to find a state feedback controller gain $K \in \mathbb{R}^{1\times n}$ such that $(A - BK)$ has any set of desired eigenvalues which contain no eigenvalues of A.

1. Select an $n \times n$ matrix F with the set of desired eigenvalues where the form of F can be arbitrarily chosen. For example, F can be a modal, observer canonical, or controller canonical form.
2. Select an arbitrary $\bar{K} \in \mathbb{R}^{1\times n}$ such that $(F, \bar{K})$ is observable.
3. Solve the unique $T \in \mathbb{R}^{n\times n}$ in the Lyapunov equation

$$AT - TF = B\bar{K}$$

4. Compute the feedback gain $K := \bar{K}T^{-1}$.

Problem 6. [*Optimization (programming)*]

We continue a linear-equality constrained least-squares problem where the goal is to purchase advertising in n different channels so as to achieve (or approximately achieve) a target set of customer views or impressions in m different demographic groups. We denote the n-vector of channel spending as s; this spending results in a set of views (across the demographic groups) given by the m-vector $v = Rs$.

We want to minimize the sum of squares of the deviation from the target set of views, given by v_{des}. In addition, we fix our total advertising spending, with the constraint $\sum_{i=1}^{n} s_i = \mathbf{1}^\top s = b$, where b is a given total advertising budget and $\mathbf{1} = [1\,1\,\cdots\,1]^\top \in \mathbb{R}^n$ denotes the all-ones n-vector. (This can also be described as allocating a total budget b across the n different channels.) This leads to the constrained least squares problem

$$\begin{array}{ll} \text{minimize} & \|Rs - v_{\text{des}}\|^2 \\ \text{subject to} & \mathbf{1}^\top s = b \end{array}$$

The solution $\hat{s}$ of this problem is not guaranteed to have nonnegative entries, as it must to make sense in this application. But we ignore this aspect of the problem here.

(a) Write down two first-order necessary conditions we studied in the class. Namely, (1) primal condition and (2) primal-dual condition. Both are linear systems.

(b) Make a python code to compute the solution $\hat{s}$ for given $R \in \mathbb{R}^{m\times n}$, $v_{\text{des}} \in \mathbb{R}^m$, and $b \in \mathbb{R}$. Run your code with the data file `advertizing_budget_data.ipynb`.

(c) Compute and compare the RMS errors with different budgets,

$$b = 1200, 1300, \cdots, 1700$$

where the RMS error is defined as

$$\text{RMS} = \frac{\|R\hat{s} - v_{\text{des}}\|}{\sqrt{m}}.$$

For comparisons, plot b vs. RMS.

(Hint!)

[17]:
```
#advertising budget
R = np.matrix([[.97,1.86,.41],[1.23,2.18,.53],[.8,1.24,.62],[1.29,.98,.
 ↪51],[1.1,1.23,.69],[.67,.34,.54],[.87,.26,.62],[1.1,.16,.48],[1.92,.22,.
 ↪71],[1.29,.12,.62]])
m,n = np.shape(R)
vdes = 1e3 * np.ones(m)
s = npl.lstsq(R,vdes)[0]
s
#will be continued in 16 re: how to add a constraint like a total budget
```

```
/Users/kwangkikim/anaconda3/envs/py-numanal/lib/python3.7/site-
packages/ipykernel_launcher.py:5: FutureWarning: `rcond` parameter will
change
to the default of machine precision times ``max(M, N)`` where M and N are
the
input matrix dimensions.
To use the future default and silence this warning we advise to pass
`rcond=None`, to keep using the old, explicitly pass `rcond=-1`.
  """
```

[17]:
```
array([  62.07662454,   99.98500403, 1442.83746254])
```

[18]:
```
rms = lambda x: np.sqrt(np.mean(np.square(x))) #not a built in numpy
function
rms(np.matmul(R,s) - vdes)
```

[18]:
```
132.6381902632653
```