Classical Control with Linear Algebra

Thomas J. Sargent and John Stachurski

December 20, 2020

1 Contents

2 Overview

In an earlier lecture Linear Quadratic Dynamic Programming Problems, we have studied how to solve a special class of dynamic optimization and prediction problems by applying the method of dynamic programming. In this class of problems

- the objective function is **quadratic** in **states** and **controls**.
- the one-step transition function is **linear**.
- shocks are IID Gaussian or martingale differences.

In this lecture and a companion lecture Classical Filtering with Linear Algebra, we study the classical theory of linear-quadratic (LQ) optimal control problems.

The classical approach does not use the two closely related methods – dynamic programming and Kalman filtering – that we describe in other lectures, namely, Linear Quadratic Dynamic Programming Problems and A First Look at the Kalman Filter.

Instead, they use either

- z-transform and lag operator methods, or
- matrix decompositions applied to linear systems of first-order conditions for optimum problems.

In this lecture and the sequel Classical Filtering with Linear Algebra, we mostly rely on elementary linear algebra.

The main tool from linear algebra we'll put to work here is LU decomposition.

We'll begin with discrete horizon problems.

Then we'll view infinite horizon problems as appropriate limits of these finite horizon problems.

Later, we will examine the close connection between LQ control and least-squares prediction and filtering problems.

These classes of problems are connected in the sense that to solve each, essentially the same mathematics is used.

Let's start with some standard imports:

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

2.1 References

Useful references include [6], [2], [4], [1], and [3].

3 A Control Problem

Let L be the **lag operator**, so that, for sequence $\{x_t\}$ we have $Lx_t = x_{t-1}$.

More generally, let $L^k x_t = x_{t-k}$ with $L^0 x_t = x_t$ and

$$d(L) = d_0 + d_1 L + ... + d_m L^m$$

where $d_0, d_1, ..., d_m$ is a given scalar sequence.

Consider the discrete-time control problem

$$\max_{\{y_t\}} \lim_{N\to\infty} \sum_{t=0}^{N} \beta^t \left\{ a_t y_t - \frac{1}{2} h y_t^2 - \frac{1}{2} \left[d(L) y_t \right]^2 \right\}, \tag{1}$$

where

- h is a positive parameter and $\beta \in (0,1)$ is a discount factor.
- $\{a_t\}_{t\geq 0}$ is a sequence of exponential order less than $\beta^{-1/2}$, by which we mean $\lim_{t\to\infty} \beta^{\frac{t}{2}} a_t = 0$.

Maximization in (1) is subject to initial conditions for $y_{-1}, y_{-2} \cdots, y_{-m}$.

Maximization is over infinite sequences $\{y_t\}_{t\geq 0}$.

3.1 Example

The formulation of the LQ problem given above is broad enough to encompass many useful models.

As a simple illustration, recall that in LQ Control: Foundations we consider a monopolist facing stochastic demand shocks and adjustment costs.

Let's consider a deterministic version of this problem, where the monopolist maximizes the discounted sum

$$\sum_{t=0}^{\infty} \beta^t \pi_t$$

and

$$\pi_t = p_t q_t - c q_t - \gamma (q_{t+1} - q_t)^2 \quad \text{with} \quad p_t = \alpha_0 - \alpha_1 q_t + d_t$$

In this expression, q_t is output, c is average cost of production, and d_t is a demand shock.

The term $\gamma (q_{t+1} - q_t)^2$ represents adjustment costs.

You will be able to confirm that the objective function can be rewritten as (1) when

- $a_t := \alpha_0 + d_t - c$
- $h := 2\alpha_1$
- $d(L) := \sqrt{2\gamma}(I - L)$

Further examples of this problem for factor demand, economic growth, and government policy problems are given in ch. IX of [5].

4 Finite Horizon Theory

We first study a finite N version of the problem.

Later we will study an infinite horizon problem solution as a limiting version of a finite horizon problem.

(This will require being careful because the limits as $N \to \infty$ of the necessary and sufficient conditions for maximizing finite N versions of (1) are not sufficient for maximizing (1))

We begin by

1. fixing $N > m$,
2. differentiating the finite version of (1) with respect to $y_0, y_1, \ldots, y_N$, and
3. setting these derivatives to zero.

For $t = 0, \ldots, N - m$ these first-order necessary conditions are the *Euler equations*.

For $t = N - m + 1, \ldots, N$, the first-order conditions are a set of *terminal conditions*.

Consider the term

$$\begin{aligned} J &= \sum_{t=0}^{N} \beta^t [d(L) y_t][d(L) y_t] \\ &= \sum_{t=0}^{N} \beta^t \left(d_0 y_t + d_1 y_{t-1} + \cdots + d_m y_{t-m}\right) \left(d_0 y_t + d_1 y_{t-1} + \cdots + d_m y_{t-m}\right) \end{aligned}$$

Differentiating J with respect to y_t for $t = 0,\ 1,\ \ldots,\ N - m$ gives

$$\begin{aligned}\frac{\partial J}{\partial y_t} &= 2\beta^t\, d_0\, d(L)y_t + 2\beta^{t+1}\, d_1\, d(L)y_{t+1} + \cdots + 2\beta^{t+m}\, d_m\, d(L)y_{t+m} \\ &= 2\beta^t \left(d_0 + d_1\, \beta L^{-1} + d_2\, \beta^2\, L^{-2} + \cdots + d_m\, \beta^m\, L^{-m}\right) d(L)y_t\end{aligned}$$

We can write this more succinctly as

$$\frac{\partial J}{\partial y_t} = 2\beta^t\, d(\beta L^{-1})\, d(L)y_t \tag{2}$$

Differentiating J with respect to y_t for $t = N - m + 1, \ldots, N$ gives

$$\begin{aligned}\frac{\partial J}{\partial y_N} &= 2\beta^N\, d_0\, d(L)y_N \\ \frac{\partial J}{\partial y_{N-1}} &= 2\beta^{N-1} \left[d_0 + \beta\, d_1\, L^{-1}\right] d(L)y_{N-1} \\ &\vdots \qquad \vdots \\ \frac{\partial J}{\partial y_{N-m+1}} &= 2\beta^{N-m+1} \left[d_0 + \beta L^{-1}\, d_1 + \cdots + \beta^{m-1}\, L^{-m+1}\, d_{m-1}\right] d(L)y_{N-m+1}\end{aligned} \tag{3}$$

With these preliminaries under our belts, we are ready to differentiate (1).

Differentiating (1) with respect to y_t for $t = 0, \ldots, N - m$ gives the Euler equations

$$\left[h + d\left(\beta L^{-1}\right) d(L)\right] y_t = a_t, \quad t = 0,\, 1,\, \ldots,\, N - m \tag{4}$$

The system of equations (4) forms a $2 \times m$ order linear *difference equation* that must hold for the values of t indicated.

Differentiating (1) with respect to y_t for $t = N - m + 1, \ldots, N$ gives the terminal conditions

$$\begin{aligned}\beta^N (a_N - h y_N - d_0\, d(L)y_N) &= 0 \\ \beta^{N-1} \left(a_{N-1} - h y_{N-1} - \left(d_0 + \beta\, d_1\, L^{-1}\right) d(L)\, y_{N-1}\right) &= 0 \\ &\vdots \\ \beta^{N-m+1} \left(a_{N-m+1} - h y_{N-m+1} - (d_0 + \beta L^{-1} d_1 + \cdots + \beta^{m-1} L^{-m+1} d_{m-1}) d(L) y_{N-m+1}\right) &= 0\end{aligned} \tag{5}$$

In the finite N problem, we want simultaneously to solve (4) subject to the m initial conditions $y_{-1}, \ldots, y_{-m}$ and the m terminal conditions (5).

These conditions uniquely pin down the solution of the finite N problem.

That is, for the finite N problem, conditions (4) and (5) are necessary and sufficient for a maximum, by concavity of the objective function.

Next, we describe how to obtain the solution using matrix methods.

4.1 Matrix Methods

Let's look at how linear algebra can be used to tackle and shed light on the finite horizon LQ control problem.

4.1.1 A Single Lag Term

Let's begin with the special case in which $m = 1$.

We want to solve the system of $N + 1$ linear equations

$$\begin{aligned} [h + d\,(\beta L^{-1})\,d\,(L)]y_t &= a_t, \quad t = 0,\ 1,\ \ldots,\ N-1 \\ \beta^N[a_N - h\,y_N - d_0\,d\,(L)y_N] &= 0 \end{aligned} \tag{6}$$

where $d(L) = d_0 + d_1 L$.

These equations are to be solved for $y_0, y_1, \ldots, y_N$ as functions of $a_0, a_1, \ldots, a_N$ and y_{-1}.

Let

$$\phi(L) = \phi_0 + \phi_1 L + \beta\phi_1 L^{-1} = h + d(\beta L^{-1})d(L) = (h + d_0^2 + d_1^2) + d_1 d_0 L + d_1 d_0 \beta L^{-1}$$

Then we can represent (6) as the matrix equation

$$\begin{bmatrix} (\phi_0 - d_1^2) & \phi_1 & 0 & 0 & \ldots & \ldots & 0 \\ \beta\phi_1 & \phi_0 & \phi_1 & 0 & \ldots & \ldots & 0 \\ 0 & \beta\phi_1 & \phi_0 & \phi_1 & \ldots & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & \ldots & \ldots & \ldots & \beta\phi_1 & \phi_0 & \phi_1 \\ 0 & \ldots & \ldots & \ldots & 0 & \beta\phi_1 & \phi_0 \end{bmatrix} \begin{bmatrix} y_N \\ y_{N-1} \\ y_{N-2} \\ \vdots \\ y_1 \\ y_0 \end{bmatrix} = \begin{bmatrix} a_N \\ a_{N-1} \\ a_{N-2} \\ \vdots \\ a_1 \\ a_0 - \phi_1 y_{-1} \end{bmatrix} \tag{7}$$

or

$$W\bar{y} = \bar{a} \tag{8}$$

Notice how we have chosen to arrange the y_t's in reverse time order.

The matrix W on the left side of (7) is "almost" a Toeplitz matrix (where each descending diagonal is constant).

There are two sources of deviation from the form of a Toeplitz matrix

1. The first element differs from the remaining diagonal elements, reflecting the terminal condition.
2. The sub-diagonal elements equal β time the super-diagonal elements.

The solution of (8) can be expressed in the form

$$\bar{y} = W^{-1}\bar{a} \tag{9}$$

which represents each element y_t of $\bar{y}$ as a function of the entire vector $\bar{a}$.

That is, y_t is a function of past, present, and future values of a's, as well as of the initial condition y_{-1}.

4.1.2 An Alternative Representation

An alternative way to express the solution to (7) or (8) is in so-called **feedback-feedforward** form.

The idea here is to find a solution expressing y_t as a function of *past* y's and *current* and *future* a's.

To achieve this solution, one can use an LU decomposition of W.

There always exists a decomposition of W of the form $W = LU$ where

- L is an $(N+1) \times (N+1)$ lower triangular matrix.
- U is an $(N+1) \times (N+1)$ upper triangular matrix.

The factorization can be normalized so that the diagonal elements of U are unity.

Using the LU representation in (9), we obtain

$$U\bar{y} = L^{-1}\bar{a} \tag{10}$$

Since L^{-1} is lower triangular, this representation expresses y_t as a function of

- lagged y's (via the term $U\bar{y}$), and
- current and future a's (via the term $L^{-1}\bar{a}$)

Because there are zeros everywhere in the matrix on the left of (7) except on the diagonal, super-diagonal, and sub-diagonal, the LU decomposition takes

- L to be zero except in the diagonal and the leading sub-diagonal.
- U to be zero except on the diagonal and the super-diagonal.

Thus, (10) has the form

$$\begin{bmatrix} 1 & U_{12} & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & U_{23} & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & U_{34} & \dots & 0 & 0 \\ 0 & 0 & 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 & U_{N,N+1} \\ 0 & 0 & 0 & 0 & \dots & 0 & 1 \end{bmatrix} \begin{bmatrix} y_N \\ y_{N-1} \\ y_{N-2} \\ y_{N-3} \\ \vdots \\ y_1 \\ y_0 \end{bmatrix} =$$

$$\begin{bmatrix} L_{11}^{-1} & 0 & 0 & \dots & 0 \\ L_{21}^{-1} & L_{22}^{-1} & 0 & \dots & 0 \\ L_{31}^{-1} & L_{32}^{-1} & L_{33}^{-1} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{N,1}^{-1} & L_{N,2}^{-1} & L_{N,3}^{-1} & \dots & 0 \\ L_{N+1,1}^{-1} & L_{N+1,2}^{-1} & L_{N+1,3}^{-1} & \dots & L_{N+1\,N+1}^{-1} \end{bmatrix} \begin{bmatrix} a_N \\ a_{N-1} \\ a_{N-2} \\ \vdots \\ a_1 \\ a_0 - \phi_1 y_{-1} \end{bmatrix}$$

where L_{ij}^{-1} is the (i,j) element of L^{-1} and U_{ij} is the (i,j) element of U.

Note how the left side for a given t involves y_t and one lagged value y_{t-1} while the right side involves all future values of the forcing process $a_t, a_{t+1}, \dots, a_N$.

4.1.3 Additional Lag Terms

We briefly indicate how this approach extends to the problem with $m > 1$.

Assume that $\beta = 1$ and let D_{m+1} be the $(m+1) \times (m+1)$ symmetric matrix whose elements are determined from the following formula:

$$D_{jk} = d_0 d_{k-j} + d_1 d_{k-j+1} + ... + d_{j-1} d_{k-1}, \qquad k \geq j$$

Let I_{m+1} be the $(m+1) \times (m+1)$ identity matrix.

Let ϕ_j be the coefficients in the expansion $\phi(L) = h + d(L^{-1})d(L)$.

Then the first order conditions (4) and (5) can be expressed as:

$$(D_{m+1} + hI_{m+1}) \begin{bmatrix} y_N \\ y_{N-1} \\ \vdots \\ y_{N-m} \end{bmatrix} = \begin{bmatrix} a_N \\ a_{N-1} \\ \vdots \\ a_{N-m} \end{bmatrix} + M \begin{bmatrix} y_{N-m+1} \\ y_{N-m-2} \\ \vdots \\ y_{N-2m} \end{bmatrix}$$

where M is $(m+1) \times m$ and

$$M_{ij} = \begin{cases} D_{i-j,\,m+1} \text{ for } i > j \\ 0 \text{ for } i \leq j \end{cases}$$

$$\begin{aligned} \phi_m y_{N-1} + \phi_{m-1} y_{N-2} + ... + \phi_0 y_{N-m-1} + \phi_1 y_{N-m-2} + \\ ... + \phi_m y_{N-2m-1} = a_{N-m-1} \\ \phi_m y_{N-2} + \phi_{m-1} y_{N-3} + ... + \phi_0 y_{N-m-2} + \phi_1 y_{N-m-3} + \\ ... + \phi_m y_{N-2m-2} = a_{N-m-2} \\ \vdots \\ \phi_m y_{m+1} + \phi_{m-1} y_m + + ... + \phi_0 y_1 + \phi_1 y_0 + \phi_m y_{-m+1} = a_1 \\ \phi_m y_m + \phi_{m-1} y_{m-1} + \phi_{m-2} + ... + \phi_0 y_0 + \phi_1 y_{-1} + ... + \phi_m y_{-m} = a_0 \end{aligned}$$

As before, we can express this equation as $W\bar{y} = \bar{a}$.

The matrix on the left of this equation is "almost" Toeplitz, the exception being the leading $m \times m$ submatrix in the upper left-hand corner.

We can represent the solution in feedback-feedforward form by obtaining a decomposition $LU = W$, and obtain

$$U\bar{y} = L^{-1}\bar{a} \tag{11}$$

$$\sum_{j=0}^{t} U_{-t+N+1,\,-t+N+j+1}\, y_{t-j} = \sum_{j=0}^{N-t} L_{-t+N+1,\,-t+N+1-j}\, \bar{a}_{t+j}\ , \qquad t = 0, 1, ..., N$$

where $L^{-1}_{t,s}$ is the element in the (t, s) position of L, and similarly for U.

The left side of equation (11) is the "feedback" part of the optimal control law for y_t, while the right-hand side is the "feedforward" part.

We note that there is a different control law for each t.

Thus, in the finite horizon case, the optimal control law is time-dependent.

It is natural to suspect that as $N \to \infty$, (11) becomes equivalent to the solution of our infinite horizon problem, which below we shall show can be expressed as

$$c(L)y_t = c(\beta L^{-1})^{-1} a_t \ ,$$

so that as $N \to \infty$ we expect that for each fixed $t, U^{-1}_{t,t-j} \to c_j$ and $L_{t,t+j}$ approaches the coefficient on L^{-j} in the expansion of $c(\beta L^{-1})^{-1}$.

This suspicion is true under general conditions that we shall study later.

For now, we note that by creating the matrix W for large N and factoring it into the LU form, good approximations to $c(L)$ and $c(\beta L^{-1})^{-1}$ can be obtained.

5 The Infinite Horizon Limit

For the infinite horizon problem, we propose to discover first-order necessary conditions by taking the limits of (4) and (5) as $N \to \infty$.

This approach is valid, and the limits of (4) and (5) as N approaches infinity are first-order necessary conditions for a maximum.

However, for the infinite horizon problem with $\beta < 1$, the limits of (4) and (5) are, in general, not sufficient for a maximum.

That is, the limits of (5) do not provide enough information uniquely to determine the solution of the Euler equation (4) that maximizes (1).

As we shall see below, a side condition on the path of y_t that together with (4) is sufficient for an optimum is

$$\sum_{t=0}^{\infty} \beta^t \, h y_t^2 < \infty \tag{12}$$

All paths that satisfy the Euler equations, except the one that we shall select below, violate this condition and, therefore, evidently lead to (much) lower values of (1) than does the optimal path selected by the solution procedure below.

Consider the *characteristic equation* associated with the Euler equation

$$h + d\left(\beta z^{-1}\right) d\left(z\right) = 0 \tag{13}$$

Notice that if $\tilde{z}$ is a root of equation (13), then so is $\beta \tilde{z}^{-1}$.

Thus, the roots of (13) come in "β-reciprocal" pairs.

Assume that the roots of (13) are distinct.

Let the roots be, in descending order according to their moduli, $z_1, z_2, \ldots, z_{2m}$.

From the reciprocal pairs property and the assumption of distinct roots, it follows that $|z_j| > \sqrt{\beta}$ for $j \leq m$ and $|z_j| < \sqrt{\beta}$ for $j > m$.

It also follows that $z_{2m-j} = \beta z_{j+1}^{-1}, j = 0, 1, \ldots, m-1$.

Therefore, the characteristic polynomial on the left side of (13) can be expressed as

$$\begin{aligned} h + d(\beta z^{-1})d(z) &= z^{-m} z_0 (z - z_1) \cdots (z - z_m)(z - z_{m+1}) \cdots (z - z_{2m}) \\ &= z^{-m} z_0 (z - z_1)(z - z_2) \cdots (z - z_m)(z - \beta z_m^{-1}) \cdots (z - \beta z_2^{-1})(z - \beta z_1^{-1}) \end{aligned} \tag{14}$$

where z_0 is a constant.

In (14), we substitute $(z - z_j) = -z_j(1 - \frac{1}{z_j} z)$ and $(z - \beta z_j^{-1}) = z(1 - \frac{\beta}{z_j} z^{-1})$ for $j = 1, \ldots, m$ to get

$$h + d(\beta z^{-1})d(z) = (-1)^m (z_0 z_1 \cdots z_m)(1 - \frac{1}{z_1} z) \cdots (1 - \frac{1}{z_m} z)(1 - \frac{1}{z_1} \beta z^{-1}) \cdots (1 - \frac{1}{z_m} \beta z^{-1})$$

Now define $c(z) = \sum_{j=0}^{m} c_j \, z^j$ as

$$c\,(z) = \left[(-1)^m z_0 \, z_1 \cdots z_m\right]^{1/2} (1 - \frac{z}{z_1}) \, (1 - \frac{z}{z_2}) \cdots (1 - \frac{z}{z_m}) \tag{15}$$

Notice that (14) can be written

$$h + d\ (\beta z^{-1})\ d\ (z) = c\,(\beta z^{-1})\, c\,(z) \tag{16}$$

It is useful to write (15) as

$$c(z) = c_0 (1 - \lambda_1 \, z) \ldots (1 - \lambda_m z) \tag{17}$$

where

$$c_0 = \left[(-1)^m \, z_0 \, z_1 \cdots z_m\right]^{1/2}; \quad \lambda_j = \frac{1}{z_j}, \ \ j = 1, \ldots, m$$

Since $|z_j| > \sqrt{\beta}$ for $j = 1, \ldots, m$ it follows that $|\lambda_j| < 1/\sqrt{\beta}$ for $j = 1, \ldots, m$.

Using (17), we can express the factorization (16) as

$$h + d(\beta z^{-1})d(z) = c_0^2 (1 - \lambda_1 z) \cdots (1 - \lambda_m z)(1 - \lambda_1 \beta z^{-1}) \cdots (1 - \lambda_m \beta z^{-1})$$

In sum, we have constructed a factorization (16) of the characteristic polynomial for the Euler equation in which the zeros of $c(z)$ exceed $\beta^{1/2}$ in modulus, and the zeros of $c\,(\beta z^{-1})$ are less than $\beta^{1/2}$ in modulus.

Using (16), we now write the Euler equation as

$$c(\beta L^{-1})\, c\,(L)\, y_t = a_t$$

The unique solution of the Euler equation that satisfies condition (12) is

$$c(L)\, y_t = c\,(\beta L^{-1})^{-1} a_t \tag{18}$$

This can be established by using an argument paralleling that in chapter IX of [5].

To exhibit the solution in a form paralleling that of [5], we use (17) to write (18) as

$$(1-\lambda_1 L)\cdots(1-\lambda_m L) y_t = \frac{c_0^{-2} a_t}{(1-\beta\lambda_1 L^{-1})\cdots(1-\beta\lambda_m L^{-1})} \tag{19}$$

Using partial fractions, we can write the characteristic polynomial on the right side of (19) as

$$\sum_{j=1}^{m} \frac{A_j}{1-\lambda_j\,\beta L^{-1}} \quad \text{where} \quad A_j := \frac{c_0^{-2}}{\prod_{i\neq j}(1-\frac{\lambda_i}{\lambda_j})}$$

Then (19) can be written

$$(1-\lambda_1 L)\cdots(1-\lambda_m L) y_t = \sum_{j=1}^{m} \frac{A_j}{1-\lambda_j\,\beta L^{-1}} a_t$$

or

$$(1-\lambda_1 L)\cdots(1-\lambda_m L) y_t = \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} (\lambda_j \beta)^k\, a_{t+k} \tag{20}$$

Equation (20) expresses the optimum sequence for y_t in terms of m lagged y's, and m weighted infinite geometric sums of future a_t's.

Furthermore, (20) is the unique solution of the Euler equation that satisfies the initial conditions and condition (12).

In effect, condition (12) compels us to solve the "unstable" roots of $h + d(\beta z^{-1})d(z)$ forward (see [5]).

The step of factoring the polynomial $h + d(\beta z^{-1})\, d(z)$ into $c\,(\beta z^{-1}) c\,(z)$, where the zeros of $c\,(z)$ all have modulus exceeding $\sqrt{\beta}$, is central to solving the problem.

We note two features of the solution (20)

- Since $|\lambda_j| < 1/\sqrt{\beta}$ for all j, it follows that $(\lambda_j\ \beta) < \sqrt{\beta}$.
- The assumption that $\{a_t\}$ is of exponential order less than $1/\sqrt{\beta}$ is sufficient to guarantee that the geometric sums of future a_t's on the right side of (20) converge.

We immediately see that those sums will converge under the weaker condition that $\{a_t\}$ is of exponential order less than ϕ^{-1} where $\phi = \max\{\beta\lambda_i, i = 1, \ldots, m\}$.

Note that with a_t identically zero, (20) implies that in general $|y_t|$ eventually grows exponentially at a rate given by $\max_i |\lambda_i|$.

The condition $\max_i |\lambda_i| < 1/\sqrt{\beta}$ guarantees that condition (12) is satisfied.

In fact, $\max_i |\lambda_i| < 1/\sqrt{\beta}$ is a necessary condition for (12) to hold.

Were (12) not satisfied, the objective function would diverge to $-\infty$, implying that the y_t path could not be optimal.

For example, with $a_t = 0$, for all t, it is easy to describe a naive (nonoptimal) policy for $\{y_t, t \geq 0\}$ that gives a finite value of (17).

We can simply let $y_t = 0$ for $t \geq 0$.

This policy involves at most m nonzero values of hy_t^2 and $[d(L)y_t]^2$, and so yields a finite value of (1).

Therefore it is easy to dominate a path that violates (12).

6 Undiscounted Problems

It is worthwhile focusing on a special case of the LQ problems above: the undiscounted problem that emerges when $\beta = 1$.

In this case, the Euler equation is

$$\left(h + d(L^{-1})d(L)\right) y_t = a_t$$

The factorization of the characteristic polynomial (16) becomes

$$\left(h + d\left(z^{-1}\right)d(z)\right) = c\left(z^{-1}\right) c\left(z\right)$$

where

$$\begin{aligned} c\left(z\right) &= c_0(1 - \lambda_1 z) \dots (1 - \lambda_m z) \\ c_0 &= \left[(-1)^m z_0 z_1 \dots z_m\right] \\ |\lambda_j| &< 1 \text{ for } j = 1, \dots, m \\ \lambda_j &= \frac{1}{z_j} \text{ for } j = 1, \dots, m \\ z_0 &= \text{ constant} \end{aligned}$$

The solution of the problem becomes

$$(1 - \lambda_1 L) \cdots (1 - \lambda_m L) y_t = \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} \lambda_j^k a_{t+k}$$

6.1 Transforming Discounted to Undiscounted Problem

Discounted problems can always be converted into undiscounted problems via a simple transformation.

Consider problem (1) with $0 < \beta < 1$.

Define the transformed variables

$$\tilde{a}_t = \beta^{t/2} a_t, \ \tilde{y}_t = \beta^{t/2} y_t \tag{21}$$

Then notice that $\beta^t \left[d\left(L\right) y_t\right]^2 = \left[\tilde{d}\left(L\right) \tilde{y}_t\right]^2$ with $\tilde{d}\left(L\right) = \sum_{j=0}^{m} \tilde{d}_j \, L^j$ and $\tilde{d}_j = \beta^{j/2} d_j$.

Then the original criterion function (1) is equivalent to

$$\lim_{N \to \infty} \sum_{t=0}^{N} \{\tilde{a}_t \, \tilde{y}_t - \frac{1}{2} h \, \tilde{y}_t^2 - \frac{1}{2} [\tilde{d}\left(L\right) \tilde{y}_t]^2\} \tag{22}$$

which is to be maximized over sequences $\{\tilde{y}_t, \ t = 0, ...\}$ subject to $\tilde{y}_{-1}, \cdots, \tilde{y}_{-m}$ given and $\{\tilde{a}_t, \ t = 1, ...\}$ a known bounded sequence.

The Euler equation for this problem is $\left[h + \tilde{d}\left(L^{-1}\right) \tilde{d}\left(L\right)\right] \tilde{y}_t = \tilde{a}_t$.

The solution is

$$\left(1 - \tilde{\lambda}_1 L\right) \cdots \left(1 - \tilde{\lambda}_m L\right) \tilde{y}_t = \sum_{j=1}^{m} \tilde{A}_j \sum_{k=0}^{\infty} \tilde{\lambda}_j^k \, \tilde{a}_{t+k}$$

or

$$\tilde{y}_t = \tilde{f}_1 \, \tilde{y}_{t-1} + \cdots + \tilde{f}_m \, \tilde{y}_{t-m} + \sum_{j=1}^{m} \tilde{A}_j \sum_{k=0}^{\infty} \tilde{\lambda}_j^k \, \tilde{a}_{t+k}, \tag{23}$$

where $\tilde{c}\left(z^{-1}\right) \tilde{c}\left(z\right) = h + \tilde{d}\left(z^{-1}\right) \tilde{d}\left(z\right)$, and where

$$\left[(-1)^m \, \tilde{z}_0 \tilde{z}_1 \ldots \tilde{z}_m\right]^{1/2} \left(1 - \tilde{\lambda}_1 \, z\right) \ldots \left(1 - \tilde{\lambda}_m \, z\right) = \tilde{c}\left(z\right), \ \text{where} \ \ |\tilde{\lambda}_j| < 1$$

We leave it to the reader to show that (23) implies the equivalent form of the solution

$$y_t = f_1 \, y_{t-1} + \cdots + f_m \, y_{t-m} + \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} \left(\lambda_j \, \beta\right)^k a_{t+k}$$

where

$$f_j = \tilde{f}_j \, \beta^{-j/2}, \ A_j = \tilde{A}_j, \ \lambda_j = \tilde{\lambda}_j \, \beta^{-1/2} \tag{24}$$

The transformations (21) and the inverse formulas (24) allow us to solve a discounted problem by first solving a related undiscounted problem.

7 Implementation

Here's the code that computes solutions to the LQ problem using the methods described above.

In [2]:
```
import numpy as np
import scipy.stats as spst
import scipy.linalg as la

class LQFilter:

    def __init__(self, d, h, y_m, r=None, h_eps=None, β=None):
        """

            Parameters
            ----------
                d : list or numpy.array (1-D or a 2-D column vector)
                        The order of the coefficients: [d_0, d_1, ..., d_m]
                h : scalar
                      Parameter of the objective function (corresponding to the
                        quadratic term)
                y_m : list or numpy.array (1-D or a 2-D column vector)
                        Initial conditions for y
                r : list or numpy.array (1-D or a 2-D column vector)
                        The order of the coefficients: [r_0, r_1, ..., r_k]
                        (optional, if not defined -> deterministic problem)
                β : scalar
                        Discount factor (optional, default value is one)
            """

            self.h = h
            self.d = np.asarray(d)
            self.m = self.d.shape[0] - 1

            self.y_m = np.asarray(y_m)

            if self.m == self.y_m.shape[0]:
                self.y_m = self.y_m.reshape(self.m, 1)
            else:
                raise ValueError("y_m must be of length m = {self.m:d}")

            #-----------------------------------------------
            # Define the coefficients of ☒ upfront
            #-----------------------------------------------
            ϕ = np.zeros(2 * self.m + 1)
            for i in range(- self.m, self.m + 1):
                ϕ[self.m - i] = np.sum(np.diag(self.d.reshape(self.m + 1, 1) \
                                               @ self.d.reshape(1, self.m + 1),
                                               k=-i
                                               )
                                       )
            ϕ[self.m] = ϕ[self.m] + self.h
            self.ϕ = ϕ

            #-----------------------------------------------------
            # If r is given calculate the vector ☒_r
            #-----------------------------------------------------
            if r is None:
                pass
            else:
                self.r = np.asarray(r)
                self.k = self.r.shape[0] - 1
                ϕ_r = np.zeros(2 * self.k + 1)
```

```
            for i in range(- self.k, self.k + 1):
                ϕ_r[self.k - i] = np.sum(np.diag(self.r.reshape(self.k + 1,
↪1) \
                                                 @ self.r.reshape(1, self.k + 1),
                                                       k=-i
                                                       )
                                         )
            if h_eps is None:
                self.ϕ_r = ϕ_r
            else:
                ϕ_r[self.k] = ϕ_r[self.k] + h_eps
                self.ϕ_r = ϕ_r

        #-----------------------------------------------------
        # If β is given, define the transformed variables
        #-----------------------------------------------------
        if β is None:
            self.β = 1
        else:
            self.β = β
            self.d = self.β**(np.arange(self.m + 1)/2) * self.d
            self.y_m = self.y_m * (self.β**(- np.arange(1, self.m + 1)/2)) \
                                  .reshape(self.m, 1)

    def construct_W_and_Wm(self, N):
        """
      This constructs the matrices W and W_m for a given number of periods N
        """

        m = self.m
        d = self.d

        W = np.zeros((N + 1, N + 1))
        W_m = np.zeros((N + 1, m))

        #---------------------------------------
        # Terminal conditions
        #---------------------------------------

        D_m1 = np.zeros((m + 1, m + 1))
        M = np.zeros((m + 1, m))

        # (1) Constuct the D_{m+1} matrix using the formula

        for j in range(m + 1):
            for k in range(j, m + 1):
                D_m1[j, k] = d[:j + 1] @ d[k - j: k + 1]

        # Make the matrix symmetric
        D_m1 = D_m1 + D_m1.T - np.diag(np.diag(D_m1))

        # (2) Construct the M matrix using the entries of D_m1

        for j in range(m):
            for i in range(j + 1, m + 1):
                M[i, j] = D_m1[i - j - 1, m]

        #-----------------------------------------------
```

```
        # Euler equations for t = 0, 1, ..., N-(m+1)
        #-------------------------------------------
        ϕ = self.ϕ

        W[:(m + 1), :(m + 1)] = D_m1 + self.h * np.eye(m + 1)
        W[:(m + 1), (m + 1):(2 * m + 1)] = M

        for i, row in enumerate(np.arange(m + 1, N + 1 - m)):
            W[row, (i + 1):(2 * m + 2 + i)] = ϕ

        for i in range(1, m + 1):
            W[N - m + i, -(2 * m + 1 - i):] = ϕ[:-i]

        for i in range(m):
            W_m[N - i, :(m - i)] = ϕ[(m + 1 + i):]

        return W, W_m

    def roots_of_characteristic(self):
        """
        This function calculates z_0 and the 2m roots of the characteristic
        equation associated with the Euler equation (1.7)

        Note:
        ------
        numpy.poly1d(roots, True) defines a polynomial using its roots
↪that can
        be evaluated at any point. If x_1, x_2, ... , x_m are the roots then
            p(x) = (x - x_1)(x - x_2)...(x - x_m)
        """
        m = self.m
        ϕ = self.ϕ

        # Calculate the roots of the 2m-polynomial
        roots = np.roots(ϕ)
        # Sort the roots according to their length (in descending order)
        roots_sorted = roots[np.argsort(abs(roots))[::-1]]

        z_0 = ϕ.sum() / np.poly1d(roots, True)(1)
        z_1_to_m = roots_sorted[:m]     # We need only those outside the
↪unit circle

        λ = 1 / z_1_to_m

        return z_1_to_m, z_0, λ

    def coeffs_of_c(self):
        '''
        This function computes the coefficients {c_j, j = 0, 1, ..., m} for
                c(z) = sum_{j = 0}^{m} c_j z^j

        Based on the expression (1.9). The order is
            c_coeffs = [c_0, c_1, ..., c_{m-1}, c_m]
        '''
        z_1_to_m, z_0 = self.roots_of_characteristic()[:2]

        c_0 = (z_0 * np.prod(z_1_to_m).real * (- 1)**self.m)**(.5)
        c_coeffs = np.poly1d(z_1_to_m, True).c * z_0 / c_0
```

```
            return c_coeffs[::-1]

        def solution(self):
            """
            This function calculates {λ_j, j=1,...,m} and {A_j, j=1,...,m}
            of the expression (1.15)
            """
            λ = self.roots_of_characteristic()[2]
            c_0 = self.coeffs_of_c()[-1]

            A = np.zeros(self.m, dtype=complex)
            for j in range(self.m):
                denom = 1 - λ/λ[j]
                A[j] = c_0**(-2) / np.prod(denom[np.arange(self.m) != j])

            return λ, A

        def construct_V(self, N):
            '''
            This function constructs the covariance matrix for x^N (see
↪section 6)
            for a given period N
            '''
            V = np.zeros((N, N))
            ϕ_r = self.ϕ_r

            for i in range(N):
                for j in range(N):
                    if abs(i-j) <= self.k:
                        V[i, j] = ϕ_r[self.k + abs(i-j)]

            return V

        def simulate_a(self, N):
            """
            Assuming that the u's are normal, this method draws a random path
            for x^N
            """
            V = self.construct_V(N + 1)
            d = spst.multivariate_normal(np.zeros(N + 1), V)

            return d.rvs()

        def predict(self, a_hist, t):
            """
            This function implements the prediction formula discussed in
↪section 6 (1.59)
            It takes a realization for a^N, and the period in which the
↪prediction is
            formed

            Output:  E[abar | a_t, a_{t-1}, ..., a_1, a_0]
            """

            N = np.asarray(a_hist).shape[0] - 1
            a_hist = np.asarray(a_hist).reshape(N + 1, 1)
            V = self.construct_V(N + 1)
```

```
        aux_matrix = np.zeros((N + 1, N + 1))
        aux_matrix[:(t + 1), :(t + 1)] = np.eye(t + 1)
        L = la.cholesky(V).T
        Ea_hist = la.inv(L) @ aux_matrix @ L @ a_hist

        return Ea_hist

    def optimal_y(self, a_hist, t=None):
        """
        - if t is NOT given it takes a_hist (list or numpy.array) as a
          deterministic a_t
        - if t is given, it solves the combined control prediction problem
          (section 7)(by default, t == None -> deterministic)

        for a given sequence of a_t (either deterministic or a particular
       realization), it calculates the optimal y_t sequence using the method
        of the lecture

        Note:
        ------
       scipy.linalg.lu normalizes L, U so that L has unit diagonal elements
        To make things consistent with the lecture, we need an auxiliary
        diagonal matrix D which renormalizes L and U
        """

        N = np.asarray(a_hist).shape[0] - 1
        W, W_m = self.construct_W_and_Wm(N)

        L, U = la.lu(W, permute_l=True)
        D = np.diag(1 / np.diag(U))
        U = D @ U
        L = L @ np.diag(1 / np.diag(D))

        J = np.fliplr(np.eye(N + 1))

        if t is None:   # If the problem is deterministic

            a_hist = J @ np.asarray(a_hist).reshape(N + 1, 1)

            #--------------------------------------------
            # Transform the 'a' sequence if β is given
            #--------------------------------------------
            if self.β != 1:
                a_hist =  a_hist * (self.β**(np.arange(N + 1) / 2))[::-1] \
                                 .reshape(N + 1, 1)

          a_bar = a_hist - W_m @ self.y_m           # a_bar from the lecture
            Uy = np.linalg.solve(L, a_bar)             # U @ y_bar = L^{-1}
           y_bar = np.linalg.solve(U, Uy)             # y_bar = U^{-1}L^{-1}

            # Reverse the order of y_bar with the matrix J
            J = np.fliplr(np.eye(N + self.m + 1))
            # y_hist : concatenated y_m and y_bar
            y_hist = J @ np.vstack([y_bar, self.y_m])

            #--------------------------------------------
            # Transform the optimal sequence back if β is given
```

```
            #-----------------------------------------
            if self.β != 1:
              y_hist = y_hist * (self.β**(- np.arange(-self.m, N + 1)/2)) \
                                  .reshape(N + 1 + self.m, 1)

            return y_hist, L, U, y_bar

        else:           # If the problem is stochastic and we look at it

            Ea_hist = self.predict(a_hist, t).reshape(N + 1, 1)
            Ea_hist = J @ Ea_hist

          a_bar = Ea_hist - W_m @ self.y_m          # a_bar from the lecture
            Uy = np.linalg.solve(L, a_bar)             # U @ y_bar = L^{-1}
           y_bar = np.linalg.solve(U, Uy)            # y_bar = U^{-1}L^{-1}

            # Reverse the order of y_bar with the matrix J
            J = np.fliplr(np.eye(N + self.m + 1))
            # y_hist : concatenated y_m and y_bar
            y_hist = J @ np.vstack([y_bar, self.y_m])

            return y_hist, L, U, y_bar
```

7.1 Example

In this application, we'll have one lag, with

$$d(L)y_t = \gamma(I - L)y_t = \gamma(y_t - y_{t-1})$$

Suppose for the moment that $\gamma = 0$.

Then the intertemporal component of the LQ problem disappears, and the agent simply wants to maximize $a_t y_t - h y_t^2/2$ in each period.

This means that the agent chooses $y_t = a_t/h$.

In the following we'll set $h = 1$, so that the agent just wants to track the $\{a_t\}$ process.

However, as we increase γ, the agent gives greater weight to a smooth time path.

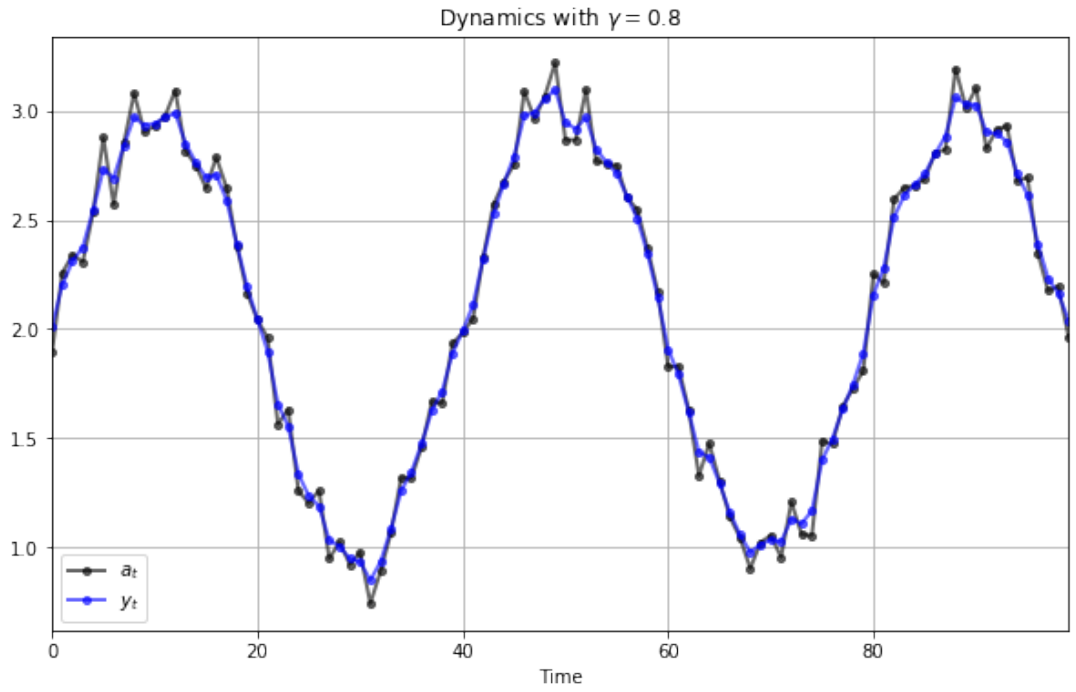
Hence $\{y_t\}$ evolves as a smoothed version of $\{a_t\}$.

The $\{a_t\}$ sequence we'll choose as a stationary cyclic process plus some white noise.

Here's some code that generates a plot when $\gamma = 0.8$

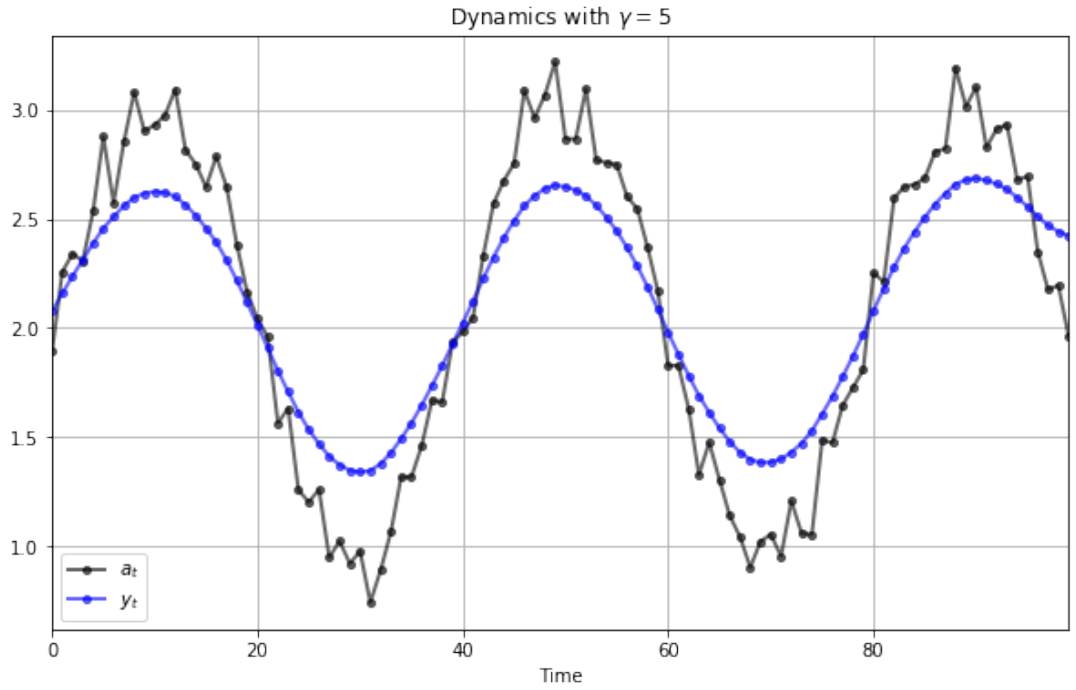
```
In [3]: # Set seed and generate a_t sequence
        np.random.seed(123)
        n = 100
        a_seq = np.sin(np.linspace(0, 5 * np.pi, n)) + 2 + 0.1 * np.random.randn(n)

        def plot_simulation(γ=0.8, m=1, h=1, y_m=2):

            d = γ * np.asarray([1, -1])
            y_m = np.asarray(y_m).reshape(m, 1)

            testlq = LQFilter(d, h, y_m)
```

```
    y_hist, L, U, y = testlq.optimal_y(a_seq)
    y = y[::-1]  # Reverse y

    # Plot simulation results

    fig, ax = plt.subplots(figsize=(10, 6))
    p_args = {'lw' : 2, 'alpha' : 0.6}
    time = range(len(y))
    ax.plot(time, a_seq / h, 'k-o', ms=4, lw=2, alpha=0.6, label='$a_t$')
    ax.plot(time, y, 'b-o', ms=4, lw=2, alpha=0.6, label='$y_t$')
    ax.set(title=rf'Dynamics with $\gamma = {γ}$',
           xlabel='Time',
           xlim=(0, max(time))
          )
    ax.legend()
    ax.grid()
    plt.show()

plot_simulation()
```



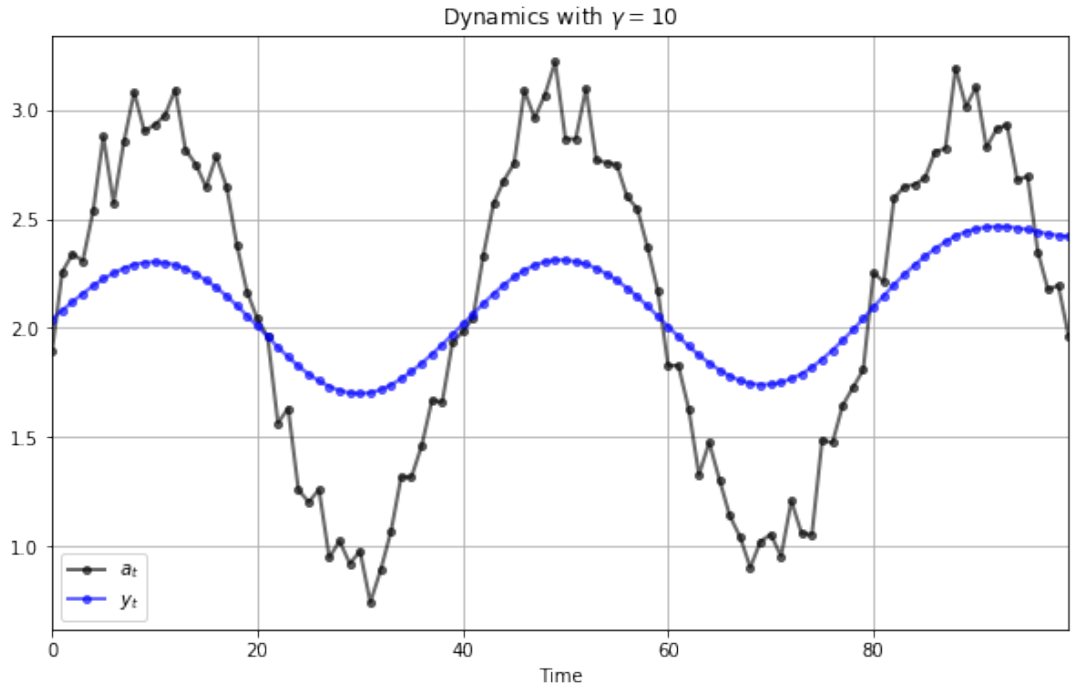
Here's what happens when we change γ to 5.0

```
In [4]: plot_simulation(γ=5)
```



And here's $\gamma = 10$

```
In [5]: plot_simulation(γ=10)
```



8 Exercises

8.1 Exercise 1

Consider solving a discounted version ($\beta < 1$) of problem (1), as follows.

Convert (1) to the undiscounted problem (22).

Let the solution of (22) in feedback form be

$$(1 - \tilde{\lambda}_1 L) \cdots (1 - \tilde{\lambda}_m L)\tilde{y}_t = \sum_{j=1}^{m} \tilde{A}_j \sum_{k=0}^{\infty} \tilde{\lambda}_j^k \tilde{a}_{t+k}$$

or

$$\tilde{y}_t = \tilde{f}_1 \tilde{y}_{t-1} + \cdots + \tilde{f}_m \tilde{y}_{t-m} + \sum_{j=1}^{m} \tilde{A}_j \sum_{k=0}^{\infty} \tilde{\lambda}_j^k \tilde{a}_{t+k} \tag{25}$$

Here

- $h + \tilde{d}(z^{-1})\tilde{d}(z) = \tilde{c}(z^{-1})\tilde{c}(z)$
- $\tilde{c}(z) = [(-1)^m \tilde{z}_0 \tilde{z}_1 \cdots \tilde{z}_m]^{1/2}(1 - \tilde{\lambda}_1 z) \cdots (1 - \tilde{\lambda}_m z)$

where the $\tilde{z}_j$ are the zeros of $h + \tilde{d}(z^{-1})\,\tilde{d}(z)$.

Prove that (25) implies that the solution for y_t in feedback form is

$$y_t = f_1 y_{t-1} + \ldots + f_m y_{t-m} + \sum_{j=1}^{m} A_j \sum_{k=0}^{\infty} \beta^k \lambda_j^k a_{t+k}$$

where $f_j = \tilde{f}_j \beta^{-j/2}$, $A_j = \tilde{A}_j$, and $\lambda_j = \tilde{\lambda}_j \beta^{-1/2}$.

8.2 Exercise 2

Solve the optimal control problem, maximize

$$\sum_{t=0}^{2} \left\{ a_t y_t - \frac{1}{2}[(1 - 2L)y_t]^2 \right\}$$

subject to y_{-1} given, and $\{a_t\}$ a known bounded sequence.

Express the solution in the "feedback form" (20), giving numerical values for the coefficients.

Make sure that the boundary conditions (5) are satisfied.

(Note: this problem differs from the problem in the text in one important way: instead of $h > 0$ in (1), $h = 0$. This has an important influence on the solution.)

8.3 Exercise 3

Solve the infinite time-optimal control problem to maximize

$$\lim_{N\to\infty} \sum_{t=0}^{N} -\frac{1}{2}[(1-2L)y_t]^2,$$

subject to y_{-1} given. Prove that the solution is

$$y_t = 2y_{t-1} = 2^{t+1}y_{-1} \qquad t > 0$$

8.4 Exercise 4

Solve the infinite time problem, to maximize

$$\lim_{N\to\infty} \sum_{t=0}^{N} (.0000001)\, y_t^2 - \frac{1}{2}[(1-2L)y_t]^2$$

subject to y_{-1} given. Prove that the solution $y_t = 2y_{t-1}$ violates condition (12), and so is not optimal.

Prove that the optimal solution is approximately $y_t = .5y_{t-1}$.

References

[1] Papoulis Athanasios and S Unnikrishna Pillai. *Probability, random variables, and stochastic processes*. Mc-Graw Hill, 1991.

[2] Lars Peter Hansen and Thomas J Sargent. Formulating and estimating dynamic linear rational expectations models. *Journal of Economic Dynamics and control*, 2:7–46, 1980.

[3] John F Muth. Optimal properties of exponentially weighted forecasts. *Journal of the american statistical association*, 55(290):299–306, 1960.

[4] Sophocles J Orfanidis. *Optimum Signal Processing: An Introduction*. McGraw Hill Publishing, New York, New York, 1988.

[5] Thomas J Sargent. *Macroeconomic Theory*. Academic Press, New York, 2nd edition, 1987.

[6] Peter Whittle. *Prediction and regulation by linear least-square methods*. English Univ. Press, 1963.