Table of contents

1. Zero-Sum Dynamic Games in Discrete Time
2. Discrete-Time Dynamic Programming
3. Solving Finite Zero-Sum Games with MATLAB
4. Linear Quadratic Dynamic Games
5. Practice Exercise
Zero-Sum Dynamic Games in Discrete Time

Computation of saddle-point equilibria of zero-sum discrete-time dynamic games in state-feedback policies

Solution methods for two-player zero-sum dynamic games in discrete time, which correspond to dynamics of the form

\[ x_{k+1} = f_k \left( \begin{array}{ccc} x_k & , & u_k & , & d_k \end{array} \right) \forall k \in \{1, \ldots, K\} \]

starting at some initial state \( x_1 \) in the state space \( X \).

At each time \( k \)
- \( P_1 \)'s action \( u_k \) is required to belong to a given action space \( U_k \).
- \( P_2 \)'s action \( d_k \) is required to belong to a given action space \( D_k \).
Zero-Sum Dynamic Games in Discrete Time

Assume finite horizon \((K < \infty)\) stage-additive costs of the form

\[
J := \sum_{k=1}^{K} g_k(x_k, u_k)
\]

that \(P_1\) wants to minimize and \(P_2\) wants to maximize.

Consider a state-FB information structure, which corresponds to policies of the form

\[
u_k = \gamma_k(x_k), \quad d_k = \sigma_k(x_k), \quad \forall k \in \{1, 2, \ldots, K\}
\]

For a state-FB policy \(\gamma\) for \(P_1\) and a state-FB policy \(\sigma\) for \(P_2\), denote by \(J(\gamma, \sigma)\) the corresponding value of the cost \(J\).
Zero-Sum Dynamic Games in Discrete Time

Goal: saddle-point pair of equilibrium policies \((\gamma^*, \sigma^*)\) for which

\[
J(\gamma^*, \sigma) \leq J(\gamma^*, \sigma^*) \leq J(\gamma, \sigma^*), \quad \forall \gamma \in \Gamma_1, \; \sigma \in \Gamma_2
\]

where \(\Gamma_1\) and \(\Gamma_2\): sets of all state-FB policies for \(P_1\) and \(P_2\).

Rewriting the saddle-point equilibrium (SPE) pair as

\[
J(\gamma^*, \sigma^*) = \min_{\gamma \in \Gamma_1} (\gamma, \sigma^*), \quad J(\gamma^*, \sigma^*) = \max_{\sigma \in \Gamma_2} (\gamma^*, \sigma)
\]

we conclude that if \(\sigma^*\) was known we could obtain \(\gamma^*\) from the single-player optimization

minimize over \(\gamma \in \Gamma_1\) the cost \(J(\gamma, \sigma^*) := \sum_{k=1}^{K} g_k(x_k, u_k, \sigma_k^*(x_k))\)

subject to the dynamics \(x_{k+1} = f_k(x_k, u_k, \sigma_k^*(x_k))\)
From Module 15:

An optimal state-FB policy $\gamma^*$ could be constructed using a backward iteration to compute the cost-to-go $V_k^1(x)$ for $P_1$ using

$$V_{K+1}^1(x) = 0, \quad V_k^1(x) = \inf_{u_k \in U_k} \left( g_k(x, u_k, \sigma_k^*(x)) + V_{k+1}^1(f_k(x, u_k, \sigma_k^*(x))) \right)$$

$\forall k \in \{1, 2, \ldots, K\}$, and then

$$\gamma_k^* := \arg\min_{u_k \in U_k} \left( g_k(x, u_k, \sigma_k^*(x)) + V_{k+1}(f_k(x, u_k, \sigma_k^*(x))) \right), \quad \forall k \in \{1, 2, \ldots, K\}$$

Moreover, the minimum $J(\gamma^*, \sigma^*)$ is given by $V_1^1(x_1)$. 
Similarly, if $\gamma^*$ was known we could obtain an optimal state-FB policy $\sigma^*$ from the single-player optimization

$$\max_{\sigma \in \Gamma_2} J(\gamma^*, \sigma) := \sum_{k=1}^{K} g_k(x_k, \gamma_k^*(x_k), d_k)$$

subject to the dynamics $x_{k+1} = f_k(x_k, \gamma_k^*(x_k), d_k)$

An optimal state-FB policy $\sigma^*$ could be constructed using a backward iteration to compute the cost-to-go $V_k^2(x)$ for $P_2$ using

$$V_{K+1}^2(x) = 0, \quad V_k^2(x) = \sup_{d_k \in D_k} (g_k(x, \gamma_k^*(x), d_k) + V_{k+1}^2(f_k(x, \gamma_k^*(x), d_k)))$$

$\forall k \in \{1, 2, \ldots, K\}$, and then

$$\sigma_k^* := \arg \max_{d_k \in D_k} (g_k(x, \gamma_k^*(x), d_k) + V_{k+1}(f_k(x, \gamma_k^*(x), d_k)))$$

$\forall k \in \{1, 2, \ldots, K\}$

Moreover, the maximum $J(\gamma^*, \sigma^*)$ is given by $V_1^2(x_1)$. 

L.R. Garcia Carrillo

COSC-6590/GSCS-6390 Games: Theory and Applications Lecture 17 - State-Feedback Zero-Sum Dynamic Games
Discrete-Time Dynamic Programming
Discrete-Time Dynamic Programming

Key to finding the saddle-point pair of eq. policies $(\gamma^*, \sigma^*)$:

- it is possible to construct a pair of state-FB policies for which the equations $V_{K+1}^1, \gamma_k^*, V_{K+1}^2, \sigma_k^*$ all hold.

Consider costs-to-go $V_K^1, V_K^2$, and state-FB policies $\gamma_k^*, \sigma_k^*$ at the last stage. For $V_{K+1}^1(x), \gamma_k^*(x), V_{K+1}^2(x), \sigma_k^*(x)$ to hold we need

$$V_K^1(x) = \inf_{u_K \in \mathcal{U}_K} g_K(x, u_K, \sigma_K^*(x)),$$
$$\gamma_K^*(x) = \arg\min_{u_K \in \mathcal{U}_K} g_K(x, u_K, \sigma_K^*(x))$$

$$V_K^2(x) = \sup_{d_K \in \mathcal{D}_K} g_K(x, \gamma_K^*(x), d_K),$$
$$\sigma_K^*(x) = \arg\min_{d_K \in \mathcal{D}_K} g_K(x, \gamma_K^*(x), d_K)$$

which can be re-written equivalently as

$$V_K^1(x) = g_K(x, \gamma_K^*(x), \sigma_K^*) \leq g_K(x, u_K, \sigma_K^*(x)), \quad \forall u_K \in \mathcal{U}_K$$
$$V_K^2(x) = g_K(x, \gamma_K^*(x), \sigma_K^*) \geq g_K(x, \gamma_K^*(x), d_K), \quad \forall d_K \in \mathcal{D}_K$$
Conclusion: \( V^1_K(x) = V^2_K(x) \).

The pair \((\gamma^*_K(x), \sigma^*_K(x)) \in \mathcal{U}_K \times \mathcal{D}_K\) must be a SPE for the zero-sum game with outcome

\[ g_K(x, u_K, d_K) \]

and actions \( u_K \in \mathcal{U}_K \) for \( P_1 \) (minimizer) and \( d_K \in \mathcal{D}_K \) for \( P_2 \) (maximizer).

Moreover, \( V^1_K(x) = V^2_K(x) \) must be the value of this game.

Only possible: if security policies exist, and security levels for both players are equal to the value of the game, i.e.,

\[ V^1_K(x) = V^2_K(x) = V_K(x) := \min_{u_K \in \mathcal{U}_K} \sup_{d_K \in \mathcal{D}_K} g_K(x, u_K, d_K) \]

\[ = \max_{d_K \in \mathcal{D}_K} \inf_{u_K \in \mathcal{U}_K} g_K(x, u_K, d_K) \]
Consider now costs-to-go $V_{K-1}^1$, $V_{K-1}^2$ and state-FB policies $\gamma_{K-1}^*$, $\sigma_{K-1}^*$ at stage $K - 1$.

For $V_{K-1}^1(x)$, $\gamma_{K-1}^*(x)$, $V_{K-1}^2(x)$, $\sigma_{K-1}^*(x)$ to hold we need

$$V_{K-1}^1(x) = \inf_{u_{K-1} \in U_{K-1}} \left( g_{K-1}(x, u_{K-1}, \sigma_{K-1}^*(x)) + V_K(f_{K-1}(x, u_{K-1}, \sigma_{K-1}^*(x))) \right)$$

$$\gamma_{K-1}^*(x) := \arg\min_{u_{K-1} \in U_{K-1}} \left( g_{K-1}(x, u_{K-1}, \sigma_{K-1}^*(x)) + V_K(f_{K-1}(x, u_{K-1}, \sigma_{K-1}^*(x))) \right)$$

$$V_{K-1}^2(x) = \sup_{d_{K-1} \in D_{K-1}} \left( g_{K-1}(x, \gamma_{K-1}^*(x), d_{K-1}) + V_K(f_{K-1}(x, \gamma_{K-1}^*(x), d_{K-1})) \right)$$

$$\sigma_{K-1}^*(x) := \arg\min_{d_{K-1} \in D_{K-1}} \left( g_{K-1}(x, \gamma_{K-1}^*(x), d_{K-1}) + V_K(f_{K-1}(x, \gamma_{K-1}^*(x), d_{K-1})) \right)$$

We omit the superscripts in $V_{K}^1$ and $V_{K}^2$ in the RHS, since we have already seen that $V_{K}^1(x) = V_{K}^2(x)$. 

L.R. Garcia Carrillo

COSC-6590/GSCS-6390 Games: Theory and Applications Lecture 17 - State-Feedback Zero-Sum Dynamic Games
Discrete-Time Dynamic Programming

**Conclusion:** \((\gamma^*_{K-1}(x), \sigma^*_{K-1}(x)) \in \mathcal{U}_{K-1} \times \mathcal{D}_{K-1}\) must be a SPE for the zero-sum game with outcome

\[
g_{K-1}(x, u_{K-1}, d_{K-1}) + V_K(f_{K-1}(x, u_{K-1}, d_{K-1}))
\]

and actions \(u_{K-1} \in \mathcal{U}_{K-1}\) for \(P_1\) (minimizer) and \(d_{K-1} \in \mathcal{D}_{K-1}\) for \(P_2\) (maximizer).

Moreover, \(V^1_{K-1}(x) = V^2_{K-1}(x)\) must be precisely equal to the value of this game.

Continuing this reasoning backwards in time all the way to the first stage, we obtain the following result.
Theorem 17.1. Assume we can recursively compute functions $V_1(x), V_2(x), \ldots, V_{K+1}(x)$, such that $\forall x \in \mathcal{X}, k \in \{1, 2, \ldots, K\}$

$$V_k(x) := \min_{u_k \in U_k} \sup_{d_k \in D_k} \left( g_k(x, u_k, d_k) + V_{K+1}(f_k(x, u_k, d_k)) \right)$$

$$= \max_{d_k \in D_k} \inf_{u_k \in U_k} \left( g_k(x, u_k, d_k) + V_{K+1}(f_k(x, u_k, d_k)) \right)$$

where $V_{K+1}(x) = 0, \forall x \in \mathcal{X}$. Then the pair $(\gamma^*, \sigma^*)$ below is a SPE in state-FB policies:

$$\gamma^*(x) := \arg \min_{u_k \in U_k} \sup_{d_k \in D_k} \left( g_k(x, u_k, d_k) + V_{K+1}(f_k(x, u_k, d_k)) \right)$$

$$\sigma^*(x) := \arg \max_{d_k \in D_k} \inf_{u_k \in U_k} \left( g_k(x, u_k, d_k) + V_{K+1}(f_k(x, u_k, d_k)) \right)$$

$\forall x \in \mathcal{X}, k \in \{1, 2, \ldots, K\}$. And the value of the game is $V_1(x_1)$. 

---

L.R. Garcia Carrillo
COSC-6590/GSCS-6390  Games: Theory and Applications  Lecture 17 - State-Feedback Zero-Sum Dynamic Games
**Attention!** Theorem 17.1 provides a sufficient condition for the existence of NE, but this condition is not necessary.

The two security levels in $V_k(x)$ may not commute for a state $x$ at some stage $k$, but there still may be a SPE for the game.

- we saw this for games in extensive form.

When the min and max do not commute in $V_k(x)$, and $\mathcal{U}_k$ and $\mathcal{D}_k$ are finite, one may want to use a mixed SPE, leading to behavioral policies

- i.e., per-stage randomization.
Proof of Theorem 17.1.

Since the inf and sup commute in $V_k(x)$ and the definitions of $\gamma_k^*$ and $\sigma_k^*$, we conclude that the pair $(\gamma_k^*(x), \sigma_k^*(x))$ is a SPE for a zero-sum game with criterion

\[
\left( g_k(x, u_k, d_k) + V_{K+1}(f_k(x, u_k, d_k)) \right)
\]

which means that

\[
g_k(x, \gamma_k^*(x), d_k) + V_{K+1}(f_k(x, \gamma_k^*(x), d_k)) \\
\leq g_k(x, \gamma_k^*(x), \sigma_k^*(x)) + V_{K+1}(f_k(x, \gamma_k^*(x), \sigma_k^*(x))) \\
\leq g_k(x, u_k, \sigma_k^*(x)) + V_{K+1}(f_k(x, u_k, \sigma_k^*(x)))
\]

\[
\forall u_K \in U_K \text{ and } d_K \in D_K.
\]
Discrete-Time Dynamic Programming

Since the middle term in these inequalities is also equal to the RHS of $V_k(x)$, we have that

$$V_k(x) = g_k(x, \gamma_k^*(x), \sigma_k^*(x)) + V_{K+1}(f_k(x, \gamma_k^*(x), \sigma_k^*(x)))$$

$$= \sup_{d \in D} \left( g_k(x, \gamma_k^*(x), d) + V_{K+1}(f_k(x, \gamma_k^*(x), d)) \right), \quad \forall x \in \mathbb{R}^n, t \in [0, T]$$

which, from Theorem 15.1 shows that $\sigma_k^*(x)$ is an optimal (maximizing) state-FB policy against $\gamma_k^*(x)$ and the maximum is equal to $V_1(x_1)$. Moreover, since we also have that

$$V_k(x) = g_k(x, \gamma_k^*(x), \sigma_k^*(x)) + V_{K+1}(f_k(x, \gamma_k^*(x), \sigma_k^*(x)))$$

$$= \inf_{u \in U} \left( g_k(x, u, \sigma_k^*(x)) + V_{K+1}(f_k(x, u, \sigma_k^*(x))) \right), \quad \forall x \in \mathbb{R}^n, t \in [0, T]$$

then $\gamma_k^*(x)$ is an optimal (minimizing) state-FB policy against $\sigma_k^*(x)$ and the minimum is equal to $V_1(x_1)$. This proves that $(\gamma^*, \sigma^*)$ is a SPE in state-FB policies with value $V_1(x_1)$. 
Moreover, since we have that

\[ V_k(x) = g_k(x, \gamma_k^*(x), \sigma_k^*(x)) + V_{K+1}(f_k(x, \gamma_k^*(x), \sigma_k^*(x))) \]

\[ = \sup_{d \in D} \left( g_k(x, \gamma_k^*(x), d) + V_{K+1}(f_k(x, \gamma_k^*(x), d)) \right), \quad \forall x \in \mathbb{R}^n, t \in [0, T] \]

which, from Theorem 15.1 shows that \( \sigma_k^*(x) \) is an optimal (maximizing) state-FB policy against \( \gamma_k^*(x) \) and the maximum is equal to \( V_1(x_1) \).

We can actually conclude that \( P_2 \) cannot get a reward larger than \( V_1(x_1) \) against \( \gamma_k^*(x) \), regardless of the information structure available to \( P_2 \).
Moreover, since we have that

\[ V_k(x) = g_k(x, \gamma_k^*(x), \sigma_k^*(x)) + V_{K+1}(f_k(x, \gamma_k^*(x), \sigma_k^*(x))) \]

\[ = \inf_{u \in U} \left( g_k(x, u, \sigma_k^*(x)) + V_{K+1}(f_k(x, u, \sigma_k^*(x))) \right), \quad \forall x \in \mathbb{R}^n, t \in [0, T] \]

which, from Theorem 15.1 shows that \( \gamma_k^*(x) \) is an optimal (minimizing) state-FB policy against \( \sigma_k^*(x) \) and the minimum is equal to \( V_1(x_1) \).

We can actually conclude that \( P_1 \) cannot get a reward larger than \( V_1(x_1) \) against \( \sigma_k^*(x) \), regardless of the information structure available to \( P_1 \).
Note 16. We can actually conclude that

- $P_2$ cannot get a reward larger than $V_1(x_1)$ against $\gamma_k^*(x)$, regardless of the information structure available to $P_2$.
- $P_1$ cannot get a reward larger than $V_1(x_1)$ against $\sigma_k^*(x)$, regardless of the information structure available to $P_1$.

In practice, this means that $\gamma_k^*(x)$ and $\sigma_k^*(x)$ are extremely safe policies for $P_1$ and $P_2$, respectively, since they guarantee a level of reward regardless of the information structure for the other player.
Solving Finite Zero-Sum Games with MATLAB
Solving Finite Zero-Sum Games with MATLAB

The backwards iteration in $V_k(x)$ can be implemented very efficiently in MATLAB\textsuperscript{®}

Enumerate all states so that the state-space can be viewed as

$$\mathcal{X} := \{1, 2, \ldots, n_X\}$$

Enumerate all actions so that the action spaces can be viewed as

$$\mathcal{U} := \{1, 2, \ldots, n_U\} \quad \mathcal{D} := \{1, 2, \ldots, n_D\}$$

Assume that all states can occur at every stage and that all actions are also available at every stage.

Functions $f_k(x, u, d)$ (the game dynamics) and $g_k(x, u, d)$ (the stage-cost) can be represented by a three-dimensional $n_X \times n_U \times n_D$ tensor. Each $V_k(x)$ can be represented by an $n_X \times 1$ columns vector with one row per state.
Solving Finite Zero-Sum Games with MATLAB

Suppose following variables are available within MATLAB®

\[ F : \text{cell-array with } K \text{ elements, each equal to an } n_X \times n_U \times n_D \]
three-dimensional matrix so that \( F\{k\} \) represents the game
dynamics function \( f_k(x, u, d), \forall x \in X, u \in U, d \in D, k \in \{1, 2, \ldots, K\} \).

- entry \( F\{k\}(i,j,l) \) of matrix \( F\{k\} \) is the state \( f_k(i, j, k) \).

\[ G : \text{cell-array with } K \text{ elements, each equal to an } n_X \times n_U \times n_D \]
three-dimensional matrix so that \( G\{k\} \) represents the stage-cost
function \( g_k(x, u, d), \forall x \in X, u \in U, d \in D, k \in \{1, 2, \ldots, K\} \).

- entry \( G\{k\}(i,j,l) \) of \( G\{k\} \) is the per-state cost \( g_k(i, j, k) \).
Solving Finite Zero-Sum Games with MATLAB

Construct $V_k(x)$ using the following MATLAB® code:

```matlab
V{K+1} = zeros(size(G{K},1),1,1);
for k = K:-1:1
    Vminmax = min(max(G{k} + V{k+1}(F{k})),[],3),[],2);
    Vmaxmin = max(min(G{k} + V{k+1}(F{k})),[],2),[],3);
    if any(Vminmax ~= Vmaxmin)
        error('Saddle - point cannot be found')
    end
    V{k} = Vminmax;
end
```

When procedure fails because $V_{\text{minmax}}$ and $V_{\text{maxmin}}$ differ, use a mixed policy using a linear program.

- indices of the states for which this is needed can be found using $k = \text{find}(V_{\text{minmax}} = V_{\text{maxmin}})$
After running the code, the following variable is created:

\( V \): cell-array with \( K + 1 \) elements, each equal to an \( n_x \times 1 \) columns vector so that \( V\{k\} \) represents \( V_k(x) \), \( \forall x \in \mathcal{X} \), \( k \in \{1, 2, \ldots, K\} \).

- entry \( V\{k\}(i) \) of the vector \( V\{k\} \) is the cost-to-go \( V_k(i) \) from state \( i \) at stage \( k \).

For a given state \( x \) at stage \( k \), the optimal actions \( u \) and \( d \) given by \( \gamma^x_k(x) \) and \( \sigma^x_k(x) \) can be obtained using

\[
[\sim, u] = \min(\max(G(x,:,:) + V\{k+1\}(F(x,:,:)), [], 3), [], 2);
\]
\[
[\sim, d] = \max(\min(G(x,:,:) + V\{k+1\}(F(x,:,:)), [], 2), [], 3);
\]
Linear Quadratic Dynamic Games
Linear Quadratic Dynamic Games

Characterized by linear dynamics of the form

\[ x_{k+1} = Ax_k + Bu_k + Ed_k, \quad x \in \mathbb{R}^n, u \in \mathbb{R}^{nu}, d \in \mathbb{R}^{nd}, k \in \{1, 2, \ldots, K\} \]

and a stage-additive quadratic cost of the form

\[ J := \sum_{k=1}^{K} (\|y_k\|^2 + \|u_k\|^2 - \mu^2\|d_k\|^2) = \sum_{k=1}^{K} (x_k' C' C x_k + u_k' u_k - \mu^2 d_k' d_k) \]

where

\[ y_k = C x_k, \quad \forall k \in \{1, 2, \ldots, K\} \]

\[ \mu : \text{a constant conversion factor that maps units of } d_k \text{ into units of } u_k \text{ and } y_k. \]
Linear Quadratic Dynamic Games

This cost function $J$ captures scenarios in which:

1. $P_1$ (minimizer) wants to make the $y_k$ small, without spending much effort in their actions $u_k$, $k \in \{1, 2, \ldots, K\}$

2. $P_2$ (maximizer) wants to make the same $y_k$ large, without spending much effort in their actions $d_k$, $k \in \{1, 2, \ldots, K\}$

Note. A conversion factor $\mu$ between units of $u$ and $y$ could be incorporated into the matrix $C$ that defines $y$. 
Linear Quadratic Dynamic Games

The equation $V_k(x)$ for this game is

$$V_k(x) := \min_{u_k \in U_k} \sup_{d_k \in D_k} \left( x'C'Cx + u'_k u_k - \mu^2 d'_k d_k + V_{k+1}(Ax + Bu_k + Ed_k) \right)$$

$$= \max_{d_k \in D_k} \inf_{u_k \in U_k} \left( x'C'Cx + u'_k u_k - \mu^2 d'_k d_k + V_{k+1}(Ax + Bu_k + Ed_k) \right)$$

\(\forall x \in \mathbb{R}^n, k \in \{1, 2, \ldots, K\}.\)

Inspired by the quadratic form of the stage cost, we will try to find a solution to $V_k(x)$ of the form

$$V_k(x) = x'P_k x, \quad \forall x \in \mathbb{R}^n, \quad k \in \{1, 2, \ldots, K + 1\}$$

for appropriately selected symmetric $n \times n$ matrices $P_k$. 
Linear Quadratic Dynamic Games

For $V_{K+1}(x) = 0$, $\forall x \in \mathcal{X}$ to hold, we need $P_{K+1} = 0$.

On the other hand, for $V_k(x)$ to hold we need

$$x' P_k x = \min_{u_k \in \mathbb{R}^{n_u}} \sup_{d_k \in \mathbb{R}^{n_d}} Q_x(u_k, d_k) = \max_{d_k \in \mathbb{R}^{n_d}} \inf_{u_k \in \mathbb{R}^{n_u}} Q_x(u_k, d_k)$$

$\forall x \in \mathbb{R}^n, k \in \{1, 2, \ldots, K\}$.

where

$$Q_x(u_k, d_k) := x' C' C x + u_k' u_k - \mu^2 d_k' d_k + (Ax + Bu_k + Ed_k)' P_{k+1} (Ax + Bu_k + Ed_k)$$

$$= \begin{bmatrix} u_k' & d_k' & x' \end{bmatrix} \begin{bmatrix} I + B' P_{k+1} B & B' P_{k+1} E & B' P_{k+1} A \\ E' P_{k+1} B & -\mu^2 I + E' P_{k+1} E & E' P_{k+1} A \\ A' P_{k+1} B & A' P_{k+1} E & C' C + A' P_{k+1} A \end{bmatrix} \begin{bmatrix} u_k \\ d_k \\ x \end{bmatrix}$$
Linear Quadratic Dynamic Games

The RHS of $x'P_kx$ can be viewed as a quadratic zero-sum game that has a saddle-point equilibrium

$$\begin{bmatrix} u^* \\ d^* \end{bmatrix} = -\begin{bmatrix} I + B'P_{k+1}B & B'P_{k+1}E \\ E'P_{k+1}B & -\mu^2 I + E'P_{k+1}E \end{bmatrix}^{-1} \begin{bmatrix} B'P_{k+1}A \\ E'P_{k+1}A \end{bmatrix} x$$

with value given by

$$x'\left(C'C + A'P_{k+1}A \right)$$

$$- [A'P_{k+1}B & A'P_{k+1}E] \begin{bmatrix} I + B'P_{k+1}B & B'P_{k+1}E \\ E'P_{k+1}B & -\mu^2 I + E'P_{k+1}E \end{bmatrix}^{-1} \begin{bmatrix} B'P_{k+1}A \\ E'P_{k+1}A \end{bmatrix} x$$

provided that

$$I + B'P_{k+1}B > 0 \quad -\mu^2 I + E'P_{k+1}E < 0$$
In this case, the conditions in $x'P_kx$ hold provided that

$$P_k = C'C + A'P_{k+1}A$$

$$- [A'P_{k+1}B \quad A'P_{k+1}E] \begin{bmatrix} I + B'P_{k+1}B & B'P_{k+1}E \\ E'P_{k+1}B & -\mu^2I + E'P_{k+1}E \end{bmatrix}^{-1} \begin{bmatrix} B'P_{k+1}A \\ E'P_{k+1}A \end{bmatrix}$$

Theorem 17.1 can be used to compute the SPE for this game and leads to the following result.

**Corollary 17.1.** Suppose we define the matrices $P_k$ according to the (backwards) recursion:

$$P_{K+1} = 0$$

$$P_k = C'C + A'P_{k+1}A$$

$$- [A'P_{k+1}B \quad A'P_{k+1}E] \begin{bmatrix} I + B'P_{k+1}B & B'P_{k+1}E \\ E'P_{k+1}B & -\mu^2I + E'P_{k+1}E \end{bmatrix}^{-1} \begin{bmatrix} B'P_{k+1}A \\ E'P_{k+1}A \end{bmatrix}$$

$$\forall k \in \{1, 2, \ldots, K\}.$$
Linear Quadratic Dynamic Games

Suppose also that

\[ I + B'P_{k+1}B > 0, \quad -\mu^2 I + E'P_{k+1}E < 0, \quad \forall k \in \{1, 2, \ldots, K\} \]

Then the pair of policies \((\gamma^*, \sigma^*)\) defined below is a SPE in state-FB policies:

\[
\begin{bmatrix}
\gamma_k^*(x) \\
\sigma_k^*(x)
\end{bmatrix} = - \begin{bmatrix}
I + B'P_{k+1}B & B'P_{k+1}E \\
E'P_{k+1}B & -\mu^2 I + E'P_{k+1}E
\end{bmatrix}^{-1} \begin{bmatrix}
B'P_{k+1}A \\
E'P_{k+1}A
\end{bmatrix} x
\]

\forall x \in \mathcal{X}, k \in \{1, 2, \ldots, K\}.

Moreover, the value of the game is equal to \(x_1P_1x_1\).
Note (Induced norm).

Since \((\gamma^*, \sigma^*)\) is a SPE with value \(x_1 P_1 x_1\), when \(P_1\) uses their security policy

\[ u_k = \gamma_k^*(x_k) \]

for every policy \(d_k = \sigma_k^*(x_k)\) for \(P_2\), we have that

\[ J(\gamma^*, \sigma^*) = x_1 P_1 x_1 \geq J(\gamma^*, \sigma) = \sum_{k=1}^{K} \left( ||y_k||^2 + ||u_k||^2 - \mu^2 ||d_k||^2 \right) \]

and therefore

\[ \sum_{k=1}^{K} ||y_k||^2 \leq x_1 P_1 x_1 + \mu^2 \sum_{k=1}^{K} ||d_k||^2 - \sum_{k=1}^{K} ||u_k||^2 \]
Linear Quadratic Dynamic Games

When \( x_1 = 0 \), this implies that

\[
\sum_{k=1}^{K} \|y_k\|^2 \leq \mu^2 \sum_{k=1}^{K} \|d_k\|^2
\]

In view of Note 16, this holds for every possible \( d_k \), regardless of the information structure available to \( P_2 \), and therefore we conclude that

\[
\sup_{d_k, k \in \{1, 2, \ldots, K\}} \sqrt{\sum_{k=1}^{K} \|y_k\|^2} \leq \mu
\]

In view of this, the control law \( u_k = \gamma_k^*(x_k) \) is said to achieve an \( \mathcal{L}_2 \)-induced norm from the disturbance \( d_k, k \in \{1, 2, \ldots, K\} \) to the output \( y_k, k \in \{1, 2, \ldots, K\} \) lower than or equal to \( \mu \).
Linear Quadratic Dynamic Games

Notation.

When $K = \infty$, the left-hand side of

$$
\sup_{d_k, k \in \{1, 2, \ldots, K\}} \frac{\sqrt{\sum_{k=1}^{K} \|y_k\|^2}}{\sqrt{\sum_{k=1}^{K} \|d_k\|^2}} \leq \mu
$$

is called the discrete-time H-infinity norm of the closed-loop and

$$u_k = \gamma_k^*(x_k)$$

guarantees an H-infinity norm smaller than or equal to $\mu$. 
Practice Exercise
Practice Exercise

17.1 (Tic-Tac-Toe). Write a MATLAB® script to compute the cost-to-go for each state of the Tic-Tac-Toe game.

Assumptions:
- $P_1$ (minimizer) places the Xs
- $P_2$ (maximizer) places the Os.

Game outcome:
- -1 when $P_1$ wins
- +1 when $P_2$ wins
- 0 when the game ends in a draw.

Hint: Draw inspiration from the code in Section 17.3, but keep in mind that Tic-Tac-Toe is a game of alternate play.
- algorithm in Section 17.3 is for simultaneous play.
Practice Exercise

The choices made for the design of the MATLAB® code:

**Alternate play:** To convert an alternate-play game like Tic-Tac-Toe into a simultaneous-play game

- expand each stage of the alternate-play game into 2 sequential stages of a simultaneous-play game.

For the Tic-Tac-Toe game, in stage

- 1: \( P_1 \) selects where to place the X. \( P_2 \) cannot place any O.
- 2: \( P_2 \) selects where to place an O. \( P_1 \) cannot place any X.

This continues, with

- \( P_1 \) placing Xs in stages 1, 3, 5, 7, and 9
- \( P_2 \) placing Os in stages 2, 4, 6, and 8

In this expanded 9-stage game, at each stage both players play simultaneously. But, one of the players has no choice to make.
Practice Exercise

**State encoding:** encode states of the game by assigning to each state an 18-bit integer. Each pair of bits in this integer is associated with one of the 9 slots in the Tic-Tac-Toe board as:

<table>
<thead>
<tr>
<th>Bit #</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slot</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

where the 9 slots are numbered as follows:

```
  1  2  3
  
  4  5  6
  7  8  9
```

The two bits associated with a slot indicate its content:

<table>
<thead>
<tr>
<th>most significant bit</th>
<th>least significant bit</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>empty slot</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>O</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>invalid</td>
</tr>
</tbody>
</table>
Practice Exercise

MATLAB® function ttt_addX(Sk)
Takes an $N \times 1$ vector Sk of integers representing states.
Generates an $N \times 9$ matrix newS that, for each of the $N$ states in Sk, computes all the states that would be obtained by adding an X to each of the 9 possible slots.

Function ttt_addX(Sk) generates two additional outputs:

invalid : $N \times 9$ boolean-valued matrix.
An entry equal to true indicates that the corresponding entry in newS does not correspond to a valid placement of an X because the corresponding slot was not empty

won : $N \times 9$ boolean-valued matrix.
An entry equal to true indicates that the corresponding entry in newS has three Xs in a row.
Practice Exercise

MATLAB® function ttt_addX(Sk)

function [newS, won, invalid] = ttt_addX(Sk)

XplayMasks = int32([bin2dec('010000 000000 000000');
                    bin2dec('000100 000000 000000');
                    bin2dec('000001 000000 000000');
                    bin2dec('000000 010000 000000');
                    bin2dec('000000 000100 000000');
                    bin2dec('000000 000001 000000');
                    bin2dec('000000 000000 010000');
                    bin2dec('000000 000000 000100');
                    bin2dec('000000 000000 000001')]);

% compute new state and test whether move is valid

newS = zeros(size(Sk,1),length(XplayMasks),'int32');
invalid = false(size(newS));
for slot = 1:length(XplayMasks)
    mask = XplayMasks(slot);
    newS(:,slot) = bitor(S,mask);
    invalid(bitand(Sk,mask + 2*mask)~=0,slot ) = true;
end
Practice Exercise

```matlab
XwinMasks = int32([bin2dec('010101 000000 000000'); % top horizontal
                    bin2dec('000000 010101 000000'); % mid horizontal
                    bin2dec('000000 000000 010101'); % bottom horizontal
                    bin2dec('010000 010000 010000'); % left vertical
                    bin2dec('000100 000100 000100'); % center vertical
                    bin2dec('000001 000001 000001'); % right vertical
                    bin2dec('010000 000100 000001'); % descend diagonal
                    bin2dec('000001 000100 010000')]); % ascend diagonal

% check if X won
won = false(size(newS));
for i = 1:length(XwinMasks)
    won = bitor(won,bitand(newS,XwinMasks(i)) == XwinMasks(i));
end
end
```
Function `ttt_add0(Sk)` : similar role, but now for adding Os.

```matlab
function [newS,won,invalid] = ttt_add0(Sk,slot)
    OplayMasks = int32([bin2dec('100000 000000 000000');
                       bin2dec('001000 000000 000000');
                       bin2dec('000010 000000 000000');
                       bin2dec('000000 100000 000000');
                       bin2dec('000000 001000 000000');
                       bin2dec('000000 000010 000000');
                       bin2dec('000000 000000 100000');
                       bin2dec('000000 000000 001000');
                       bin2dec('000000 000000 000010')]);

    % compute new state and test whether move is valid
    newS = zeros(size(Sk,1),length(OplayMasks),'int32');
    invalid = false(size(newS));
    for slot = 1:length(OplayMasks)
        mask = OplayMasks(slot);
        newS(:,slot) = bitor(Sk,mask);
        invalid(bitand(Sk,mask + mask/2)~=0,slot) = true;
    end
```

L.R. Garcia Carrillo
COSC-6590/GSCS-6390  Games: Theory and Applications  Lecture 17 - State-Feedback Zero-Sum Dynamic Games
Practice Exercise

OwinMasks = int32([bin2dec('101010 000000 000000'); % top horizontal
                    bin2dec('000000 101010 000000'); % mid horizontal
                    bin2dec('000000 000000 101010'); % bottom horizontal
                    bin2dec('100000 100000 100000'); % left vertical
                    bin2dec('001000 001000 001000'); % center vertical
                    bin2dec('000010 000010 000010'); % right vertical
                    bin2dec('100000 001000 000010'); % descend diagonal
                    bin2dec('000010 001000 100000')]); % ascend diagonal

% check if 0 won

won = false(size(newS));
for i = 1:length(OwinMasks)
    won = bitor(won,bitand(newS,OwinMasks(i)) == OwinMasks(i));
end
end
Practice Exercise

**State enumeration:** To compute cost-to-go, enumerate all states that can occur at each stage of the Tic-Tac-Toe game.

```matlab
function S = ttt_states(S0)
    K = 9;
    S = cell(K+1,1);
    S{1} = S0;
    for k = 1:K
        if rem(k,2) == 1 % player X (minimizer) plays at odd stages
            [newS,won,invalid] = ttt_addX(S{k}); % compute all next states
        else % player O (minimizer) plays at even stages
            [newS,won,invalid] = ttt_addO(S{k}); % compute all next states
        end
        % stack all states in a column vector
        newS = reshape(newS,[],1);
        won = reshape(won,[],1);
        invalid = reshape(invalid,[],1);
        % store (unique) list of states for which the game continues
        S{k+1} = unique(newS(~invalid & ~won ));
    end
    end
```

Returns cell-array `S` with 10 elements. Each entry `S{k}` is a vector containing all valid stage-`k` states for which game has not yet finished. Removes **game-over** states from `S{k}`: no cost-to-go for these.
### Practice Exercise

**Final code:** The following code computes the cost-to-go for each state in the cell-array S computed by the function `ttt_states()`.

```matlab
K = 9;
V = cell(K+1,1);
V{K+1} = zeros(size(S{K+1}),’int8’);
for k = K:-1:1
    if rem(k,2) == 1
        % player X (minimizer) plays at odd stages
        [newS,won,invalid] = ttt_addX(S{k}); % compute all next states
        % convert states to indices in S{k+1}
        % to get their costs-to-go from V{k+1} states
        [exists,newSndx] = ismember(newS,S{k+1});
        % compute all possible values
        newV = zeros(size(newS),’int8’);
        newV(exists) = V{k+1}(newSndx(exists));
        newV(won) = -1;
        newV(invalid) = + Inf; % penalize invalid actions for minimizer
        V{k} = min(newV,[],2); % pick best for minimizer
    end
end
```
L.R. Garcia Carrillo
COSC-6590/GSCS-6390  Games: Theory and Applications  Lecture 17 - State-Feedback Zero-Sum Dynamic Games
Practice Exercise

```
else
    % player 0 (maximizer) plays at even stages
    [newS,won,invalid] = ttt_add0(S{k}); % compute all next states
    % convert states to indices in S{k+1}
    % to get their costs-to-go from V{k+1}
    [exists,newS] = ismember(newS,S{k+1});
    % compute all possible values
    newV = zeros(size(newS),'int8');
    newV(exists) = V{k+1}(newS(exists));
    newV(won) = 1;
    newV(invalid) = -Inf; % penalize invalid actions for maximizer
    V{k} = max(newV,[],2); % pick best for maximizer
end
```

This code returns a cell-array $V$ with 10 elements.

Each entry $V{k}$ of $V$ is an array with the same size as $S{k}$ whose entries are equal to the cost-to-go from the corresponding state in $S{k}$ at stage $k$. 
Practice Exercise

Code has same structure as the code in Section 17.3, but it is optimized to take advantage of the structure of this game:

1.- Since $P_1$ places an X at the odd stages and $P_2$ places an O the even stages, we find an if statement inside the for loop that allows the construction of the cost-to-go $V\{k\}$ to differ depending on whether $k$ is even or odd.

2.- For the code in Section 17.3, the matrix $F\{k\}$ contains all possible states that can be reached at stage $k+1$ for all possible actions for each player.

Functions `ttt_addX(S\{k\})` and `ttt_add0(S\{k\})` provide this set of states at the even and odd stages, respectively.

The variable `newS` corresponds to $F\{k\}$ in the code in Section 17.3, but `newS` contains invalid states that need to be ignored.
The code in Section 17.3 uses $G\{k\} + V\{k+1\} (F\{k\})$ to add the per-stage cost $G\{k\}$ at stage $k$ with the cost-to-go $V\{k+1\} (F\{k\})$ from stage $k+1$.

In the Tic-Tac-Toe game, the per-stage cost is always zero unless the game finishes, so there is no need to add the per-stage cost until one of the players wins.

When a player wins, we do not need to consider the cost-to-go from subsequent stages because the game will end.

The variable `newV` corresponds to $G\{k\} + V\{k+1\} (F\{k\})$ in the code in Section 17.3.
Practice Exercise

4.- When \( k \) is odd only \( P_1 \) (minimizer) can make a choice: there is no maximization to carry out over actions of \( P_2 \). \( V_{\text{minmax}} \) and \( V_{\text{maxmin}} \) are obtained with a simple minimization and are always equal to each other.

When \( k \) is even only \( P_2 \) (maximizer) can make a choice: there is no minimization to carry out over actions of \( P_1 \). \( V_{\text{minmax}} \) and \( V_{\text{maxmin}} \) are obtained with a simple maximization and are always equal to each other.

This means that we do not need to compute \( V_{\text{minmax}} \) and \( V_{\text{maxmin}} \) and test if they are equal, before assigning their value to \( V\{k\} \).
End of Lecture

17 - State-Feedback Zero-Sum Dynamic Games

Questions?