Part VI

Time Series Models

CHAPTER

TWENTYSEVEN

COVARIANCE STATIONARY PROCESSES

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

27.1 Overview

In this lecture we study covariance stationary linear stochastic processes, a class of models routinely used to study economic and financial time series.

This class has the advantage of being

1. simple enough to be described by an elegant and comprehensive theory
2. relatively broad in terms of the kinds of dynamics it can represent

We consider these models in both the time and frequency domain.

27.1.1 ARMA Processes

We will focus much of our attention on linear covariance stationary models with a finite number of parameters.

In particular, we will study stationary ARMA processes, which form a cornerstone of the standard theory of time series analysis.

Every ARMA process can be represented in linear state space form.

However, ARMA processes have some important structure that makes it valuable to study them separately.

27.1.2 Spectral Analysis

Analysis in the frequency domain is also called spectral analysis.

In essence, spectral analysis provides an alternative representation of the autocovariance function of a covariance stationary process.

Having a second representation of this important object

- shines a light on the dynamics of the process in question
- allows for a simpler, more tractable representation in some important cases

The famous *Fourier transform* and its inverse are used to map between the two representations.

27.1.3 Other Reading

For supplementary reading, see

- [LS18], chapter 2
- [Sar87], chapter 11
- John Cochrane's notes on time series analysis, chapter 8
- [Shi95], chapter 6
- [CC08], all

Let's start with some imports:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import quantecon as qe
```

27.2 Introduction

Consider a sequence of random variables $\{X_t\}$ indexed by $t \in \mathbb{Z}$ and taking values in $\mathbb{R}$.

Thus, $\{X_t\}$ begins in the infinite past and extends to the infinite future — a convenient and standard assumption.

As in other fields, successful economic modeling typically assumes the existence of features that are constant over time.

If these assumptions are correct, then each new observation $X_t, X_{t+1}, \ldots$ can provide additional information about the time-invariant features, allowing us to learn from as data arrive.

For this reason, we will focus in what follows on processes that are *stationary* — or become so after a transformation (see for example *this lecture*).

27.2.1 Definitions

A real-valued stochastic process $\{X_t\}$ is called *covariance stationary* if

1. Its mean $\mu := \mathbb{E}X_t$ does not depend on t.
2. For all k in $\mathbb{Z}$, the k-th autocovariance $\gamma(k) := \mathbb{E}(X_t - \mu)(X_{t+k} - \mu)$ is finite and depends only on k.

The function $\gamma \colon \mathbb{Z} \to \mathbb{R}$ is called the *autocovariance function* of the process.

Throughout this lecture, we will work exclusively with zero-mean (i.e., $\mu = 0$) covariance stationary processes.

The zero-mean assumption costs nothing in terms of generality since working with non-zero-mean processes involves no more than adding a constant.

27.2.2 Example 1: White Noise

Perhaps the simplest class of covariance stationary processes is the white noise processes.

A process $\{\epsilon_t\}$ is called a *white noise process* if

1. $\mathbb{E}\epsilon_t = 0$
2. $\gamma(k) = \sigma^2 \mathbf{1}\{k = 0\}$ for some $\sigma > 0$

(Here $\mathbf{1}\{k = 0\}$ is defined to be 1 if $k = 0$ and zero otherwise)

White noise processes play the role of **building blocks** for processes with more complicated dynamics.

27.2.3 Example 2: General Linear Processes

From the simple building block provided by white noise, we can construct a very flexible family of covariance stationary processes — the *general linear processes*

$$X_t = \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j}, \qquad t \in \mathbb{Z} \tag{1}$$

where

- $\{\epsilon_t\}$ is white noise
- $\{\psi_t\}$ is a square summable sequence in $\mathbb{R}$ (that is, $\sum_{t=0}^{\infty} \psi_t^2 < \infty$)

The sequence $\{\psi_t\}$ is often called a *linear filter*.

Equation (1) is said to present a **moving average** process or a moving average representation.

With some manipulations, it is possible to confirm that the autocovariance function for (1) is

$$\gamma(k) = \sigma^2 \sum_{j=0}^{\infty} \psi_j \psi_{j+k} \tag{2}$$

By the Cauchy-Schwartz inequality, one can show that $\gamma(k)$ satisfies equation (2).

Evidently, $\gamma(k)$ does not depend on t.

27.2.4 Wold Representation

Remarkably, the class of general linear processes goes a long way towards describing the entire class of zero-mean covariance stationary processes.

In particular, Wold's decomposition theorem states that every zero-mean covariance stationary process $\{X_t\}$ can be written as

$$X_t = \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j} + \eta_t$$

where

- $\{\epsilon_t\}$ is white noise
- $\{\psi_t\}$ is square summable
- $\psi_0 \epsilon_t$ is the one-step ahead prediction error in forecasting X_t as a linear least-squares function of the infinite history $X_{t-1}, X_{t-2}, \ldots$
- η_t can be expressed as a linear function of $X_{t-1}, X_{t-2}, \ldots$ and is perfectly predictable over arbitrarily long horizons

For the method of constructing a Wold representation, intuition, and further discussion, see [Sar87], p. 286.

27.2.5 AR and MA

General linear processes are a very broad class of processes.

It often pays to specialize to those for which there exists a representation having only finitely many parameters.

(Experience and theory combine to indicate that models with a relatively small number of parameters typically perform better than larger models, especially for forecasting)

One very simple example of such a model is the first-order autoregressive or AR(1) process

$$X_t = \phi X_{t-1} + \epsilon_t \quad \text{where} \quad |\phi| < 1 \quad \text{and } \{\epsilon_t\} \text{ is white noise} \tag{3}$$

By direct substitution, it is easy to verify that $X_t = \sum_{j=0}^{\infty} \phi^j \epsilon_{t-j}$.

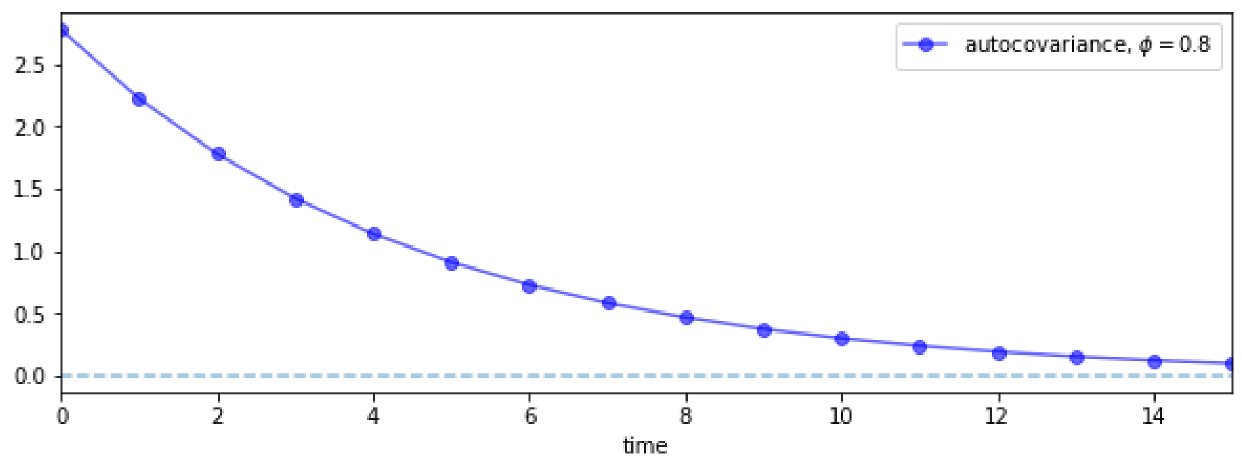
Hence $\{X_t\}$ is a general linear process.

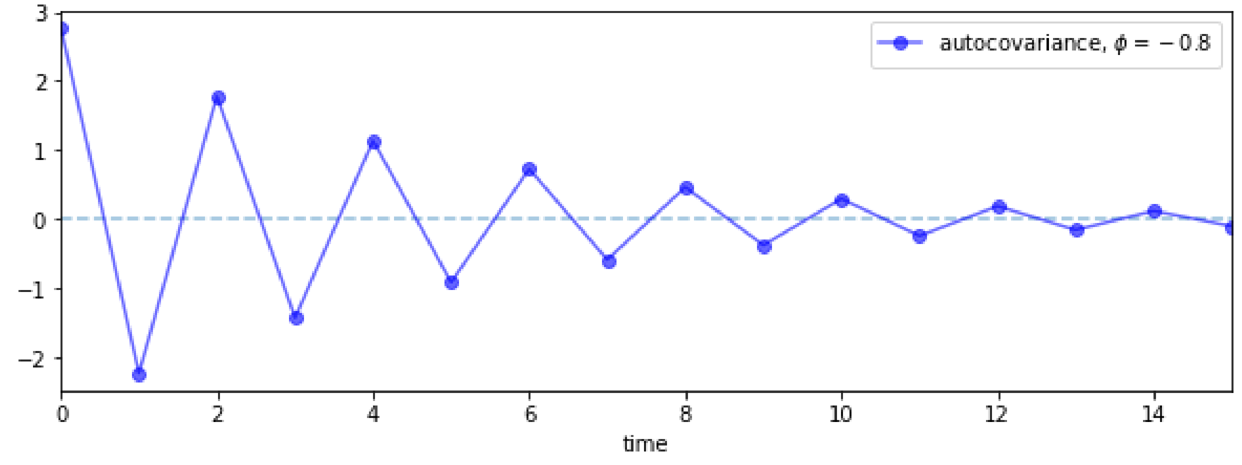
Applying (2) to the previous expression for X_t, we get the AR(1) autocovariance function

$$\gamma(k) = \phi^k \frac{\sigma^2}{1 - \phi^2}, \qquad k = 0, 1, \ldots \tag{4}$$

The next figure plots an example of this function for $\phi = 0.8$ and $\phi = -0.8$ with $\sigma = 1$.

```python
num_rows, num_cols = 2, 1
fig, axes = plt.subplots(num_rows, num_cols, figsize=(10, 8))
plt.subplots_adjust(hspace=0.4)

for i, ϕ in enumerate((0.8, -0.8)):
    ax = axes[i]
    times = list(range(16))
    acov = [ϕ**k / (1 - ϕ**2) for k in times]
    ax.plot(times, acov, 'bo-', alpha=0.6,
            label=f'autocovariance, $\phi = {ϕ:.2}$')
    ax.legend(loc='upper right')
    ax.set(xlabel='time', xlim=(0, 15))
    ax.hlines(0, 0, 15, linestyle='--', alpha=0.5)
plt.show()
```





Another very simple process is the MA(1) process (here MA means “moving average”)

$$X_t = \epsilon_t + \theta \epsilon_{t-1}$$

You will be able to verify that

$$\gamma(0) = \sigma^2 (1 + \theta^2), \quad \gamma(1) = \sigma^2 \theta, \quad \text{and} \quad \gamma(k) = 0 \quad \forall \, k > 1$$

The AR(1) can be generalized to an AR(p) and likewise for the MA(1).

Putting all of this together, we get the

27.2.6 ARMA Processes

A stochastic process $\{X_t\}$ is called an *autoregressive moving average process*, or ARMA(p, q), if it can be written as

$$X_t = \phi_1 X_{t-1} + \cdots + \phi_p X_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + \cdots + \theta_q \epsilon_{t-q} \tag{5}$$

where $\{\epsilon_t\}$ is white noise.

An alternative notation for ARMA processes uses the *lag operator* L.

Def. Given arbitrary variable Y_t, let $L^k Y_t := Y_{t-k}$.

It turns out that

- lag operators facilitate succinct representations for linear stochastic processes
- algebraic manipulations that treat the lag operator as an ordinary scalar are legitimate

Using L, we can rewrite (5) as

$$L^0 X_t - \phi_1 L^1 X_t - \cdots - \phi_p L^p X_t = L^0 \epsilon_t + \theta_1 L^1 \epsilon_t + \cdots + \theta_q L^q \epsilon_t \tag{6}$$

If we let $\phi(z)$ and $\theta(z)$ be the polynomials

$$\phi(z) := 1 - \phi_1 z - \cdots - \phi_p z^p \quad \text{and} \quad \theta(z) := 1 + \theta_1 z + \cdots + \theta_q z^q \tag{7}$$

then (6) becomes

$$\phi(L) X_t = \theta(L) \epsilon_t \tag{8}$$

In what follows we **always assume** that the roots of the polynomial $\phi(z)$ lie outside the unit circle in the complex plane.

This condition is sufficient to guarantee that the ARMA(p, q) process is covariance stationary.

In fact, it implies that the process falls within the class of general linear processes *described above*.

That is, given an ARMA(p, q) process $\{X_t\}$ satisfying the unit circle condition, there exists a square summable sequence $\{\psi_t\}$ with $X_t = \sum_{j=0}^{\infty} \psi_j \epsilon_{t-j}$ for all t.

The sequence $\{\psi_t\}$ can be obtained by a recursive procedure outlined on page 79 of [CC08].

The function $t \mapsto \psi_t$ is often called the *impulse response function*.

27.3 Spectral Analysis

Autocovariance functions provide a great deal of information about covariance stationary processes.

In fact, for zero-mean Gaussian processes, the autocovariance function characterizes the entire joint distribution.

Even for non-Gaussian processes, it provides a significant amount of information.

It turns out that there is an alternative representation of the autocovariance function of a covariance stationary process, called the *spectral density*.

At times, the spectral density is easier to derive, easier to manipulate, and provides additional intuition.

27.3.1 Complex Numbers

Before discussing the spectral density, we invite you to recall the main properties of complex numbers (or *skip to the next section*).

It can be helpful to remember that, in a formal sense, complex numbers are just points $(x, y) \in \mathbb{R}^2$ endowed with a specific notion of multiplication.

When (x, y) is regarded as a complex number, x is called the *real part* and y is called the *imaginary part*.

The *modulus* or *absolute value* of a complex number $z = (x, y)$ is just its Euclidean norm in $\mathbb{R}^2$, but is usually written as $|z|$ instead of $\|z\|$.

The product of two complex numbers (x, y) and (u, v) is defined to be $(xu - vy, xv + yu)$, while addition is standard pointwise vector addition.

When endowed with these notions of multiplication and addition, the set of complex numbers forms a field — addition and multiplication play well together, just as they do in $\mathbb{R}$.

The complex number (x, y) is often written as $x + iy$, where i is called the *imaginary unit* and is understood to obey $i^2 = -1$.

The $x + iy$ notation provides an easy way to remember the definition of multiplication given above, because, proceeding naively,

$$(x + iy)(u + iv) = xu - yv + i(xv + yu)$$

Converted back to our first notation, this becomes $(xu - vy, xv + yu)$ as promised.

Complex numbers can be represented in the polar form $re^{i\omega}$ where

$$re^{i\omega} := r(\cos(\omega) + i\sin(\omega)) = x + iy$$

where $x = r\cos(\omega), y = r\sin(\omega)$, and $\omega = \arctan(y/z)$ or $\tan(\omega) = y/x$.

27.3.2 Spectral Densities

Let $\{X_t\}$ be a covariance stationary process with autocovariance function γ satisfying $\sum_k \gamma(k)^2 < \infty$.

The *spectral density* f of $\{X_t\}$ is defined as the discrete time Fourier transform of its autocovariance function γ.

$$f(\omega) := \sum_{k \in \mathbb{Z}} \gamma(k) e^{-i\omega k}, \qquad \omega \in \mathbb{R}$$

(Some authors normalize the expression on the right by constants such as $1/\pi$ — the convention chosen makes little difference provided you are consistent).

Using the fact that γ is *even*, in the sense that $\gamma(t) = \gamma(-t)$ for all t, we can show that

$$f(\omega) = \gamma(0) + 2\sum_{k \geq 1} \gamma(k)\cos(\omega k) \tag{9}$$

It is not difficult to confirm that f is

- real-valued
- even ($f(\omega) = f(-\omega)$), and
- 2π-periodic, in the sense that $f(2\pi + \omega) = f(\omega)$ for all ω

It follows that the values of f on $[0, \pi]$ determine the values of f on all of $\mathbb{R}$ — the proof is an exercise.

For this reason, it is standard to plot the spectral density only on the interval $[0, \pi]$.

27.3.3 Example 1: White Noise

Consider a white noise process $\{\epsilon_t\}$ with standard deviation σ.

It is easy to check that in this case $f(\omega) = \sigma^2$. So f is a constant function.

As we will see, this can be interpreted as meaning that “all frequencies are equally present”.

(White light has this property when frequency refers to the visible spectrum, a connection that provides the origins of the term “white noise”)

27.3.4 Example 2: AR and MA and ARMA

It is an exercise to show that the MA(1) process $X_t = \theta \epsilon_{t-1} + \epsilon_t$ has a spectral density

$$f(\omega) = \sigma^2 (1 + 2\theta \cos(\omega) + \theta^2) \tag{10}$$

With a bit more effort, it's possible to show (see, e.g., p. 261 of [Sar87]) that the spectral density of the AR(1) process $X_t = \phi X_{t-1} + \epsilon_t$ is

$$f(\omega) = \frac{\sigma^2}{1 - 2\phi \cos(\omega) + \phi^2} \tag{11}$$

More generally, it can be shown that the spectral density of the ARMA process (5) is

$$f(\omega) = \left| \frac{\theta(e^{i\omega})}{\phi(e^{i\omega})} \right|^2 \sigma^2 \tag{12}$$

where

- σ is the standard deviation of the white noise process $\{\epsilon_t\}$.
- the polynomials $\phi(\cdot)$ and $\theta(\cdot)$ are as defined in (7).

The derivation of (12) uses the fact that convolutions become products under Fourier transformations.

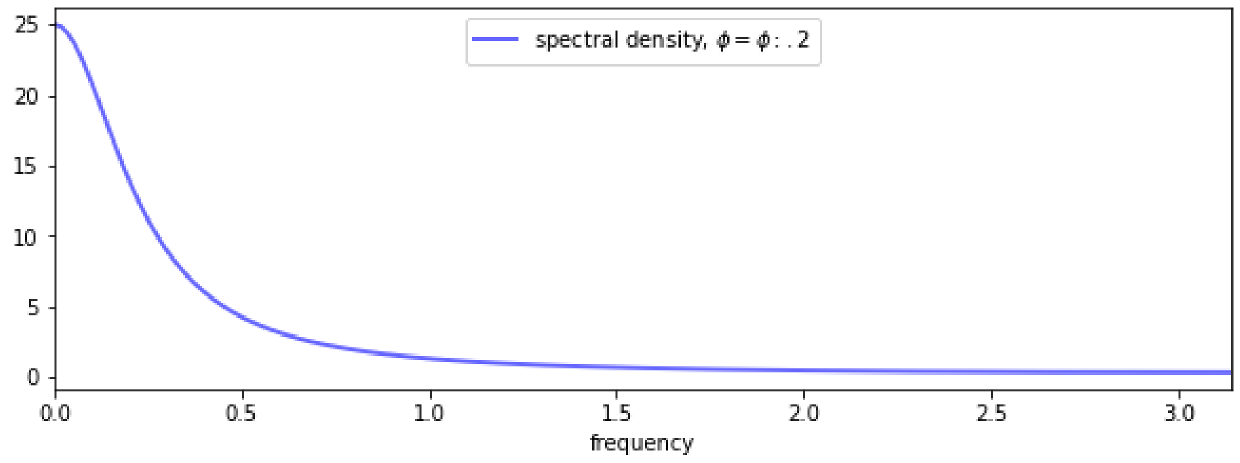
The proof is elegant and can be found in many places — see, for example, [Sar87], chapter 11, section 4.

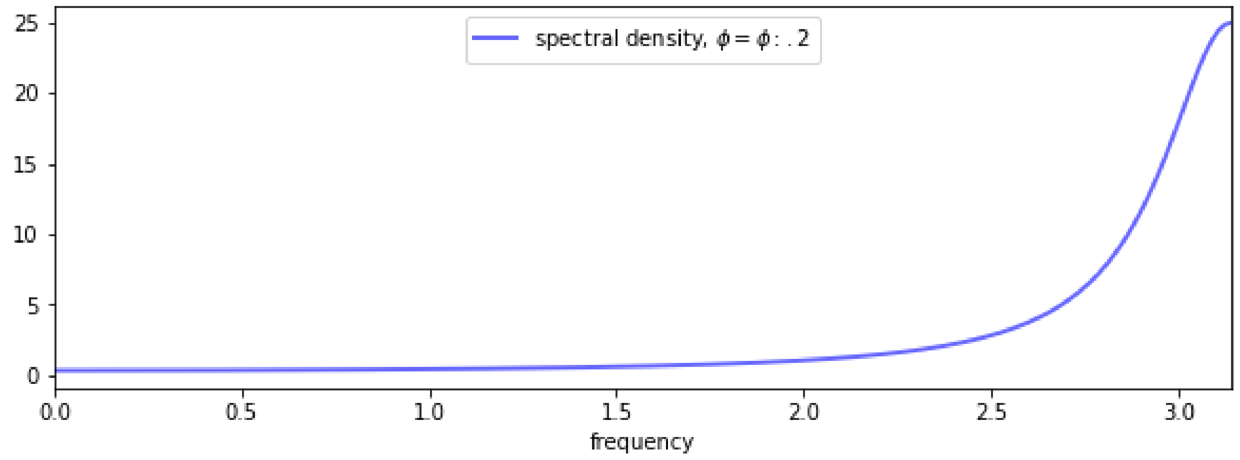
It's a nice exercise to verify that (10) and (11) are indeed special cases of (12).

27.3.5 Interpreting the Spectral Density

Plotting (11) reveals the shape of the spectral density for the AR(1) model when ϕ takes the values 0.8 and -0.8 respectively.

```python
def ar1_sd(ϕ, ω):
    return 1 / (1 - 2 * ϕ * np.cos(ω) + ϕ**2)

ωs = np.linspace(0, np.pi, 180)
num_rows, num_cols = 2, 1
fig, axes = plt.subplots(num_rows, num_cols, figsize=(10, 8))
plt.subplots_adjust(hspace=0.4)

# Autocovariance when phi = 0.8
for i, ϕ in enumerate((0.8, -0.8)):
    ax = axes[i]
    sd = ar1_sd(ϕ, ωs)
    ax.plot(ωs, sd, 'b-', alpha=0.6, lw=2,
            label='spectral density, $\phi = {ϕ:.2}$')
    ax.legend(loc='upper center')
    ax.set(xlabel='frequency', xlim=(0, np.pi))
plt.show()
```





These spectral densities correspond to the autocovariance functions for the AR(1) process shown above.

Informally, we think of the spectral density as being large at those $\omega \in [0, \pi]$ at which the autocovariance function seems approximately to exhibit big damped cycles.

To see the idea, let's consider why, in the lower panel of the preceding figure, the spectral density for the case $\phi = -0.8$ is large at $\omega = \pi$.

Recall that the spectral density can be expressed as

$$f(\omega) = \gamma(0) + 2 \sum_{k \geq 1} \gamma(k) \cos(\omega k) = \gamma(0) + 2 \sum_{k \geq 1} (-0.8)^k \cos(\omega k) \tag{13}$$

When we evaluate this at $\omega = \pi$, we get a large number because $\cos(\pi k)$ is large and positive when $(-0.8)^k$ is positive, and large in absolute value and negative when $(-0.8)^k$ is negative.

Hence the product is always large and positive, and hence the sum of the products on the right-hand side of (13) is large.

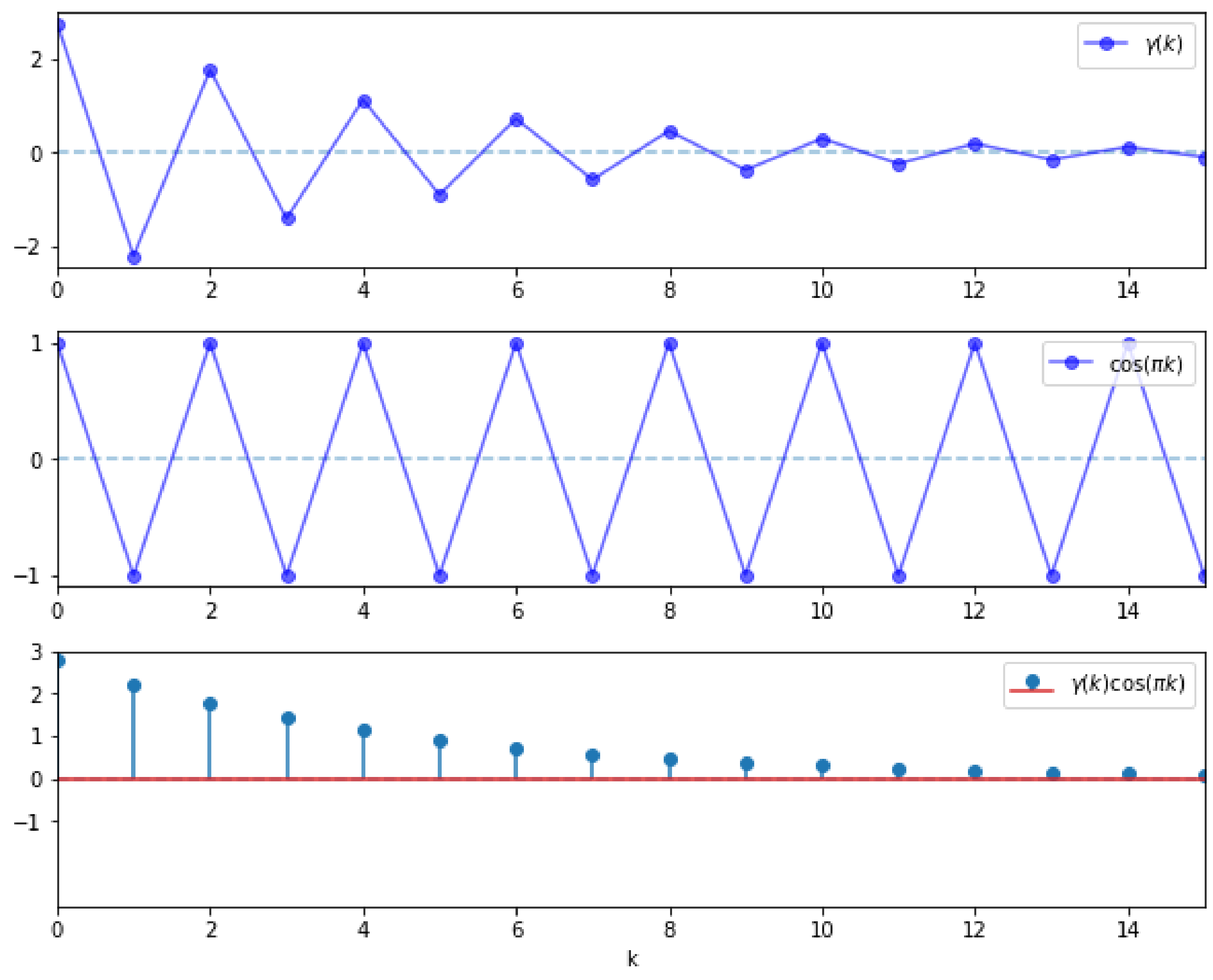
These ideas are illustrated in the next figure, which has k on the horizontal axis.

```
ϕ = -0.8
times = list(range(16))
y1 = [ϕ**k / (1 - ϕ**2) for k in times]
y2 = [np.cos(np.pi * k) for k in times]
```

(continues on next page)

(continued from previous page)

```
y3 = [a * b for a, b in zip(y1, y2)]

num_rows, num_cols = 3, 1
fig, axes = plt.subplots(num_rows, num_cols, figsize=(10, 8))
plt.subplots_adjust(hspace=0.25)

# Autocovariance when ϕ = -0.8
ax = axes[0]
ax.plot(times, y1, 'bo-', alpha=0.6, label='$\gamma(k)$')
ax.legend(loc='upper right')
ax.set(xlim=(0, 15), yticks=(-2, 0, 2))
ax.hlines(0, 0, 15, linestyle='--', alpha=0.5)

# Cycles at frequency π
ax = axes[1]
ax.plot(times, y2, 'bo-', alpha=0.6, label='$\cos(\pi k)$')
ax.legend(loc='upper right')
ax.set(xlim=(0, 15), yticks=(-1, 0, 1))
ax.hlines(0, 0, 15, linestyle='--', alpha=0.5)

# Product
ax = axes[2]
ax.stem(times, y3, label='$\gamma(k) \cos(\pi k)$')
ax.legend(loc='upper right')
ax.set(xlim=(0, 15), ylim=(-3, 3), yticks=(-1, 0, 1, 2, 3))
ax.hlines(0, 0, 15, linestyle='--', alpha=0.5)
ax.set_xlabel("k")

plt.show()
```



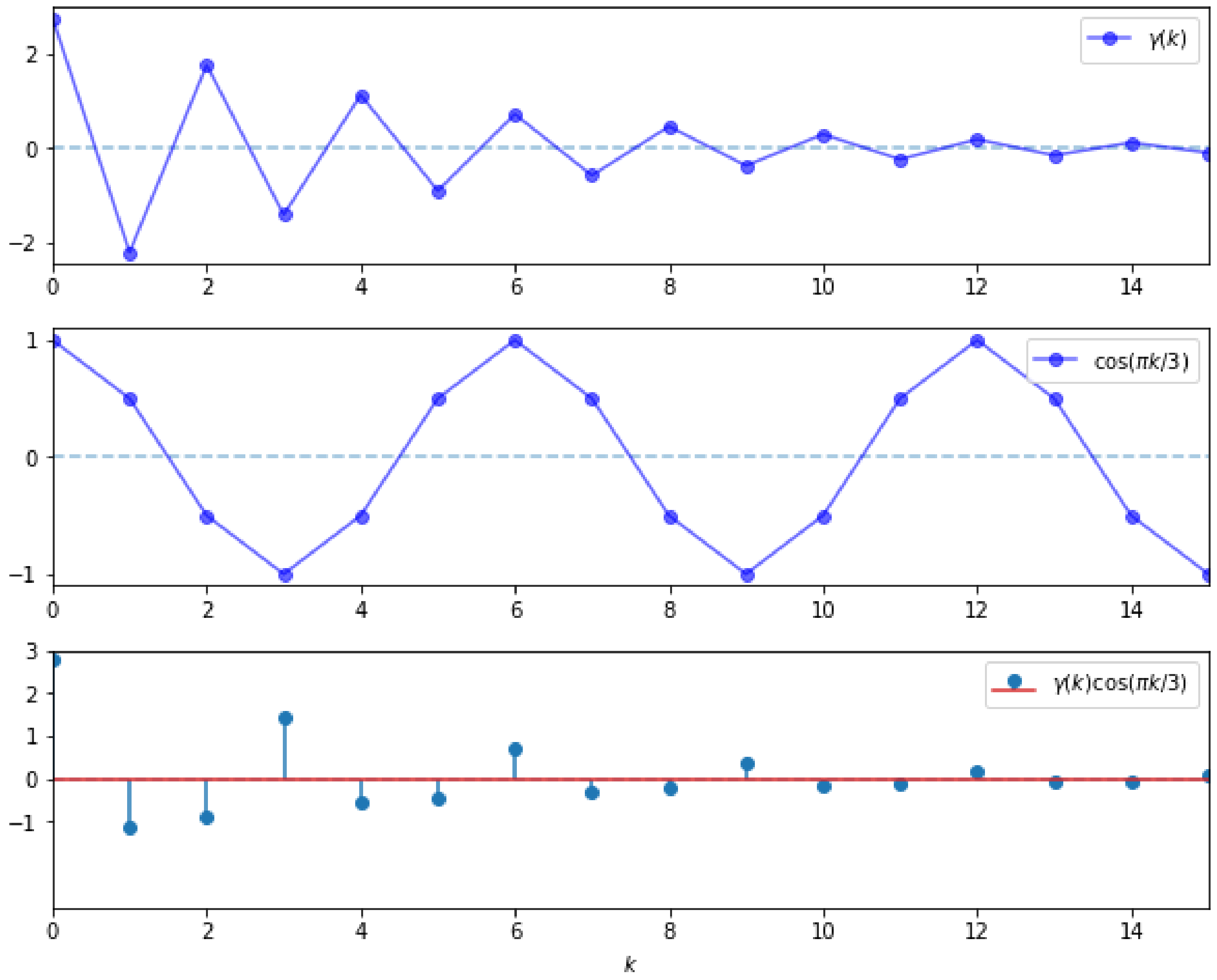
On the other hand, if we evaluate $f(\omega)$ at $\omega = \pi/3$, then the cycles are not matched, the sequence $\gamma(k)\cos(\omega k)$ contains both positive and negative terms, and hence the sum of these terms is much smaller.

```
ϕ = -0.8
times = list(range(16))
y1 = [ϕ**k / (1 - ϕ**2) for k in times]
y2 = [np.cos(np.pi * k/3) for k in times]
y3 = [a * b for a, b in zip(y1, y2)]

num_rows, num_cols = 3, 1
fig, axes = plt.subplots(num_rows, num_cols, figsize=(10, 8))
plt.subplots_adjust(hspace=0.25)

# Autocovariance when phi = -0.8
ax = axes[0]
ax.plot(times, y1, 'bo-', alpha=0.6, label='$\gamma(k)$')
ax.legend(loc='upper right')
ax.set(xlim=(0, 15), yticks=(-2, 0, 2))
ax.hlines(0, 0, 15, linestyle='--', alpha=0.5)

# Cycles at frequency π
ax = axes[1]
ax.plot(times, y2, 'bo-', alpha=0.6, label='$\cos(\pi k/3)$')
```

(continues on next page)

(continued from previous page)

```
ax.legend(loc='upper right')
ax.set(xlim=(0, 15), yticks=(-1, 0, 1))
ax.hlines(0, 0, 15, linestyle='--', alpha=0.5)

# Product
ax = axes[2]
ax.stem(times, y3, label='$\gamma(k) \cos(\pi k/3)$')
ax.legend(loc='upper right')
ax.set(xlim=(0, 15), ylim=(-3, 3), yticks=(-1, 0, 1, 2, 3))
ax.hlines(0, 0, 15, linestyle='--', alpha=0.5)
ax.set_xlabel("$k$")

plt.show()
```



In summary, the spectral density is large at frequencies ω where the autocovariance function exhibits damped cycles.

27.3.6 Inverting the Transformation

We have just seen that the spectral density is useful in the sense that it provides a frequency-based perspective on the autocovariance structure of a covariance stationary process.

Another reason that the spectral density is useful is that it can be "inverted" to recover the autocovariance function via the *inverse Fourier transform*.

In particular, for all $k \in \mathbb{Z}$, we have

$$\gamma(k) = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(\omega) e^{i\omega k} d\omega \tag{14}$$

This is convenient in situations where the spectral density is easier to calculate and manipulate than the autocovariance function.

(For example, the expression (12) for the ARMA spectral density is much easier to work with than the expression for the ARMA autocovariance)

27.3.7 Mathematical Theory

This section is loosely based on [Sar87], p. 249-253, and included for those who

- would like a bit more insight into spectral densities
- and have at least some background in Hilbert space theory

Others should feel free to skip to the *next section* — none of this material is necessary to progress to computation.

Recall that every separable Hilbert space H has a countable orthonormal basis $\{h_k\}$.

The nice thing about such a basis is that every $f \in H$ satisfies

$$f = \sum_k \alpha_k h_k \quad \text{where} \quad \alpha_k := \langle f, h_k \rangle \tag{15}$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product in H.

Thus, f can be represented to any degree of precision by linearly combining basis vectors.

The scalar sequence $\alpha = \{\alpha_k\}$ is called the *Fourier coefficients* of f, and satisfies $\sum_k |\alpha_k|^2 < \infty$.

In other words, α is in ℓ_2, the set of square summable sequences.

Consider an operator T that maps $\alpha \in \ell_2$ into its expansion $\sum_k \alpha_k h_k \in H$.

The Fourier coefficients of $T\alpha$ are just $\alpha = \{\alpha_k\}$, as you can verify by confirming that $\langle T\alpha, h_k \rangle = \alpha_k$.

Using elementary results from Hilbert space theory, it can be shown that

- T is one-to-one — if α and β are distinct in ℓ_2, then so are their expansions in H.
- T is onto — if $f \in H$ then its preimage in ℓ_2 is the sequence α given by $\alpha_k = \langle f, h_k \rangle$.
- T is a linear isometry — in particular, $\langle \alpha, \beta \rangle = \langle T\alpha, T\beta \rangle$.

Summarizing these results, we say that any separable Hilbert space is isometrically isomorphic to ℓ_2.

In essence, this says that each separable Hilbert space we consider is just a different way of looking at the fundamental space ℓ_2.

With this in mind, let's specialize to a setting where

- $\gamma \in \ell_2$ is the autocovariance function of a covariance stationary process, and f is the spectral density.

- $H = L_2$, where L_2 is the set of square summable functions on the interval $[-\pi, \pi]$, with inner product $\langle g, h \rangle = \int_{-\pi}^{\pi} g(\omega)h(\omega)d\omega$.
- $\{h_k\}$ = the orthonormal basis for L_2 given by the set of trigonometric functions.

$$h_k(\omega) = \frac{e^{i\omega k}}{\sqrt{2\pi}}, \quad k \in \mathbb{Z}, \quad \omega \in [-\pi, \pi]$$

Using the definition of T from above and the fact that f is even, we now have

$$T\gamma = \sum_{k\in\mathbb{Z}} \gamma(k) \frac{e^{i\omega k}}{\sqrt{2\pi}} = \frac{1}{\sqrt{2\pi}} f(\omega) \tag{16}$$

In other words, apart from a scalar multiple, the spectral density is just a transformation of $\gamma \in \ell_2$ under a certain linear isometry — a different way to view γ.

In particular, it is an expansion of the autocovariance function with respect to the trigonometric basis functions in L_2.

As discussed above, the Fourier coefficients of $T\gamma$ are given by the sequence γ, and, in particular, $\gamma(k) = \langle T\gamma, h_k \rangle$.

Transforming this inner product into its integral expression and using (16) gives (14), justifying our earlier expression for the inverse transform.

27.4 Implementation

Most code for working with covariance stationary models deals with ARMA models.

Python code for studying ARMA models can be found in the `tsa` submodule of statsmodels.

Since this code doesn't quite cover our needs — particularly vis-a-vis spectral analysis — we've put together the module arma.py, which is part of QuantEcon.py package.

The module provides functions for mapping ARMA(p, q) models into their

1. impulse response function
2. simulated time series
3. autocovariance function
4. spectral density

27.4.1 Application

Let's use this code to replicate the plots on pages 68–69 of [LS18].

Here are some functions to generate the plots

```python
def plot_impulse_response(arma, ax=None):
    if ax is None:
        ax = plt.gca()
    yi = arma.impulse_response()
    ax.stem(list(range(len(yi))), yi)
    ax.set(xlim=(-0.5), ylim=(min(yi)-0.1, max(yi)+0.1),
                title='Impulse response', xlabel='time', ylabel='response')
    return ax

def plot_spectral_density(arma, ax=None):
    if ax is None:
```

(continues on next page)

(continued from previous page)

```
        ax = plt.gca()
    w, spect = arma.spectral_density(two_pi=False)
    ax.semilogy(w, spect)
    ax.set(xlim=(0, np.pi), ylim=(0, np.max(spect)),
           title='Spectral density', xlabel='frequency', ylabel='spectrum')
    return ax

def plot_autocovariance(arma, ax=None):
    if ax is None:
        ax = plt.gca()
    acov = arma.autocovariance()
    ax.stem(list(range(len(acov))), acov)
    ax.set(xlim=(-0.5, len(acov) - 0.5), title='Autocovariance',
           xlabel='time', ylabel='autocovariance')
    return ax

def plot_simulation(arma, ax=None):
    if ax is None:
        ax = plt.gca()
    x_out = arma.simulation()
    ax.plot(x_out)
    ax.set(title='Sample path', xlabel='time', ylabel='state space')
    return ax

def quad_plot(arma):
    """
    Plots the impulse response, spectral_density, autocovariance,
    and one realization of the process.

    """
    num_rows, num_cols = 2, 2
    fig, axes = plt.subplots(num_rows, num_cols, figsize=(10, 7))
    plot_functions = [plot_impulse_response,
                      plot_spectral_density,
                      plot_autocovariance,
                      plot_simulation]
    for plot_func, ax in zip(plot_functions, axes.flatten()):
        plot_func(arma, ax)
    plt.tight_layout()
    plt.show()
```

Now let's call these functions to generate plots.

As a warmup, let's make sure things look right when we for the pure white noise model $X_t = \epsilon_t$.

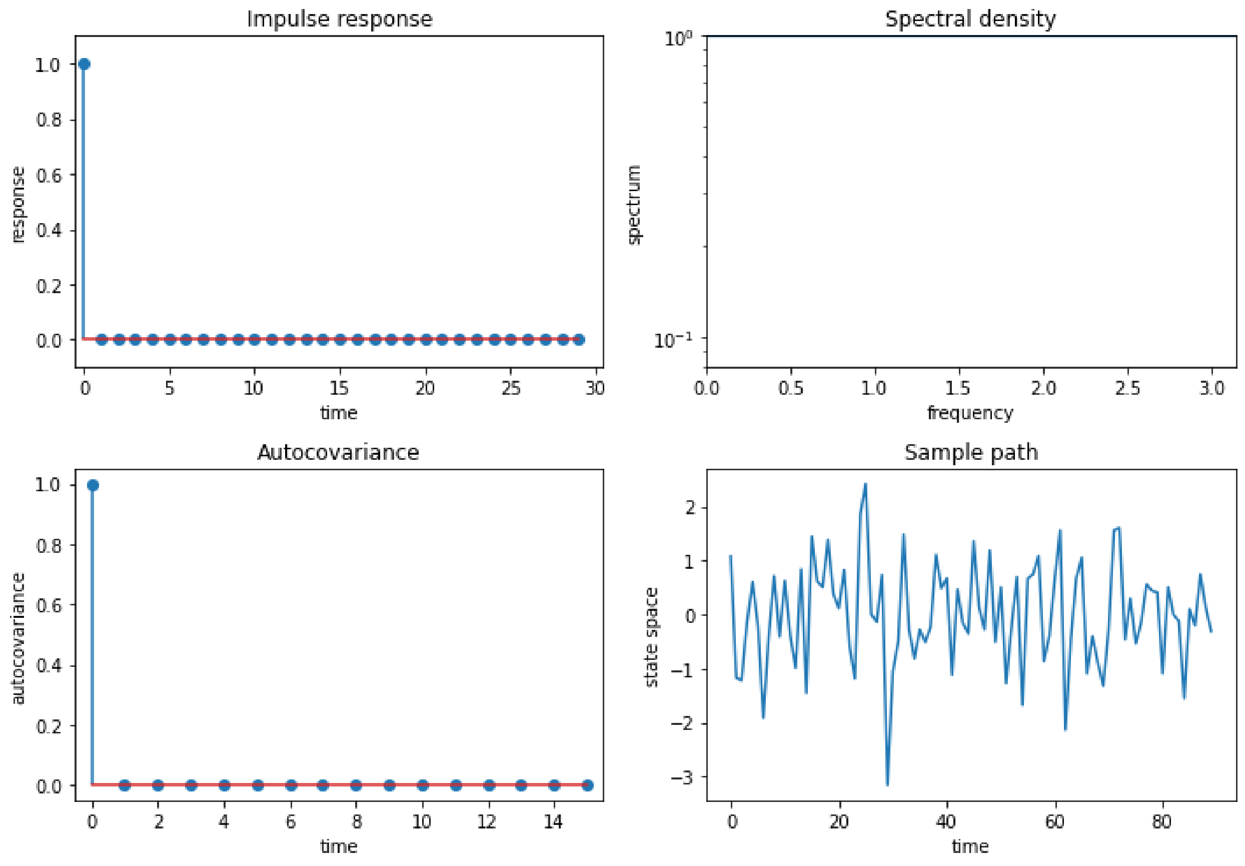
```
ϕ = 0.0
θ = 0.0
arma = qe.ARMA(ϕ, θ)
quad_plot(arma)
```

```
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/numpy/core/_asarray.
↪py:83: ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
<ipython-input-7-4995e3f7b93d>:15: UserWarning: Attempted to set non-positive bottom␣
↪ylim on a log-scaled axis.
Invalid limit will be ignored.
  ax.set(xlim=(0, np.pi), ylim=(0, np.max(spect)),
```

(continues on next page)

(continued from previous page)

```
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/matplotlib/
↪transforms.py:948: ComplexWarning: Casting complex values to real discards the␣
↪imaginary part
  self._points[:, 1] = interval
```



If we look carefully, things look good: the spectrum is the flat line at 10^0 at the very top of the spectrum graphs, which is at it should be.

Also

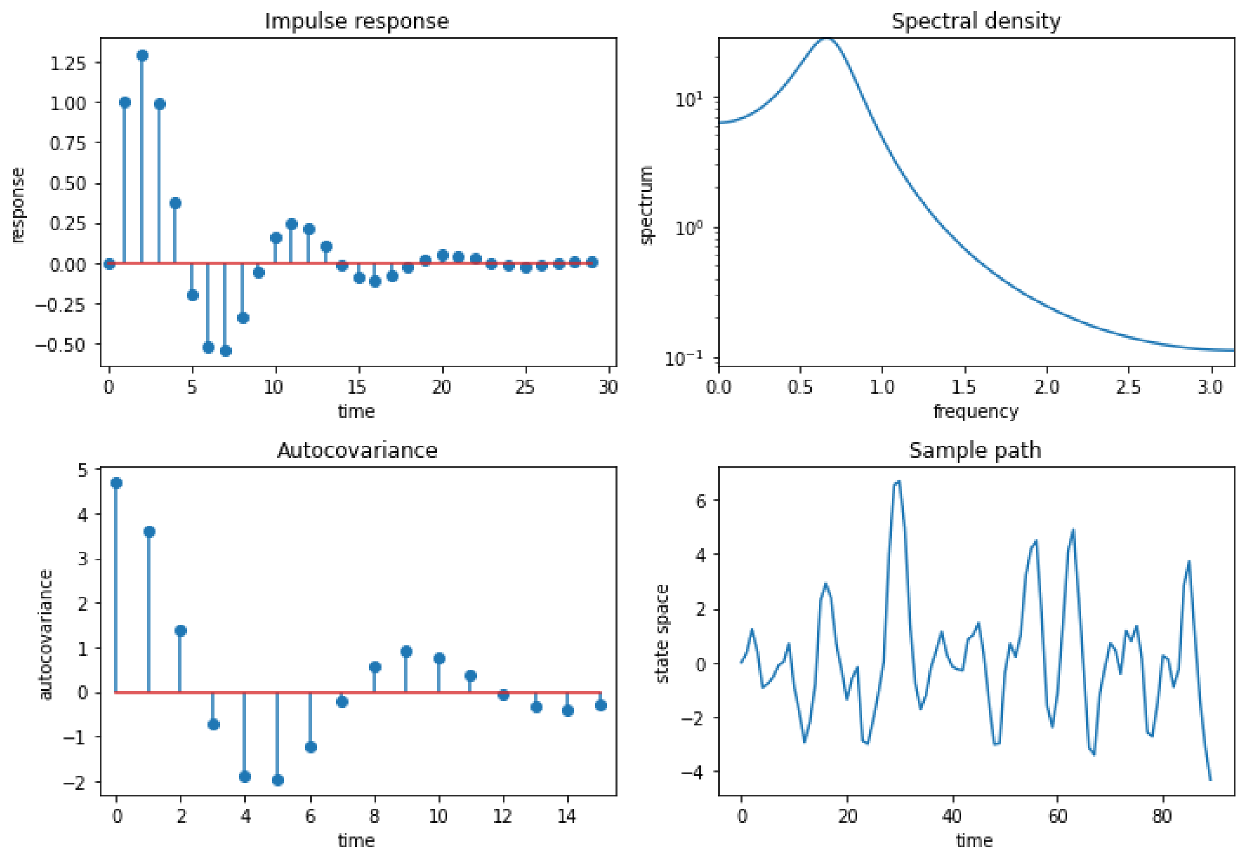
- the variance equals $1 = \frac{1}{2\pi} \int_{-\pi}^{\pi} 1 d\omega$ as it should.
- the covariogram and impulse response look as they should.
- it is actually challenging to visualize a time series realization of white noise – a sequence of surprises – but this too looks pretty good.

To get some more examples, as our laboratory we'll replicate quartets of graphs that [LS18] use to teach "how to read spectral densities".

Ljunqvist and Sargent's first model is $X_t = 1.3X_{t-1} - .7X_{t-2} + \epsilon_t$

```
ϕ = 1.3, -.7
θ = 0.0
arma = qe.ARMA(ϕ, θ)
quad_plot(arma)
```

```
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/numpy/core/_asarray.
↪py:83: ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
<ipython-input-7-4995e3f7b93d>:15: UserWarning: Attempted to set non-positive bottom␣
↪ylim on a log-scaled axis.
Invalid limit will be ignored.
  ax.set(xlim=(0, np.pi), ylim=(0, np.max(spect)),
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/matplotlib/
↪transforms.py:948: ComplexWarning: Casting complex values to real discards the␣
↪imaginary part
  self._points[:, 1] = interval
```



Ljungqvist and Sargent's second model is $X_t = .9X_{t-1} + \epsilon_t$

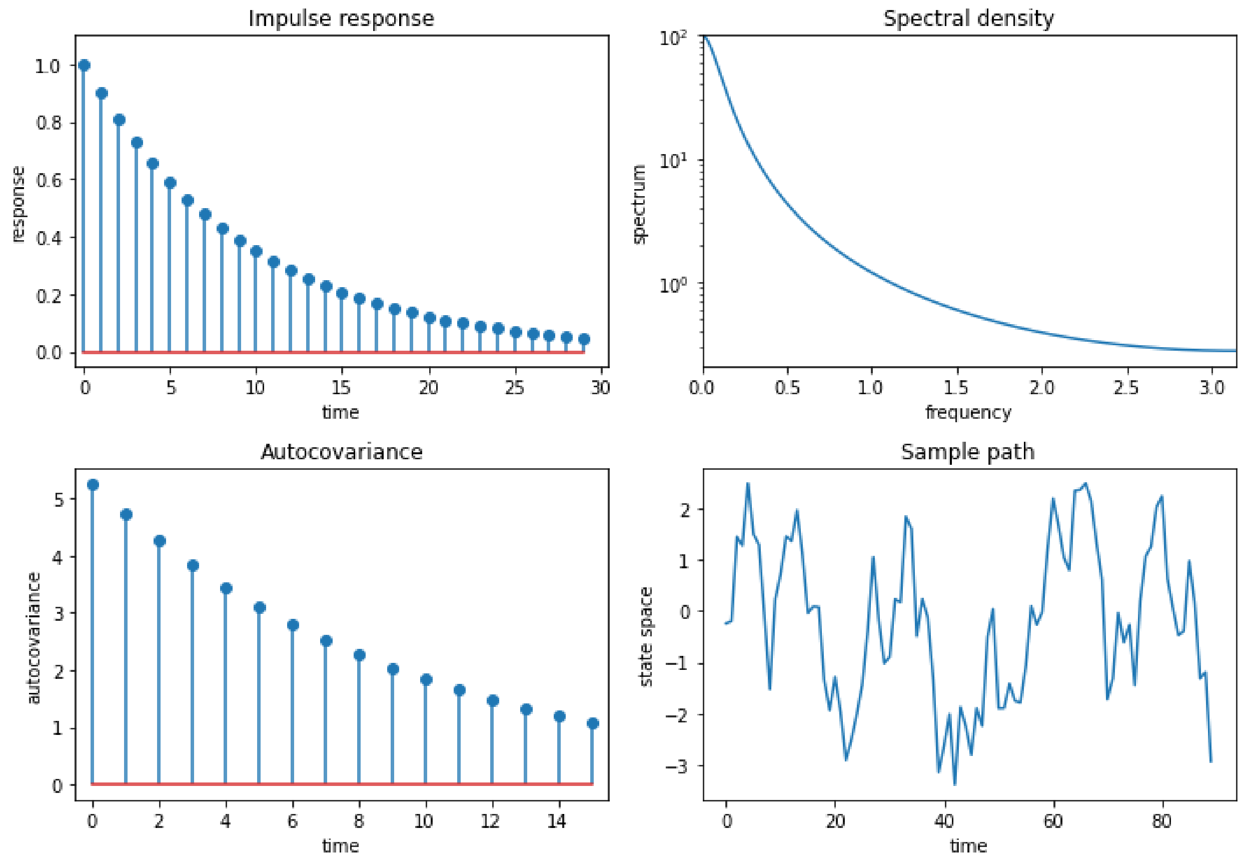
```
ϕ = 0.9
θ = -0.0
arma = qe.ARMA(ϕ, θ)
quad_plot(arma)
```

```
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/numpy/core/_asarray.
↪py:83: ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
<ipython-input-7-4995e3f7b93d>:15: UserWarning: Attempted to set non-positive bottom␣
↪ylim on a log-scaled axis.
Invalid limit will be ignored.
  ax.set(xlim=(0, np.pi), ylim=(0, np.max(spect)),
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/matplotlib/
↪transforms.py:948: ComplexWarning: Casting complex values to real discards the␣
↪imaginary part
```

(continues on next page)

(continued from previous page)

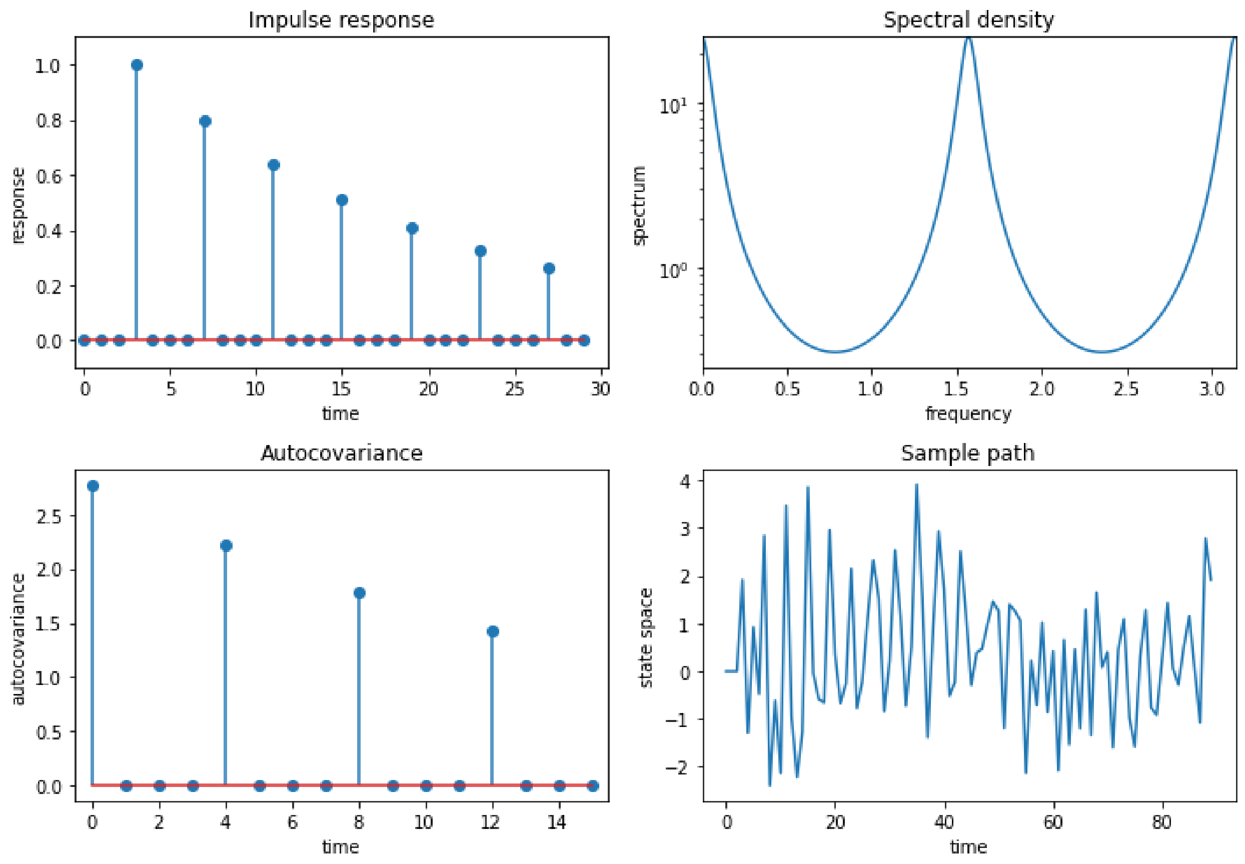
```
  self._points[:, 1] = interval
```



Ljungqvist and Sargent's third model is $X_t = .8X_{t-4} + \epsilon_t$

```
ϕ = 0., 0., 0., .8
θ = -0.0
arma = qe.ARMA(ϕ, θ)
quad_plot(arma)
```

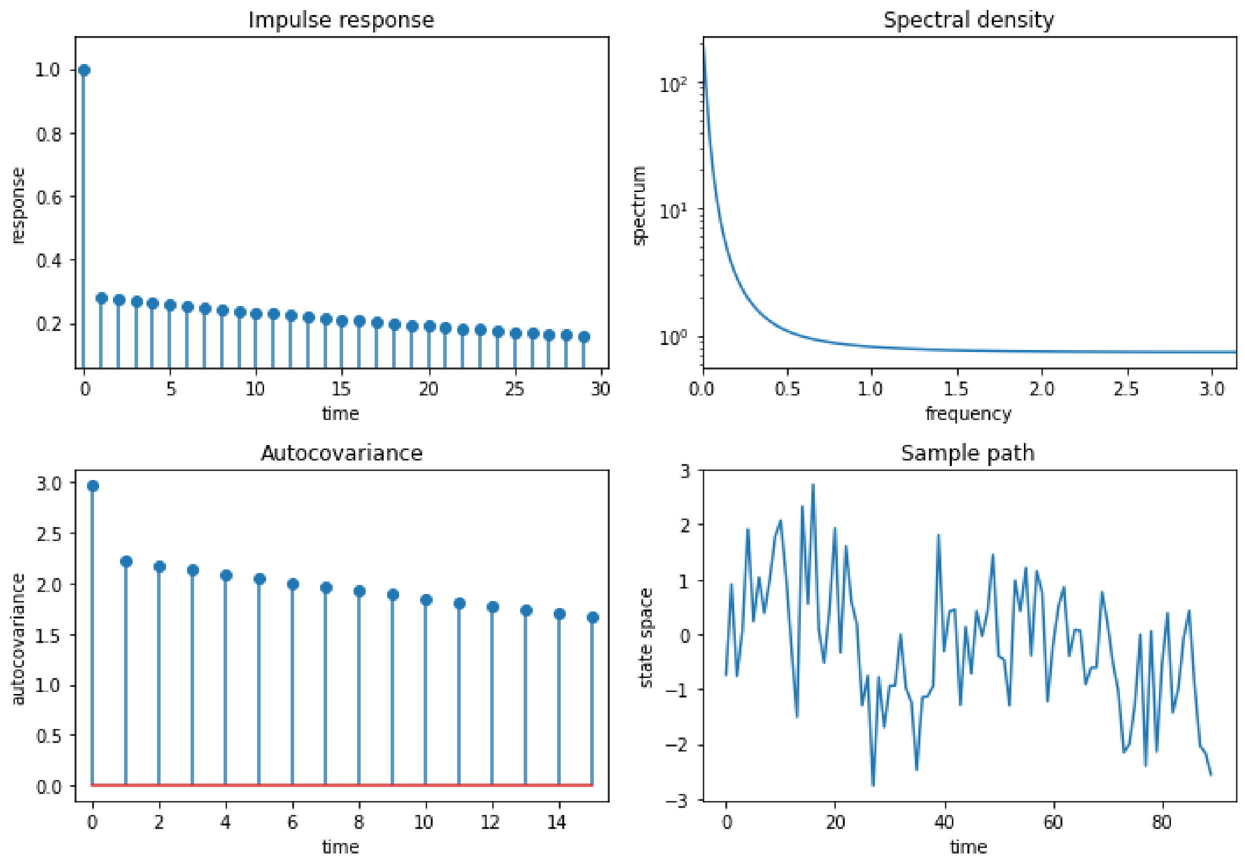
```
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/numpy/core/_asarray.
↪py:83: ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
<ipython-input-7-4995e3f7b93d>:15: UserWarning: Attempted to set non-positive bottom␣
↪ylim on a log-scaled axis.
Invalid limit will be ignored.
  ax.set(xlim=(0, np.pi), ylim=(0, np.max(spect)),
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/matplotlib/
↪transforms.py:948: ComplexWarning: Casting complex values to real discards the␣
↪imaginary part
  self._points[:, 1] = interval
```



Ljungqvist and Sargent's fourth model is $X_t = .98X_{t-1} + \epsilon_t - .7\epsilon_{t-1}$

```
ϕ = .98
θ = -0.7
arma = qe.ARMA(ϕ, θ)
quad_plot(arma)
```

```
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/numpy/core/_asarray.
↪py:83: ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
<ipython-input-7-4995e3f7b93d>:15: UserWarning: Attempted to set non-positive bottom␣
↪ylim on a log-scaled axis.
Invalid limit will be ignored.
  ax.set(xlim=(0, np.pi), ylim=(0, np.max(spect)),
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/matplotlib/
↪transforms.py:948: ComplexWarning: Casting complex values to real discards the␣
↪imaginary part
  self._points[:, 1] = interval
```



27.4.2 Explanation

The call

```
arma = ARMA(ϕ, θ, σ)
```

creates an instance `arma` that represents the ARMA(p, q) model

$$X_t = \phi_1 X_{t-1} + ... + \phi_p X_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + ... + \theta_q \epsilon_{t-q}$$

If `ϕ` and `θ` are arrays or sequences, then the interpretation will be

- `ϕ` holds the vector of parameters $(\phi_1, \phi_2, ..., \phi_p)$.
- `θ` holds the vector of parameters $(\theta_1, \theta_2, ..., \theta_q)$.

The parameter σ is always a scalar, the standard deviation of the white noise.

We also permit `ϕ` and `θ` to be scalars, in which case the model will be interpreted as

$$X_t = \phi X_{t-1} + \epsilon_t + \theta \epsilon_{t-1}$$

The two numerical packages most useful for working with ARMA models are `scipy.signal` and `numpy.fft`.

The package `scipy.signal` expects the parameters to be passed into its functions in a manner consistent with the alternative ARMA notation (8).

For example, the impulse response sequence $\{\psi_t\}$ discussed above can be obtained using `scipy.signal.dimpulse`, and the function call should be of the form

```
times, ψ = dimpulse((ma_poly, ar_poly, 1), n=impulse_length)
```

where `ma_poly` and `ar_poly` correspond to the polynomials in (7) — that is,

- `ma_poly` is the vector $(1, \theta_1, \theta_2, \dots, \theta_q)$
- `ar_poly` is the vector $(1, -\phi_1, -\phi_2, \dots, -\phi_p)$

To this end, we also maintain the arrays `ma_poly` and `ar_poly` as instance data, with their values computed automatically from the values of `phi` and `theta` supplied by the user.

If the user decides to change the value of either `theta` or `phi` ex-post by assignments such as `arma.phi = (0.5, 0.2)` or `arma.theta = (0, -0.1)`.

then `ma_poly` and `ar_poly` should update automatically to reflect these new parameters.

This is achieved in our implementation by using descriptors.

27.4.3 Computing the Autocovariance Function

As discussed above, for ARMA processes the spectral density has a *simple representation* that is relatively easy to calculate.

Given this fact, the easiest way to obtain the autocovariance function is to recover it from the spectral density via the inverse Fourier transform.

Here we use NumPy's Fourier transform package np.fft, which wraps a standard Fortran-based package called FFTPACK.

A look at the np.fft documentation shows that the inverse transform np.fft.ifft takes a given sequence $A_0, A_1, \dots, A_{n-1}$ and returns the sequence $a_0, a_1, \dots, a_{n-1}$ defined by

$$a_k = \frac{1}{n} \sum_{t=0}^{n-1} A_t e^{ik2\pi t/n}$$

Thus, if we set $A_t = f(\omega_t)$, where f is the spectral density and $\omega_t := 2\pi t/n$, then

$$a_k = \frac{1}{n} \sum_{t=0}^{n-1} f(\omega_t) e^{i\omega_t k} = \frac{1}{2\pi} \frac{2\pi}{n} \sum_{t=0}^{n-1} f(\omega_t) e^{i\omega_t k}, \qquad \omega_t := 2\pi t/n$$

For n sufficiently large, we then have

$$a_k \approx \frac{1}{2\pi} \int_0^{2\pi} f(\omega) e^{i\omega k} d\omega = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(\omega) e^{i\omega k} d\omega$$

(You can check the last equality)

In view of (14), we have now shown that, for n sufficiently large, $a_k \approx \gamma(k)$ — which is exactly what we want to compute.

CHAPTER

TWENTYEIGHT

ESTIMATION OF SPECTRA

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

28.1 Overview

In a *previous lecture*, we covered some fundamental properties of covariance stationary linear stochastic processes.

One objective for that lecture was to introduce spectral densities — a standard and very useful technique for analyzing such processes.

In this lecture, we turn to the problem of estimating spectral densities and other related quantities from data.

Estimates of the spectral density are computed using what is known as a periodogram — which in turn is computed via the famous fast Fourier transform.

Once the basic technique has been explained, we will apply it to the analysis of several key macroeconomic time series.

For supplementary reading, see [Sar87] or [CC08].

Let's start with some standard imports:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from quantecon import ARMA, periodogram, ar_periodogram
```

28.2 Periodograms

Recall that the spectral density f of a covariance stationary process with autocorrelation function γ can be written

$$f(\omega) = \gamma(0) + 2 \sum_{k \geq 1} \gamma(k) \cos(\omega k), \qquad \omega \in \mathbb{R}$$

Now consider the problem of estimating the spectral density of a given time series, when γ is unknown.

In particular, let $X_0, \ldots, X_{n-1}$ be n consecutive observations of a single time series that is assumed to be covariance stationary.

The most common estimator of the spectral density of this process is the *periodogram* of $X_0, \ldots, X_{n-1}$, which is defined as

$$I(\omega) := \frac{1}{n} \left| \sum_{t=0}^{n-1} X_t e^{it\omega} \right|^2, \qquad \omega \in \mathbb{R} \tag{1}$$

(Recall that $|z|$ denotes the modulus of complex number z)

Alternatively, $I(\omega)$ can be expressed as

$$I(\omega) = \frac{1}{n} \left\{ \left[\sum_{t=0}^{n-1} X_t \cos(\omega t) \right]^2 + \left[\sum_{t=0}^{n-1} X_t \sin(\omega t) \right]^2 \right\}$$

It is straightforward to show that the function I is even and 2π-periodic (i.e., $I(\omega) = I(-\omega)$ and $I(\omega + 2\pi) = I(\omega)$ for all $\omega \in \mathbb{R}$).

From these two results, you will be able to verify that the values of I on $[0, \pi]$ determine the values of I on all of $\mathbb{R}$.

The next section helps to explain the connection between the periodogram and the spectral density.

28.2.1 Interpretation

To interpret the periodogram, it is convenient to focus on its values at the *Fourier frequencies*

$$\omega_j := \frac{2\pi j}{n}, \quad j = 0, \ldots, n-1$$

In what sense is $I(\omega_j)$ an estimate of $f(\omega_j)$?

The answer is straightforward, although it does involve some algebra.

With a bit of effort, one can show that for any integer $j > 0$,

$$\sum_{t=0}^{n-1} e^{it\omega_j} = \sum_{t=0}^{n-1} \exp\left\{ i 2\pi j \frac{t}{n} \right\} = 0$$

Letting $\bar{X}$ denote the sample mean $n^{-1} \sum_{t=0}^{n-1} X_t$, we then have

$$nI(\omega_j) = \left| \sum_{t=0}^{n-1} (X_t - \bar{X}) e^{it\omega_j} \right|^2 = \sum_{t=0}^{n-1} (X_t - \bar{X}) e^{it\omega_j} \sum_{r=0}^{n-1} (X_r - \bar{X}) e^{-ir\omega_j}$$

By carefully working through the sums, one can transform this to

$$nI(\omega_j) = \sum_{t=0}^{n-1} (X_t - \bar{X})^2 + 2 \sum_{k=1}^{n-1} \sum_{t=k}^{n-1} (X_t - \bar{X})(X_{t-k} - \bar{X}) \cos(\omega_j k)$$

Now let

$$\hat{\gamma}(k) := \frac{1}{n} \sum_{t=k}^{n-1} (X_t - \bar{X})(X_{t-k} - \bar{X}), \qquad k = 0, 1, \ldots, n-1$$

This is the sample autocovariance function, the natural "plug-in estimator" of the *autocovariance function* γ.

("Plug-in estimator" is an informal term for an estimator found by replacing expectations with sample means)

With this notation, we can now write

$$I(\omega_j) = \hat{\gamma}(0) + 2 \sum_{k=1}^{n-1} \hat{\gamma}(k) \cos(\omega_j k)$$

Recalling our expression for f given *above*, we see that $I(\omega_j)$ is just a sample analog of $f(\omega_j)$.

28.2.2 Calculation

Let's now consider how to compute the periodogram as defined in (1).

There are already functions available that will do this for us — an example is `statsmodels.tsa.stattools.periodogram` in the `statsmodels` package.

However, it is very simple to replicate their results, and this will give us a platform to make useful extensions.

The most common way to calculate the periodogram is via the discrete Fourier transform, which in turn is implemented through the fast Fourier transform algorithm.

In general, given a sequence $a_0, \ldots, a_{n-1}$, the discrete Fourier transform computes the sequence

$$A_j := \sum_{t=0}^{n-1} a_t \exp\left\{i 2\pi \frac{tj}{n}\right\}, \qquad j = 0, \ldots, n-1$$

With `numpy.fft.fft` imported as `fft` and $a_0, \ldots, a_{n-1}$ stored in NumPy array `a`, the function call `fft(a)` returns the values $A_0, \ldots, A_{n-1}$ as a NumPy array.

It follows that when the data $X_0, \ldots, X_{n-1}$ are stored in array `X`, the values $I(\omega_j)$ at the Fourier frequencies, which are given by

$$\frac{1}{n} \left| \sum_{t=0}^{n-1} X_t \exp\left\{i 2\pi \frac{tj}{n}\right\} \right|^2, \qquad j = 0, \ldots, n-1$$

can be computed by `np.abs(fft(X))**2 / len(X)`.

Note: The NumPy function `abs` acts elementwise, and correctly handles complex numbers (by computing their modulus, which is exactly what we need).

A function called `periodogram` that puts all this together can be found here.

Let's generate some data for this function using the `ARMA` class from QuantEcon.py (see the *lecture on linear processes* for more details).

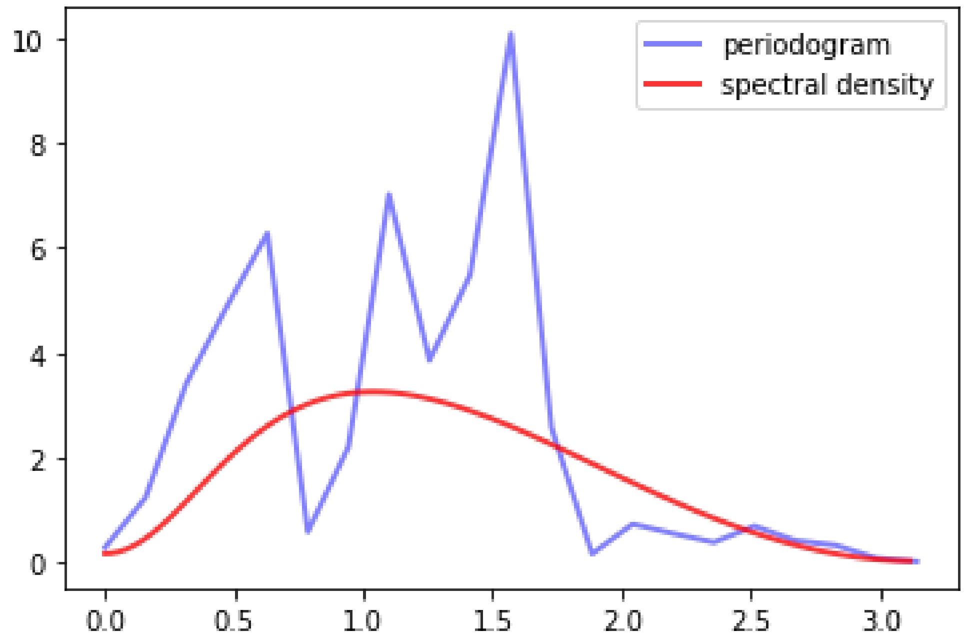
Here's a code snippet that, once the preceding code has been run, generates data from the process

$$X_t = 0.5 X_{t-1} + \epsilon_t - 0.8 \epsilon_{t-2} \tag{2}$$

where $\{\epsilon_t\}$ is white noise with unit variance, and compares the periodogram to the actual spectral density

```
n = 40                              # Data size
ϕ, θ = 0.5, (0, -0.8)               # AR and MA parameters
lp = ARMA(ϕ, θ)
X = lp.simulation(ts_length=n)

fig, ax = plt.subplots()
x, y = periodogram(X)
ax.plot(x, y, 'b-', lw=2, alpha=0.5, label='periodogram')
x_sd, y_sd = lp.spectral_density(two_pi=False, res=120)
ax.plot(x_sd, y_sd, 'r-', lw=2, alpha=0.8, label='spectral density')
ax.legend()
plt.show()
```

```
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/numpy/core/_asarray.
↪py:83: ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
```

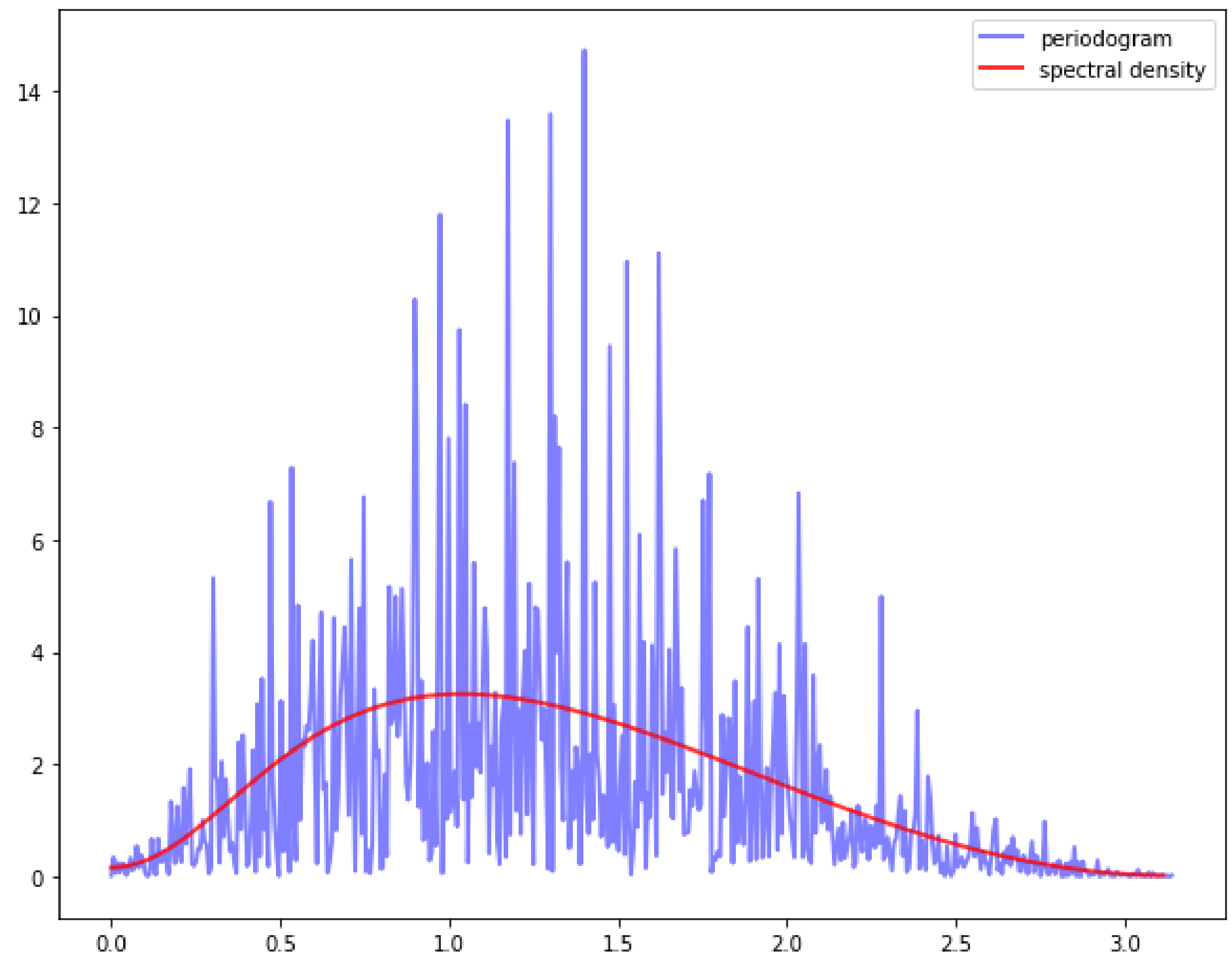


This estimate looks rather disappointing, but the data size is only 40, so perhaps it's not surprising that the estimate is poor.

However, if we try again with `n = 1200` the outcome is not much better

The periodogram is far too irregular relative to the underlying spectral density.

This brings us to our next topic.


periodogram
spectral density
14
12
10
8
6
4
2
0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

28.3 Smoothing

There are two related issues here.

One is that, given the way the fast Fourier transform is implemented, the number of points ω at which $I(\omega)$ is estimated increases in line with the amount of data.

In other words, although we have more data, we are also using it to estimate more values.

A second issue is that densities of all types are fundamentally hard to estimate without parametric assumptions.

Typically, nonparametric estimation of densities requires some degree of smoothing.

The standard way that smoothing is applied to periodograms is by taking local averages.

In other words, the value $I(\omega_j)$ is replaced with a weighted average of the adjacent values

$$I(\omega_{j-p}), I(\omega_{j-p+1}), \dots, I(\omega_j), \dots, I(\omega_{j+p})$$

This weighted average can be written as

$$I_S(\omega_j) := \sum_{\ell=-p}^{p} w(\ell) I(\omega_{j+\ell}) \tag{3}$$

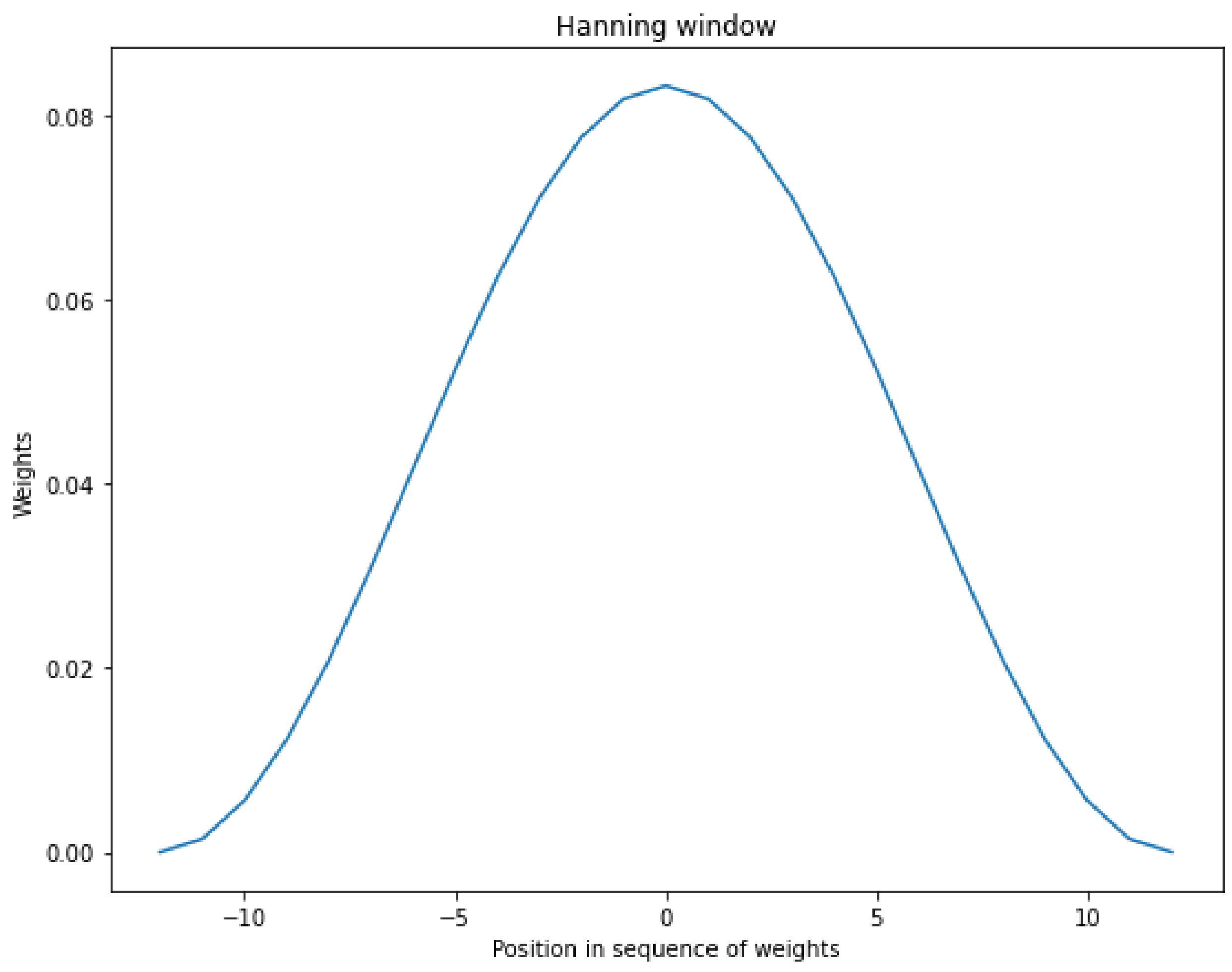
where the weights $w(-p), \dots, w(p)$ are a sequence of $2p + 1$ nonnegative values summing to one.

In general, larger values of p indicate more smoothing — more on this below.

The next figure shows the kind of sequence typically used.

Note the smaller weights towards the edges and larger weights in the center, so that more distant values from $I(\omega_j)$ have less weight than closer ones in the sum (3).

```python
def hanning_window(M):
    w = [0.5 - 0.5 * np.cos(2 * np.pi * n/(M-1)) for n in range(M)]
    return w

window = hanning_window(25) / np.abs(sum(hanning_window(25)))
x = np.linspace(-12, 12, 25)
fig, ax = plt.subplots(figsize=(9, 7))
ax.plot(x, window)
ax.set_title("Hanning window")
ax.set_ylabel("Weights")
ax.set_xlabel("Position in sequence of weights")
plt.show()
```



28.3.1 Estimation with Smoothing

Our next step is to provide code that will not only estimate the periodogram but also provide smoothing as required.

Such functions have been written in estspec.py and are available once you've installed QuantEcon.py.

The GitHub listing displays three functions, `smooth()`, `periodogram()`, `ar_periodogram()`. We will discuss the first two here and the third one *below*.

The `periodogram()` function returns a periodogram, optionally smoothed via the `smooth()` function.

Regarding the `smooth()` function, since smoothing adds a nontrivial amount of computation, we have applied a fairly terse array-centric method based around `np.convolve`.

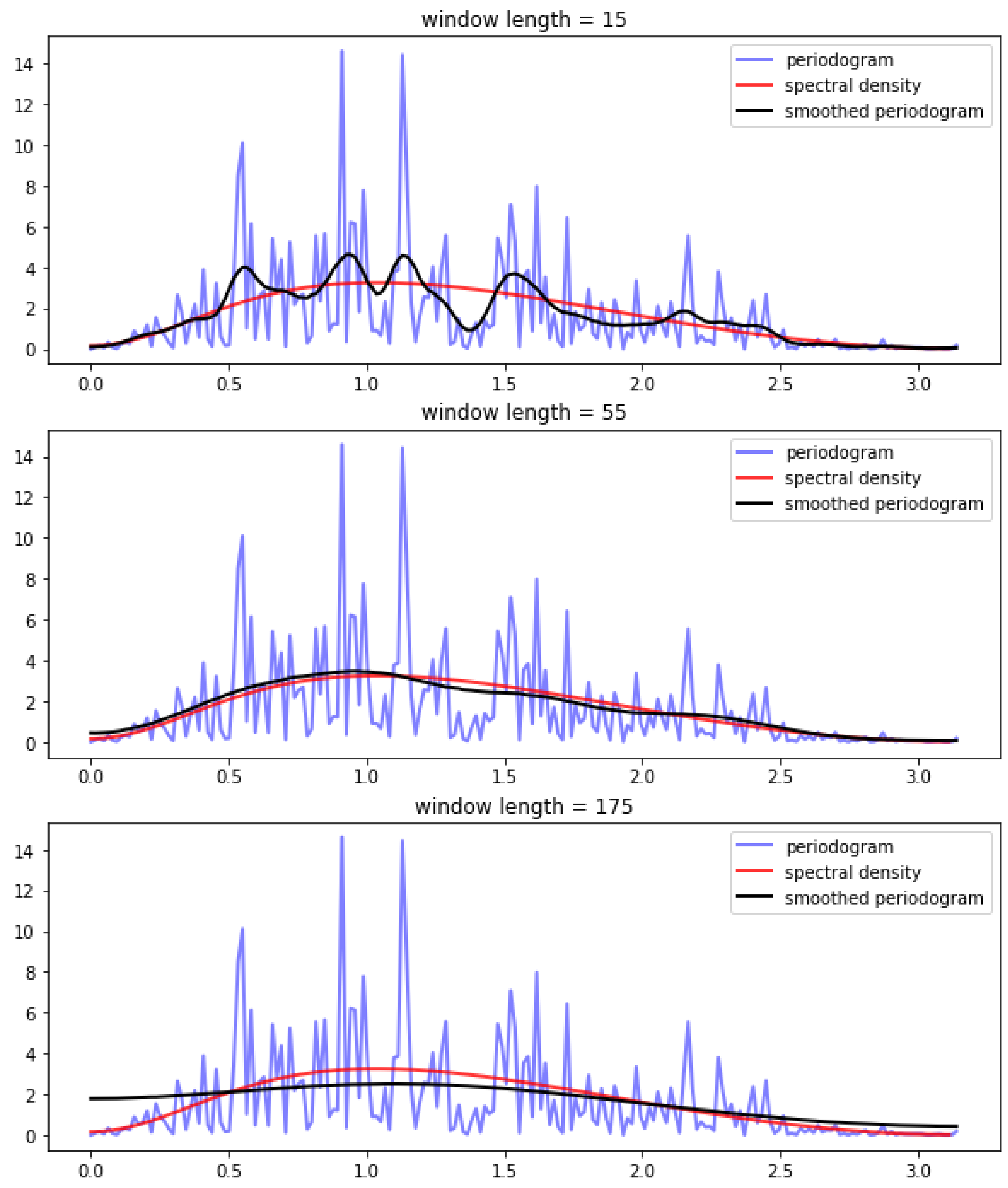
Readers are left either to explore or simply to use this code according to their interests.

The next three figures each show smoothed and unsmoothed periodograms, as well as the population or "true" spectral density.

(The model is the same as before — see equation (2) — and there are 400 observations)

From the top figure to bottom, the window length is varied from small to large.

In looking at the figure, we can see that for this model and data size, the window length chosen in the middle figure provides the best fit.


window length = 15
window length = 55
window length = 175
periodogram
spectral density
smoothed periodogram
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0
2
4
6
8
10
12
14

Relative to this value, the first window length provides insufficient smoothing, while the third gives too much smoothing.

Of course in real estimation problems, the true spectral density is not visible and the choice of appropriate smoothing will have to be made based on judgement/priors or some other theory.

28.3.2 Pre-Filtering and Smoothing

In the code listing, we showed three functions from the file `estspec.py`.

The third function in the file (`ar_periodogram()`) adds a pre-processing step to periodogram smoothing.

First, we describe the basic idea, and after that we give the code.

The essential idea is to

1. Transform the data in order to make estimation of the spectral density more efficient.
2. Compute the periodogram associated with the transformed data.
3. Reverse the effect of the transformation on the periodogram, so that it now estimates the spectral density of the original process.

Step 1 is called *pre-filtering* or *pre-whitening*, while step 3 is called *recoloring*.

The first step is called pre-whitening because the transformation is usually designed to turn the data into something closer to white noise.

Why would this be desirable in terms of spectral density estimation?

The reason is that we are smoothing our estimated periodogram based on estimated values at nearby points — recall (3).

The underlying assumption that makes this a good idea is that the true spectral density is relatively regular — the value of $I(\omega)$ is close to that of $I(\omega')$ when ω is close to ω'.

This will not be true in all cases, but it is certainly true for white noise.

For white noise, I is as regular as possible — *it is a constant function*.

In this case, values of $I(\omega')$ at points ω' near to ω provided the maximum possible amount of information about the value $I(\omega)$.

Another way to put this is that if I is relatively constant, then we can use a large amount of smoothing without introducing too much bias.

28.3.3 The AR(1) Setting

Let's examine this idea more carefully in a particular setting — where the data are assumed to be generated by an AR(1) process.

(More general ARMA settings can be handled using similar techniques to those described below)

Suppose in particular that $\{X_t\}$ is covariance stationary and AR(1), with

$$X_{t+1} = \mu + \phi X_t + \epsilon_{t+1} \tag{4}$$

where μ and $\phi \in (-1, 1)$ are unknown parameters and $\{\epsilon_t\}$ is white noise.

It follows that if we regress X_{t+1} on X_t and an intercept, the residuals will approximate white noise.

Let

- g be the spectral density of $\{\epsilon_t\}$ — a constant function, as discussed above

- I_0 be the periodogram estimated from the residuals — an estimate of g
- f be the spectral density of $\{X_t\}$ — the object we are trying to estimate

In view of *an earlier result* we obtained while discussing ARMA processes, f and g are related by

$$f(\omega) = \left| \frac{1}{1 - \phi e^{i\omega}} \right|^2 g(\omega) \tag{5}$$

This suggests that the recoloring step, which constructs an estimate I of f from I_0, should set

$$I(\omega) = \left| \frac{1}{1 - \hat{\phi} e^{i\omega}} \right|^2 I_0(\omega)$$

where $\hat{\phi}$ is the OLS estimate of ϕ.

The code for `ar_periodogram()` — the third function in `estspec.py` — does exactly this. (See the code here).

The next figure shows realizations of the two kinds of smoothed periodograms

1. "standard smoothed periodogram", the ordinary smoothed periodogram, and
2. "AR smoothed periodogram", the pre-whitened and recolored one generated by `ar_periodogram()`

The periodograms are calculated from time series drawn from (4) with $\mu = 0$ and $\phi = -0.9$.

Each time series is of length 150.

The difference between the three subfigures is just randomness — each one uses a different draw of the time series.

In all cases, periodograms are fit with the "hamming" window and window length of 65.

Overall, the fit of the AR smoothed periodogram is much better, in the sense of being closer to the true spectral density.

28.4 Exercises

28.4.1 Exercise 1

Replicate *this figure* (modulo randomness).

The model is as in equation (2) and there are 400 observations.

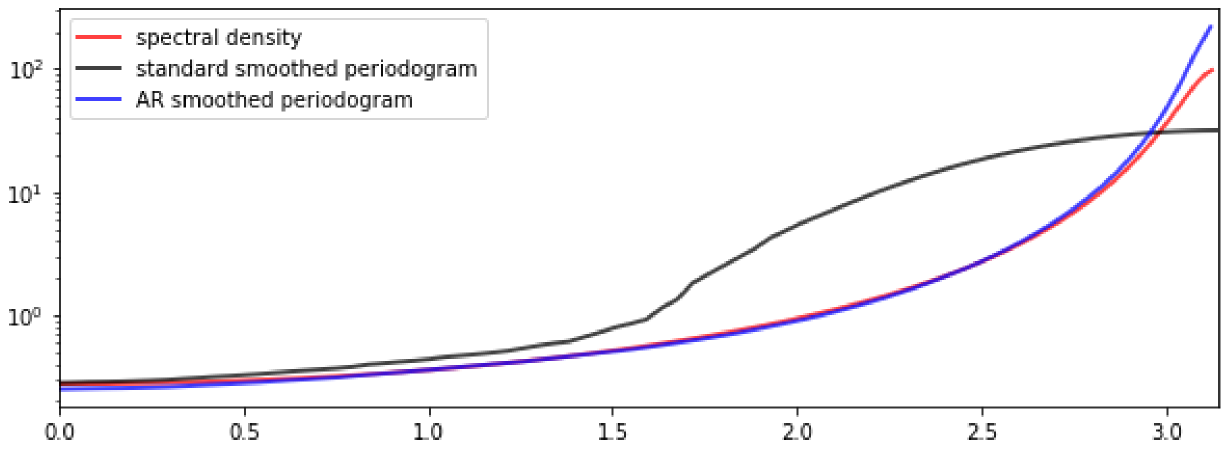
For the smoothed periodogram, the window type is "hamming".

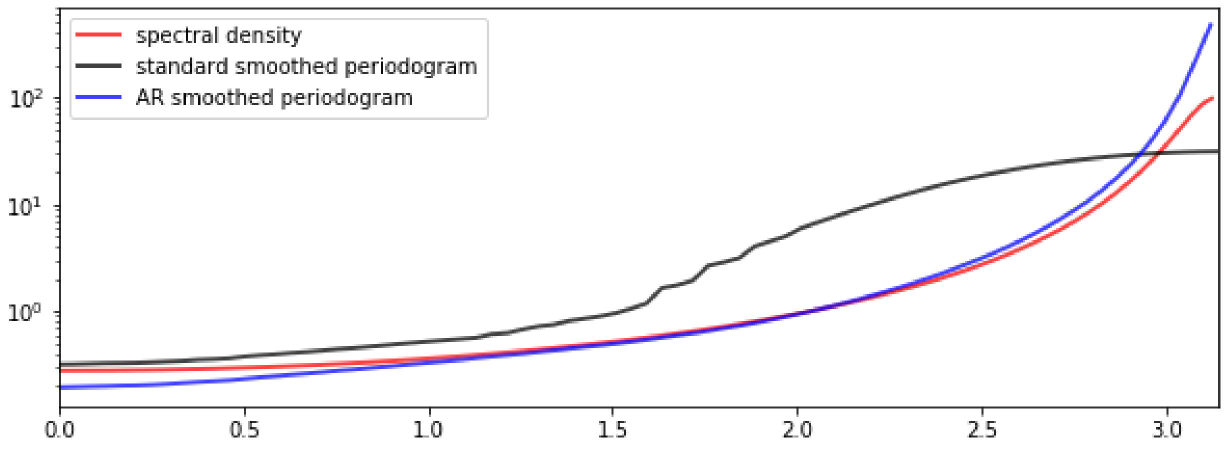
28.4.2 Exercise 2

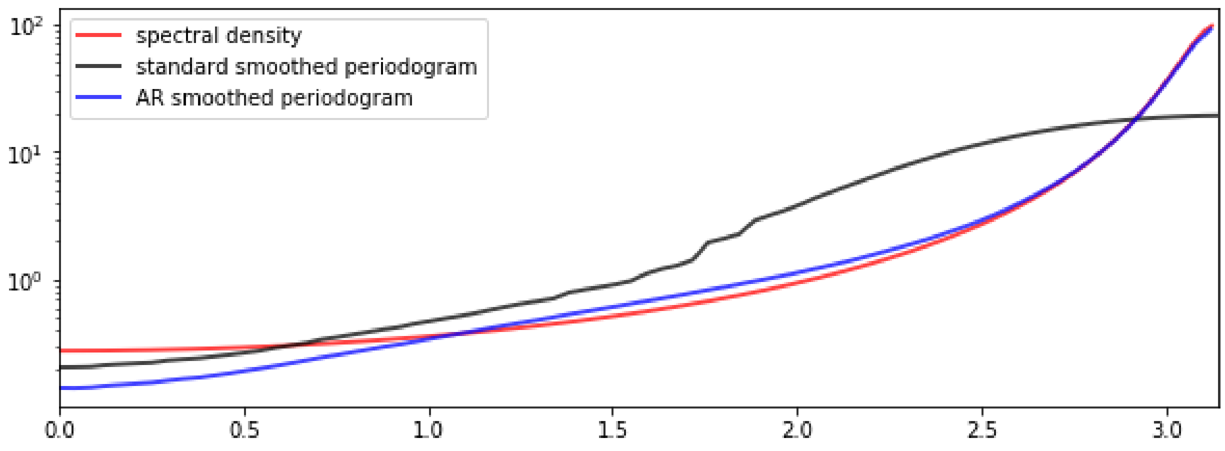
Replicate *this figure* (modulo randomness).

The model is as in equation (4), with $\mu = 0$, $\phi = -0.9$ and 150 observations in each time series.

All periodograms are fit with the "hamming" window and window length of 65.


spectral density
standard smoothed periodogram
AR smoothed periodogram
10^2
10^1
10^0
0.0
0.5
1.0
1.5
2.0
2.5
3.0


spectral density
standard smoothed periodogram
AR smoothed periodogram
10^2
10^1
10^0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

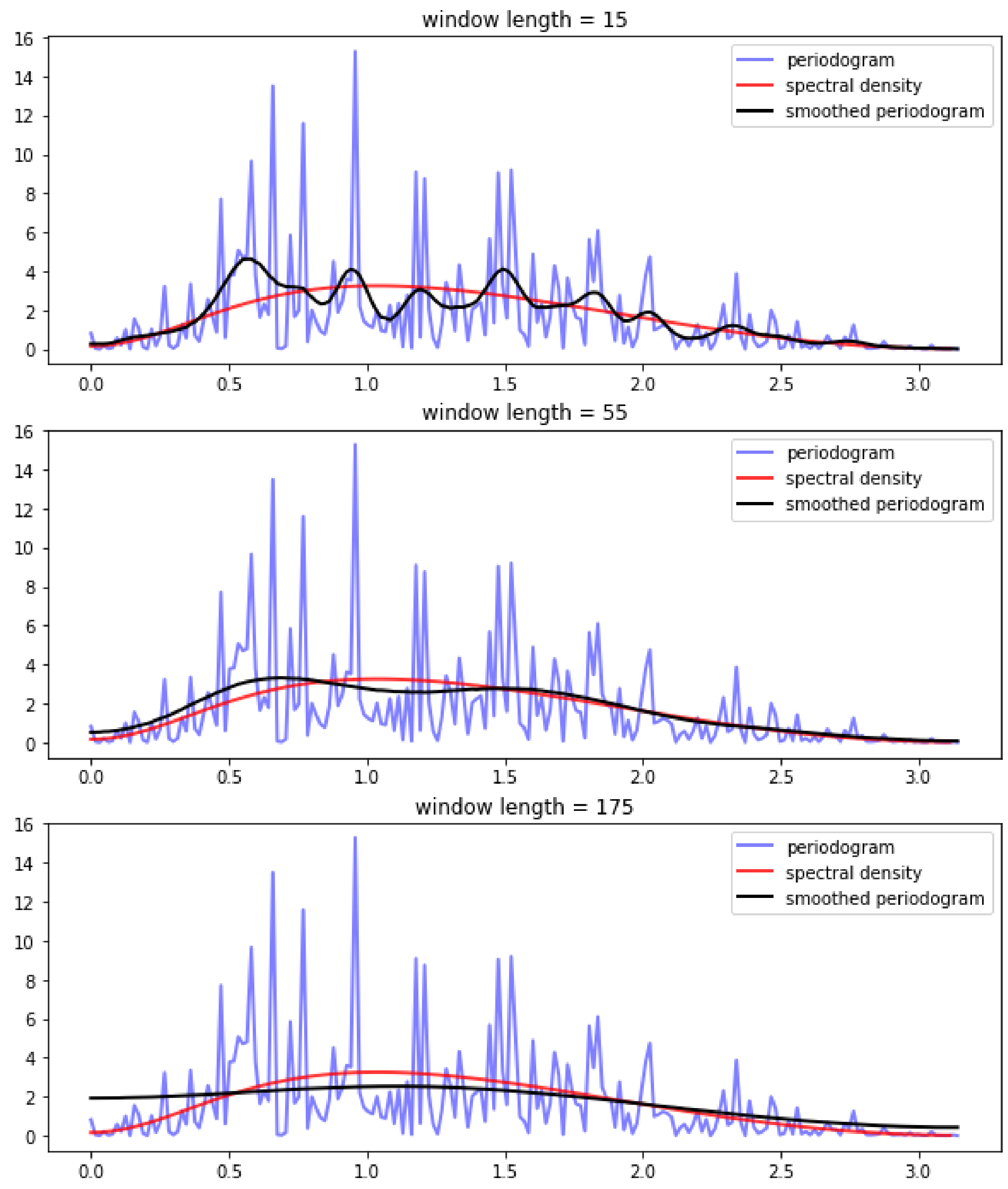

spectral density
standard smoothed periodogram
AR smoothed periodogram
10^2
10^1
10^0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

28.5 Solutions

28.5.1 Exercise 1

```
## Data
n = 400
ϕ = 0.5
θ = 0, -0.8
lp = ARMA(ϕ, θ)
X = lp.simulation(ts_length=n)

fig, ax = plt.subplots(3, 1, figsize=(10, 12))

for i, wl in enumerate((15, 55, 175)):  # Window lengths

    x, y = periodogram(X)
    ax[i].plot(x, y, 'b-', lw=2, alpha=0.5, label='periodogram')

    x_sd, y_sd = lp.spectral_density(two_pi=False, res=120)
    ax[i].plot(x_sd, y_sd, 'r-', lw=2, alpha=0.8, label='spectral density')

    x, y_smoothed = periodogram(X, window='hamming', window_len=wl)
    ax[i].plot(x, y_smoothed, 'k-', lw=2, label='smoothed periodogram')

    ax[i].legend()
    ax[i].set_title(f'window length = {wl}')
plt.show()
```

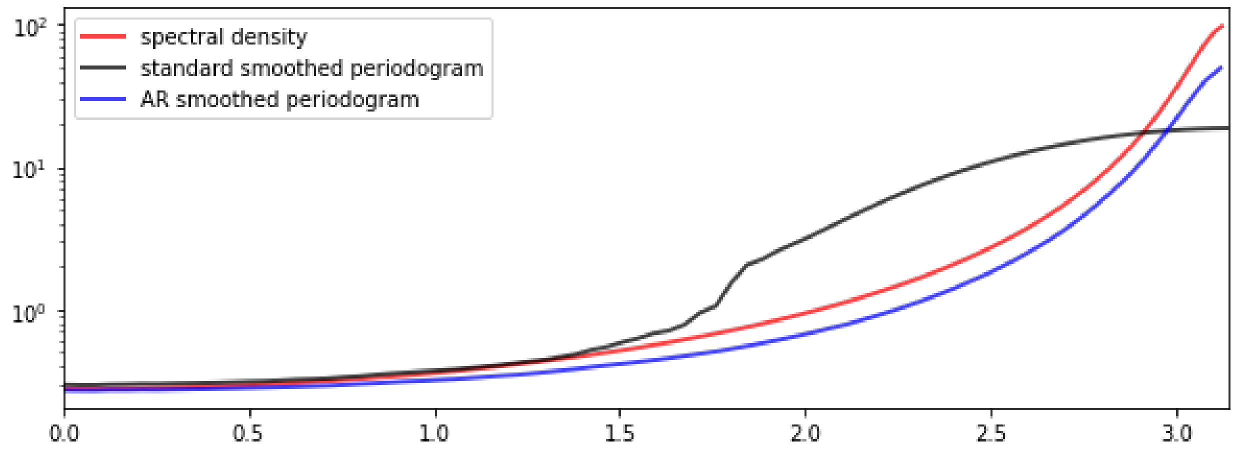
```
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/numpy/core/_asarray.
↪py:83: ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/numpy/core/_asarray.
↪py:83: ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/numpy/core/_asarray.
↪py:83: ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
```

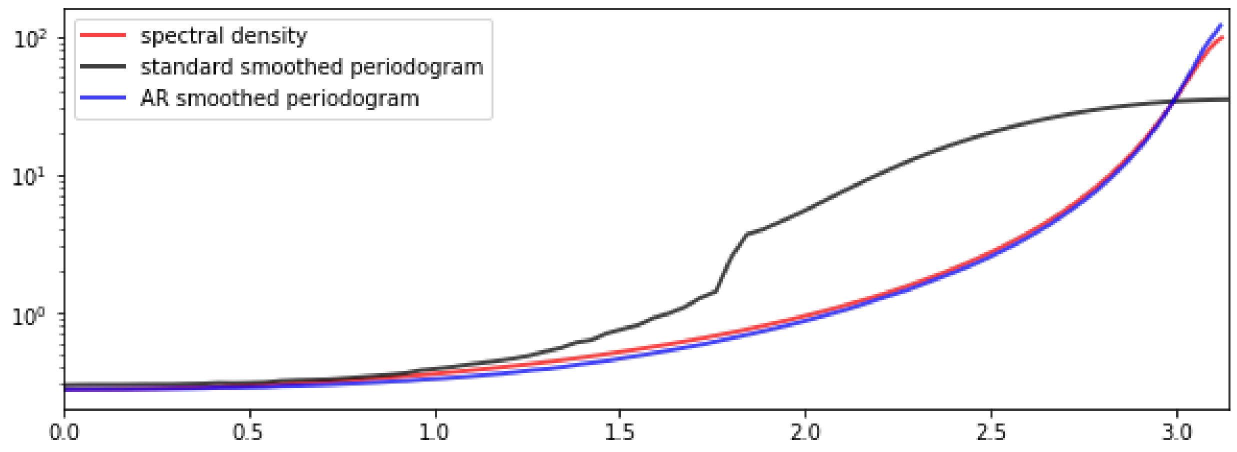

window length = 15
periodogram
spectral density
smoothed periodogram
window length = 55
periodogram
spectral density
smoothed periodogram
window length = 175
periodogram
spectral density
smoothed periodogram
0
2
4
6
8
10
12
14
16
0.0
0.5
1.0
1.5
2.0
2.5
3.0

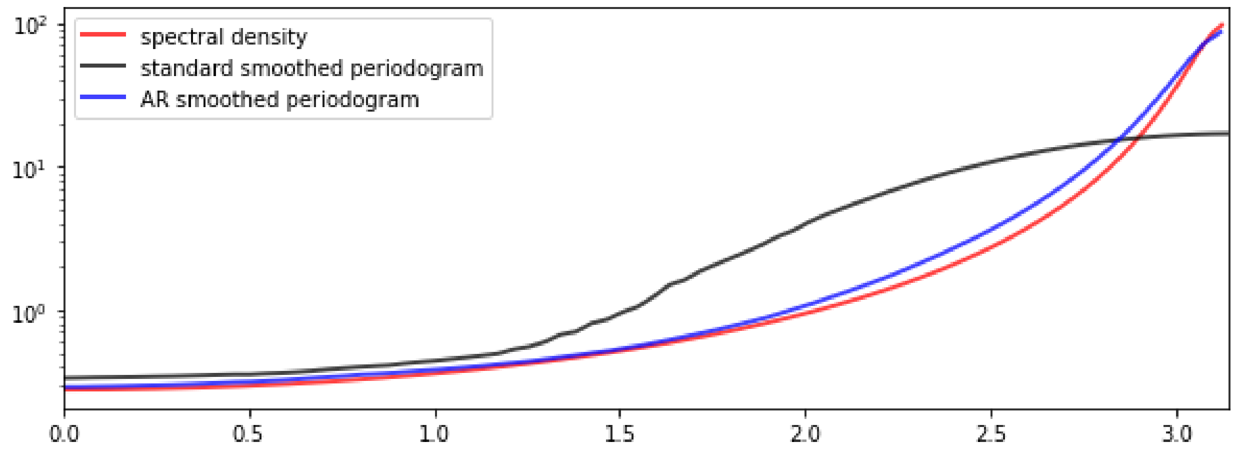
28.5.2 Exercise 2

```
lp = ARMA(-0.9)
wl = 65


fig, ax = plt.subplots(3, 1, figsize=(10,12))

for i in range(3):
    X = lp.simulation(ts_length=150)
    ax[i].set_xlim(0, np.pi)

    x_sd, y_sd = lp.spectral_density(two_pi=False, res=180)
    ax[i].semilogy(x_sd, y_sd, 'r-', lw=2, alpha=0.75,
        label='spectral density')

    x, y_smoothed = periodogram(X, window='hamming', window_len=wl)
    ax[i].semilogy(x, y_smoothed, 'k-', lw=2, alpha=0.75,
        label='standard smoothed periodogram')

    x, y_ar = ar_periodogram(X, window='hamming', window_len=wl)
    ax[i].semilogy(x, y_ar, 'b-', lw=2, alpha=0.75,
        label='AR smoothed periodogram')

    ax[i].legend(loc='upper left')
plt.show()
```

```
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/numpy/core/_asarray.
↪py:83: ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/numpy/core/_asarray.
↪py:83: ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/numpy/core/_asarray.
↪py:83: ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
```


spectral density
standard smoothed periodogram
AR smoothed periodogram
10^2
10^1
10^0
0.0
0.5
1.0
1.5
2.0
2.5
3.0


spectral density
standard smoothed periodogram
AR smoothed periodogram
10^2
10^1
10^0
0.0
0.5
1.0
1.5
2.0
2.5
3.0


spectral density
standard smoothed periodogram
AR smoothed periodogram
10^2
10^1
10^0
0.0
0.5
1.0
1.5
2.0
2.5
3.0

CHAPTER

TWENTYNINE

ADDITIVE AND MULTIPLICATIVE FUNCTIONALS

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
```

29.1 Overview

Many economic time series display persistent growth that prevents them from being asymptotically stationary and ergodic.

For example, outputs, prices, and dividends typically display irregular but persistent growth.

Asymptotic stationarity and ergodicity are key assumptions needed to make it possible to learn by applying statistical methods.

Are there ways to model time series that have persistent growth that still enable statistical learning based on a law of large numbers for an asymptotically stationary and ergodic process?

The answer provided by Hansen and Scheinkman [HS09] is yes.

They described two classes of time series models that accommodate growth.

They are

1. **additive functionals** that display random "arithmetic growth"
2. **multiplicative functionals** that display random "geometric growth"

These two classes of processes are closely connected.

If a process $\{y_t\}$ is an additive functional and $\phi_t = \exp(y_t)$, then $\{\phi_t\}$ is a multiplicative functional.

Hansen and Sargent [HS08b] (chs. 5 and 8) describe discrete time versions of additive and multiplicative functionals.

In this lecture, we describe both additive functionals and multiplicative functionals.

We also describe and compute decompositions of additive and multiplicative processes into four components:

1. a **constant**
2. a **trend** component
3. an asymptotically **stationary** component
4. a **martingale**

We describe how to construct, simulate, and interpret these components.

More details about these concepts and algorithms can be found in Hansen and Sargent [HS08b].

Let's start with some imports:

```python
import numpy as np
import scipy as sp
import scipy.linalg as la
import quantecon as qe
import matplotlib.pyplot as plt
%matplotlib inline
from scipy.stats import norm, lognorm
```

29.2 A Particular Additive Functional

Hansen and Sargent [HS08b] describe a general class of additive functionals.

This lecture focuses on a subclass of these: a scalar process $\{y_t\}_{t=0}^{\infty}$ whose increments are driven by a Gaussian vector autoregression.

Our special additive functional displays interesting time series behavior while also being easy to construct, simulate, and analyze by using linear state-space tools.

We construct our additive functional from two pieces, the first of which is a **first-order vector autoregression** (VAR)

$$x_{t+1} = Ax_t + Bz_{t+1} \tag{1}$$

Here

- x_t is an $n \times 1$ vector,
- A is an $n \times n$ stable matrix (all eigenvalues lie within the open unit circle),
- $z_{t+1} \sim N(0, I)$ is an $m \times 1$ IID shock,
- B is an $n \times m$ matrix, and
- $x_0 \sim N(\mu_0, \Sigma_0)$ is a random initial condition for x

The second piece is an equation that expresses increments of $\{y_t\}_{t=0}^{\infty}$ as linear functions of

- a scalar constant ν,
- the vector x_t, and
- the same Gaussian vector z_{t+1} that appears in the VAR (1)

In particular,

$$y_{t+1} - y_t = \nu + Dx_t + Fz_{t+1} \tag{2}$$

Here $y_0 \sim N(\mu_{y0}, \Sigma_{y0})$ is a random initial condition for y.

The nonstationary random process $\{y_t\}_{t=0}^{\infty}$ displays systematic but random *arithmetic growth*.

29.2.1 Linear State-Space Representation

A convenient way to represent our additive functional is to use a linear state space system.

To do this, we set up state and observation vectors

$$\hat{x}_t = \begin{bmatrix} 1 \\ x_t \\ y_t \end{bmatrix} \quad \text{and} \quad \hat{y}_t = \begin{bmatrix} x_t \\ y_t \end{bmatrix}$$

Next we construct a linear system

$$\begin{bmatrix} 1 \\ x_{t+1} \\ y_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & A & 0 \\ \nu & D & 1 \end{bmatrix} \begin{bmatrix} 1 \\ x_t \\ y_t \end{bmatrix} + \begin{bmatrix} 0 \\ B \\ F \end{bmatrix} z_{t+1}$$

$$\begin{bmatrix} x_t \\ y_t \end{bmatrix} = \begin{bmatrix} 0 & I & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ x_t \\ y_t \end{bmatrix}$$

This can be written as

$$\begin{aligned} \hat{x}_{t+1} &= \hat{A}\hat{x}_t + \hat{B}z_{t+1} \\ \hat{y}_t &= \hat{D}\hat{x}_t \end{aligned}$$

which is a standard linear state space system.

To study it, we could map it into an instance of LinearStateSpace from QuantEcon.py.

But here we will use a different set of code for simulation, for reasons described below.

29.3 Dynamics

Let's run some simulations to build intuition.

In doing so we'll assume that z_{t+1} is scalar and that $\tilde{x}_t$ follows a 4th-order scalar autoregression.

$$\tilde{x}_{t+1} = \phi_1 \tilde{x}_t + \phi_2 \tilde{x}_{t-1} + \phi_3 \tilde{x}_{t-2} + \phi_4 \tilde{x}_{t-3} + \sigma z_{t+1} \tag{3}$$

in which the zeros z of the polynomial

$$\phi(z) = (1 - \phi_1 z - \phi_2 z^2 - \phi_3 z^3 - \phi_4 z^4)$$

are strictly greater than unity in absolute value.

(Being a zero of $\phi(z)$ means that $\phi(z) = 0$)

Let the increment in $\{y_t\}$ obey

$$y_{t+1} - y_t = \nu + \tilde{x}_t + \sigma z_{t+1}$$

with an initial condition for y_0.

While (3) is not a first order system like (1), we know that it can be mapped into a first order system.

- For an example of such a mapping, see this example.

In fact, this whole model can be mapped into the additive functional system definition in (1) – (2) by appropriate selection of the matrices A, B, D, F.

You can try writing these matrices down now as an exercise — correct expressions appear in the code below.

29.3.1 Simulation

When simulating we embed our variables into a bigger system.

This system also constructs the components of the decompositions of y_t and of $\exp(y_t)$ proposed by Hansen and Scheinkman [HS09].

All of these objects are computed using the code below

```
class AMF_LSS_VAR:
    """
    This class transforms an additive (multiplicative)
    functional into a QuantEcon linear state space system.
    """

    def __init__(self, A, B, D, F=None, ν=None):
        # Unpack required elements
        self.nx, self.nk = B.shape
        self.A, self.B = A, B

        # Checking the dimension of D (extended from the scalar case)
        if len(D.shape) > 1 and D.shape[0] != 1:
            self.nm = D.shape[0]
            self.D = D
        elif len(D.shape) > 1 and D.shape[0] == 1:
            self.nm = 1
            self.D = D
        else:
            self.nm = 1
            self.D = np.expand_dims(D, 0)

        # Create space for additive decomposition
        self.add_decomp = None
        self.mult_decomp = None

        # Set F
        if not np.any(F):
            self.F = np.zeros((self.nk, 1))
        else:
            self.F = F

        # Set ν
        if not np.any(ν):
            self.ν = np.zeros((self.nm, 1))
        elif type(ν) == float:
            self.ν = np.asarray([[ν]])
        elif len(ν.shape) == 1:
            self.ν = np.expand_dims(ν, 1)
        else:
            self.ν = ν
```

(continues on next page)

(continued from previous page)

```
        if self.ν.shape[0] != self.D.shape[0]:
            raise ValueError("The dimension of ν is inconsistent with D!")

        # Construct BIG state space representation
        self.lss = self.construct_ss()

    def construct_ss(self):
        """
        This creates the state space representation that can be passed
        into the quantecon LSS class.
        """
        # Pull out useful info
        nx, nk, nm = self.nx, self.nk, self.nm
        A, B, D, F, ν = self.A, self.B, self.D, self.F, self.ν
        if self.add_decomp:
            ν, H, g = self.add_decomp
        else:
            ν, H, g = self.additive_decomp()

        # Auxiliary blocks with 0's and 1's to fill out the lss matrices
        nx0c = np.zeros((nx, 1))
        nx0r = np.zeros(nx)
        nx1 = np.ones(nx)
        nk0 = np.zeros(nk)
        ny0c = np.zeros((nm, 1))
        ny0r = np.zeros(nm)
        ny1m = np.eye(nm)
        ny0m = np.zeros((nm, nm))
        nyx0m = np.zeros_like(D)

        # Build A matrix for LSS
        # Order of states is: [1, t, xt, yt, mt]
        A1 = np.hstack([1, 0, nx0r, ny0r, ny0r])          # Transition for 1
        A2 = np.hstack([1, 1, nx0r, ny0r, ny0r])          # Transition for t
        # Transition for x_{t+1}
        A3 = np.hstack([nx0c, nx0c, A, nyx0m.T, nyx0m.T])
        # Transition for y_{t+1}
        A4 = np.hstack([ν, ny0c, D, ny1m, ny0m])
        # Transition for m_{t+1}
        A5 = np.hstack([ny0c, ny0c, nyx0m, ny0m, ny1m])
        Abar = np.vstack([A1, A2, A3, A4, A5])

        # Build B matrix for LSS
        Bbar = np.vstack([nk0, nk0, B, F, H])

        # Build G matrix for LSS
        # Order of observation is: [xt, yt, mt, st, tt]
        # Selector for x_{t}
        G1 = np.hstack([nx0c, nx0c, np.eye(nx), nyx0m.T, nyx0m.T])
        G2 = np.hstack([ny0c, ny0c, nyx0m, ny1m, ny0m])    # Selector for y_{t}
        # Selector for martingale
        G3 = np.hstack([ny0c, ny0c, nyx0m, ny0m, ny1m])
        G4 = np.hstack([ny0c, ny0c, -g, ny0m, ny0m])  # Selector for stationary
        G5 = np.hstack([ny0c, ν, nyx0m, ny0m, ny0m])  # Selector for trend
        Gbar = np.vstack([G1, G2, G3, G4, G5])

        # Build H matrix for LSS
```

(continues on next page)

(continued from previous page)

```
        Hbar = np.zeros((Gbar.shape[0], nk))

        # Build LSS type
        x0 = np.hstack([1, 0, nx0r, ny0r, ny0r])
        S0 = np.zeros((len(x0), len(x0)))
        lss = qe.lss.LinearStateSpace(Abar, Bbar, Gbar, Hbar, mu_0=x0, Sigma_0=S0)

        return lss

    def additive_decomp(self):
        """
        Return values for the martingale decomposition
            - v     : unconditional mean difference in Y
            - H     : coefficient for the (linear) martingale component (κ_a)
            - g     : coefficient for the stationary component g(x)
            - Y_0   : it should be the function of X_0 (for now set it to 0.0)
        """
        I = np.identity(self.nx)
        A_res = la.solve(I - self.A, I)
        g = self.D @ A_res
        H = self.F + self.D @ A_res @ self.B

        return self.v, H, g

    def multiplicative_decomp(self):
        """
        Return values for the multiplicative decomposition (Example 5.4.4.)
            - v_tilde  : eigenvalue
            - H        : vector for the Jensen term
        """
        v, H, g = self.additive_decomp()
        v_tilde = v + (.5)*np.expand_dims(np.diag(H @ H.T), 1)

        return v_tilde, H, g

    def loglikelihood_path(self, x, y):
        A, B, D, F = self.A, self.B, self.D, self.F
        k, T = y.shape
        FF = F @ F.T
        FFinv = la.inv(FF)
        temp = y[:, 1:] - y[:, :-1] - D @ x[:, :-1]
        obs =  temp * FFinv * temp
        obssum = np.cumsum(obs)
        scalar = (np.log(la.det(FF)) + k*np.log(2*np.pi))*np.arange(1, T)

        return -(.5)*(obssum + scalar)

    def loglikelihood(self, x, y):
        llh = self.loglikelihood_path(x, y)

        return llh[-1]
```

Plotting

The code below adds some functions that generate plots for instances of the `AMF_LSS_VAR` *class*.

```python
def plot_given_paths(amf, T, ypath, mpath, spath, tpath,
                     mbounds, sbounds, horline=0, show_trend=True):

    # Allocate space
    trange = np.arange(T)

    # Create figure
    fig, ax = plt.subplots(2, 2, sharey=True, figsize=(15, 8))

    # Plot all paths together
    ax[0, 0].plot(trange, ypath[0, :], label="$y_t$", color="k")
    ax[0, 0].plot(trange, mpath[0, :], label="$m_t$", color="m")
    ax[0, 0].plot(trange, spath[0, :], label="$s_t$", color="g")
    if show_trend:
        ax[0, 0].plot(trange, tpath[0, :], label="$t_t$", color="r")
    ax[0, 0].axhline(horline, color="k", linestyle="-.")
    ax[0, 0].set_title("One Path of All Variables")
    ax[0, 0].legend(loc="upper left")

    # Plot Martingale Component
    ax[0, 1].plot(trange, mpath[0, :], "m")
    ax[0, 1].plot(trange, mpath.T, alpha=0.45, color="m")
    ub = mbounds[1, :]
    lb = mbounds[0, :]

    ax[0, 1].fill_between(trange, lb, ub, alpha=0.25, color="m")
    ax[0, 1].set_title("Martingale Components for Many Paths")
    ax[0, 1].axhline(horline, color="k", linestyle="-.")

    # Plot Stationary Component
    ax[1, 0].plot(spath[0, :], color="g")
    ax[1, 0].plot(spath.T, alpha=0.25, color="g")
    ub = sbounds[1, :]
    lb = sbounds[0, :]
    ax[1, 0].fill_between(trange, lb, ub, alpha=0.25, color="g")
    ax[1, 0].axhline(horline, color="k", linestyle="-.")
    ax[1, 0].set_title("Stationary Components for Many Paths")

    # Plot Trend Component
    if show_trend:
        ax[1, 1].plot(tpath.T, color="r")
    ax[1, 1].set_title("Trend Components for Many Paths")
    ax[1, 1].axhline(horline, color="k", linestyle="-.")

    return fig

def plot_additive(amf, T, npaths=25, show_trend=True):
    """
    Plots for the additive decomposition.
    Acts on an instance amf of the AMF_LSS_VAR class

    """
    # Pull out right sizes so we know how to increment
    nx, nk, nm = amf.nx, amf.nk, amf.nm
```

(continues on next page)

(continued from previous page)

```
# Allocate space (nm is the number of additive functionals -
# we want npaths for each)
mpath = np.empty((nm*npaths, T))
mbounds = np.empty((nm*2, T))
spath = np.empty((nm*npaths, T))
sbounds = np.empty((nm*2, T))
tpath = np.empty((nm*npaths, T))
ypath = np.empty((nm*npaths, T))

# Simulate for as long as we wanted
moment_generator = amf.lss.moment_sequence()
# Pull out population moments
for t in range (T):
    tmoms = next(moment_generator)
    ymeans = tmoms[1]
    yvar = tmoms[3]

    # Lower and upper bounds - for each additive functional
    for ii in range(nm):
        li, ui = ii*2, (ii+1)*2
        mscale = np.sqrt(yvar[nx+nm+ii, nx+nm+ii])
        sscale = np.sqrt(yvar[nx+2*nm+ii, nx+2*nm+ii])
        if mscale == 0.0:
            mscale = 1e-12   # avoids a RuntimeWarning from calculating ppf
        if sscale == 0.0:    # of normal distribution with std dev = 0.
            sscale = 1e-12   # sets std dev to small value instead

        madd_dist = norm(ymeans[nx+nm+ii], mscale)
        sadd_dist = norm(ymeans[nx+2*nm+ii], sscale)

        mbounds[li:ui, t] = madd_dist.ppf([0.01, .99])
        sbounds[li:ui, t] = sadd_dist.ppf([0.01, .99])

# Pull out paths
for n in range(npaths):
    x, y = amf.lss.simulate(T)
    for ii in range(nm):
        ypath[npaths*ii+n, :] = y[nx+ii, :]
        mpath[npaths*ii+n, :] = y[nx+nm + ii, :]
        spath[npaths*ii+n, :] = y[nx+2*nm + ii, :]
        tpath[npaths*ii+n, :] = y[nx+3*nm + ii, :]

add_figs = []

for ii in range(nm):
    li, ui = npaths*(ii), npaths*(ii+1)
    LI, UI = 2*(ii), 2*(ii+1)
    add_figs.append(plot_given_paths(amf, T,
                                     ypath[li:ui,:],
                                     mpath[li:ui,:],
                                     spath[li:ui,:],
                                     tpath[li:ui,:],
                                     mbounds[LI:UI,:],
                                     sbounds[LI:UI,:],
                                     show_trend=show_trend))
```

(continues on next page)

(continued from previous page)

```
        add_figs[ii].suptitle(f'Additive decomposition of $y_{ii+1}$',
                                fontsize=14)

    return add_figs


def plot_multiplicative(amf, T, npaths=25, show_trend=True):
    """
    Plots for the multiplicative decomposition

    """
    # Pull out right sizes so we know how to increment
    nx, nk, nm = amf.nx, amf.nk, amf.nm
    # Matrices for the multiplicative decomposition
    v_tilde, H, g = amf.multiplicative_decomp()

    # Allocate space (nm is the number of functionals -
    # we want npaths for each)
    mpath_mult = np.empty((nm*npaths, T))
    mbounds_mult = np.empty((nm*2, T))
    spath_mult = np.empty((nm*npaths, T))
    sbounds_mult = np.empty((nm*2, T))
    tpath_mult = np.empty((nm*npaths, T))
    ypath_mult = np.empty((nm*npaths, T))

    # Simulate for as long as we wanted
    moment_generator = amf.lss.moment_sequence()
    # Pull out population moments
    for t in range(T):
        tmoms = next(moment_generator)
        ymeans = tmoms[1]
        yvar = tmoms[3]

        # Lower and upper bounds - for each multiplicative functional
        for ii in range(nm):
            li, ui = ii*2, (ii+1)*2
            Mdist = lognorm(np.sqrt(yvar[nx+nm+ii, nx+nm+ii]).item(),
                            scale=np.exp(ymeans[nx+nm+ii] \
                                                    - t * (.5)
                                                    * np.expand_dims(
                                                        np.diag(H @ H.T),
                                                        1
                                                        )[ii]
                                                    ).item()
                                                )
            Sdist = lognorm(np.sqrt(yvar[nx+2*nm+ii, nx+2*nm+ii]).item(),
                            scale = np.exp(-ymeans[nx+2*nm+ii]).item())
            mbounds_mult[li:ui, t] = Mdist.ppf([.01, .99])
            sbounds_mult[li:ui, t] = Sdist.ppf([.01, .99])

    # Pull out paths
    for n in range(npaths):
        x, y = amf.lss.simulate(T)
        for ii in range(nm):
            ypath_mult[npaths*ii+n, :] = np.exp(y[nx+ii, :])
            mpath_mult[npaths*ii+n, :] = np.exp(y[nx+nm + ii, :] \
                                                - np.arange(T)*(.5)
```

(continues on next page)

(continued from previous page)

```
                                                    * np.expand_dims(np.diag(H
                                                                    @ H.T),
                                                                    1)[ii]
                                                    )
            spath_mult[npaths*ii+n, :] = 1/np.exp(-y[nx+2*nm + ii, :])
            tpath_mult[npaths*ii+n, :] = np.exp(y[nx+3*nm + ii, :]
                                                    + np.arange(T)*(.5)
                                                    * np.expand_dims(np.diag(H
                                                                    @ H.T),
                                                                    1)[ii]
                                                    )

    mult_figs = []

    for ii in range(nm):
        li, ui = npaths*(ii), npaths*(ii+1)
        LI, UI = 2*(ii), 2*(ii+1)

        mult_figs.append(plot_given_paths(amf,T,
                                                    ypath_mult[li:ui,:],
                                                    mpath_mult[li:ui,:],
                                                    spath_mult[li:ui,:],
                                                    tpath_mult[li:ui,:],
                                                    mbounds_mult[LI:UI,:],
                                                    sbounds_mult[LI:UI,:],
                                                    1,
                                                    show_trend=show_trend))
        mult_figs[ii].suptitle(f'Multiplicative decomposition of \
                                            $y_{ii+1}$', fontsize=14)

    return mult_figs

def plot_martingale_paths(amf, T, mpath, mbounds, horline=1, show_trend=False):
    # Allocate space
    trange = np.arange(T)

    # Create figure
    fig, ax = plt.subplots(1, 1, figsize=(10, 6))

    # Plot Martingale Component
    ub = mbounds[1, :]
    lb = mbounds[0, :]
    ax.fill_between(trange, lb, ub, color="#ffccff")
    ax.axhline(horline, color="k", linestyle="-.")
    ax.plot(trange, mpath.T, linewidth=0.25, color="#4c4c4c")

    return fig

def plot_martingales(amf, T, npaths=25):

    # Pull out right sizes so we know how to increment
    nx, nk, nm = amf.nx, amf.nk, amf.nm
    # Matrices for the multiplicative decomposition
    v_tilde, H, g = amf.multiplicative_decomp()

    # Allocate space (nm is the number of functionals -
    # we want npaths for each)
```

(continues on next page)

(continued from previous page)

```
    mpath_mult = np.empty((nm*npaths, T))
    mbounds_mult = np.empty((nm*2, T))

    # Simulate for as long as we wanted
    moment_generator = amf.lss.moment_sequence()
    # Pull out population moments
    for t in range (T):
        tmoms = next(moment_generator)
        ymeans = tmoms[1]
        yvar = tmoms[3]

        # Lower and upper bounds - for each functional
        for ii in range(nm):
            li, ui = ii*2, (ii+1)*2
            Mdist = lognorm(np.sqrt(yvar[nx+nm+ii, nx+nm+ii]).item(),
                            scale= np.exp(ymeans[nx+nm+ii] \
                                                        - t * (.5)
                                                        * np.expand_dims(
                                                            np.diag(H @ H.T),
                                                            1)[ii]

                                                    ).item()
                            )
            mbounds_mult[li:ui, t] = Mdist.ppf([.01, .99])

    # Pull out paths
    for n in range(npaths):
        x, y = amf.lss.simulate(T)
        for ii in range(nm):
            mpath_mult[npaths*ii+n, :] = np.exp(y[nx+nm + ii, :] \
                                                    - np.arange(T) * (.5)
                                                    * np.expand_dims(np.diag(H
                                                                        @ H.T),
                                                                    1)[ii]
                                                    )

    mart_figs = []

    for ii in range(nm):
        li, ui = npaths*(ii), npaths*(ii+1)
        LI, UI = 2*(ii), 2*(ii+1)
        mart_figs.append(plot_martingale_paths(amf, T, mpath_mult[li:ui, :],
                                                    mbounds_mult[LI:UI, :],
                                                    horline=1))
        mart_figs[ii].suptitle(f'Martingale components for many paths of \
                                $y_{ii+1}$', fontsize=14)

    return mart_figs
```

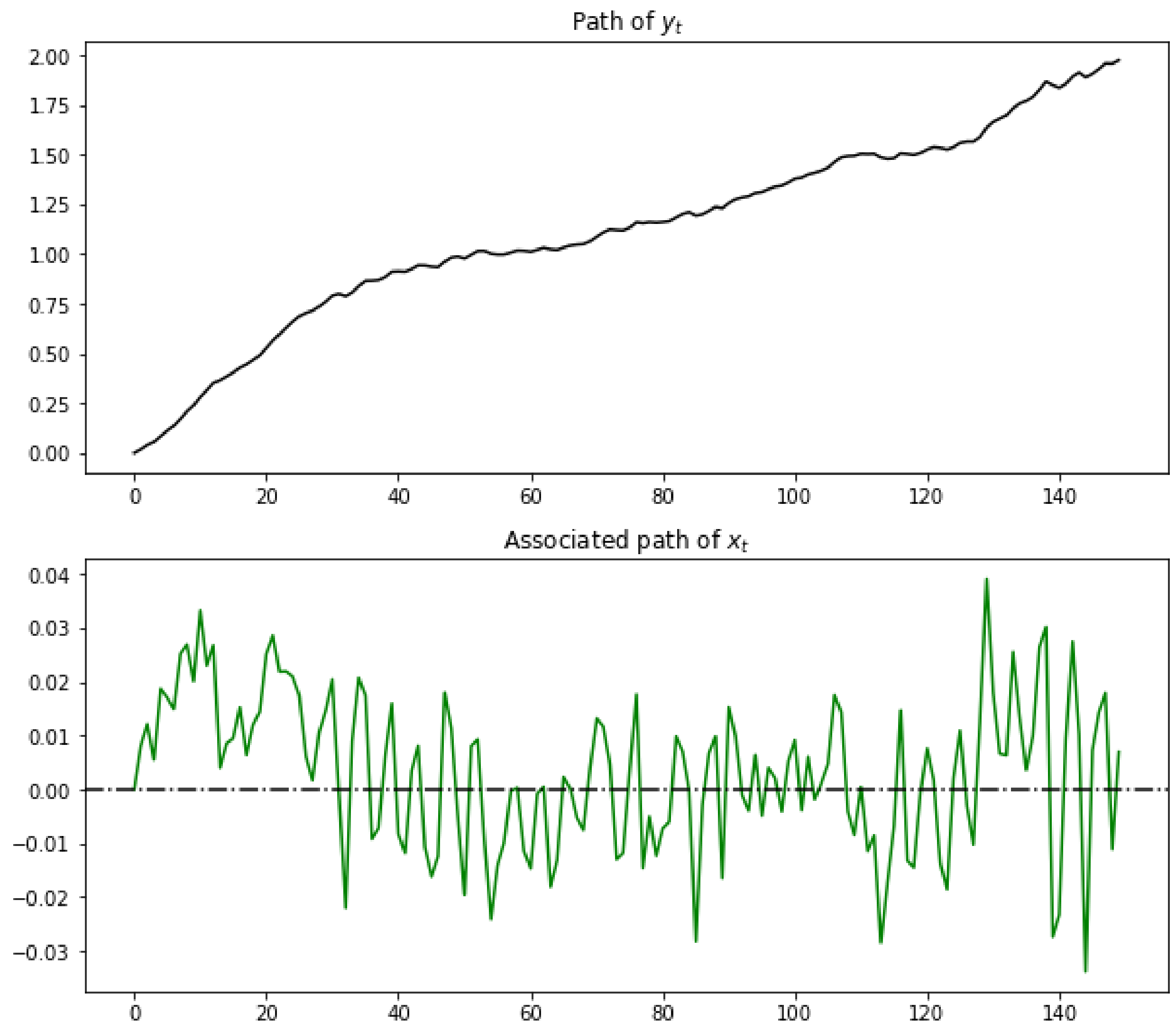
For now, we just plot y_t and x_t, postponing until later a description of exactly how we compute them.

```
ϕ_1, ϕ_2, ϕ_3, ϕ_4 = 0.5, -0.2, 0, 0.5
σ = 0.01
ν = 0.01   # Growth rate

# A matrix should be n x n
A = np.array([[ϕ_1, ϕ_2, ϕ_3, ϕ_4],
```

(continues on next page)

(continued from previous page)

```
              [  1,    0,    0,    0],
              [  0,    1,    0,    0],
              [  0,    0,    1,    0]])

# B matrix should be n x k
B = np.array([[σ, 0, 0, 0]]).T

D = np.array([1, 0, 0, 0]) @ A
F = np.array([1, 0, 0, 0]) @ B

amf = AMF_LSS_VAR(A, B, D, F, ν=ν)

T = 150
x, y = amf.lss.simulate(T)

fig, ax = plt.subplots(2, 1, figsize=(10, 9))

ax[0].plot(np.arange(T), y[amf.nx, :], color='k')
ax[0].set_title('Path of $y_t$')
ax[1].plot(np.arange(T), y[0, :], color='g')
ax[1].axhline(0, color='k', linestyle='-.')
ax[1].set_title('Associated path of $x_t$')
plt.show()
```



Notice the irregular but persistent growth in y_t.

29.3.2 Decomposition

Hansen and Sargent [HS08b] describe how to construct a decomposition of an additive functional into four parts:

- a constant inherited from initial values x_0 and y_0
- a linear trend
- a martingale
- an (asymptotically) stationary component

To attain this decomposition for the particular class of additive functionals defined by (1) and (2), we first construct the matrices

$$
\begin{aligned}
H &:= F + D(I - A)^{-1}B \\
g &:= D(I - A)^{-1}
\end{aligned}
$$

Then the Hansen-Scheinkman [HS09] decomposition is

$$y_t = \underbrace{t\nu}_{\text{trend component}} + \overbrace{\sum_{j=1}^{t} H z_j}^{\text{Martingale component}} - \underbrace{g x_t}_{\text{stationary component}} + \overbrace{g x_0 + y_0}^{\text{initial conditions}}$$

At this stage, you should pause and verify that $y_{t+1} - y_t$ satisfies (2).

It is convenient for us to introduce the following notation:

- $\tau_t = \nu t$, a linear, deterministic trend
- $m_t = \sum_{j=1}^{t} H z_j$, a martingale with time $t+1$ increment $H z_{t+1}$
- $s_t = g x_t$, an (asymptotically) stationary component

We want to characterize and simulate components τ_t, m_t, s_t of the decomposition.

A convenient way to do this is to construct an appropriate instance of a linear state space system by using LinearStateSpace from QuantEcon.py.

This will allow us to use the routines in LinearStateSpace to study dynamics.

To start, observe that, under the dynamics in (1) and (2) and with the definitions just given,

$$\begin{bmatrix} 1 \\ t+1 \\ x_{t+1} \\ y_{t+1} \\ m_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & A & 0 & 0 \\ \nu & 0 & D & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ t \\ x_t \\ y_t \\ m_t \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ B \\ F \\ H \end{bmatrix} z_{t+1}$$

and

$$\begin{bmatrix} x_t \\ y_t \\ \tau_t \\ m_t \\ s_t \end{bmatrix} = \begin{bmatrix} 0 & 0 & I & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & \nu & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & -g & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ t \\ x_t \\ y_t \\ m_t \end{bmatrix}$$

With

$$\tilde{x} := \begin{bmatrix} 1 \\ t \\ x_t \\ y_t \\ m_t \end{bmatrix} \quad \text{and} \quad \tilde{y} := \begin{bmatrix} x_t \\ y_t \\ \tau_t \\ m_t \\ s_t \end{bmatrix}$$

we can write this as the linear state space system

$$\begin{aligned} \tilde{x}_{t+1} &= \tilde{A} \tilde{x}_t + \tilde{B} z_{t+1} \\ \tilde{y}_t &= \tilde{D} \tilde{x}_t \end{aligned}$$

By picking out components of $\tilde{y}_t$, we can track all variables of interest.

29.4 Code

The class `AMF_LSS_VAR` mentioned *above* does all that we want to study our additive functional.

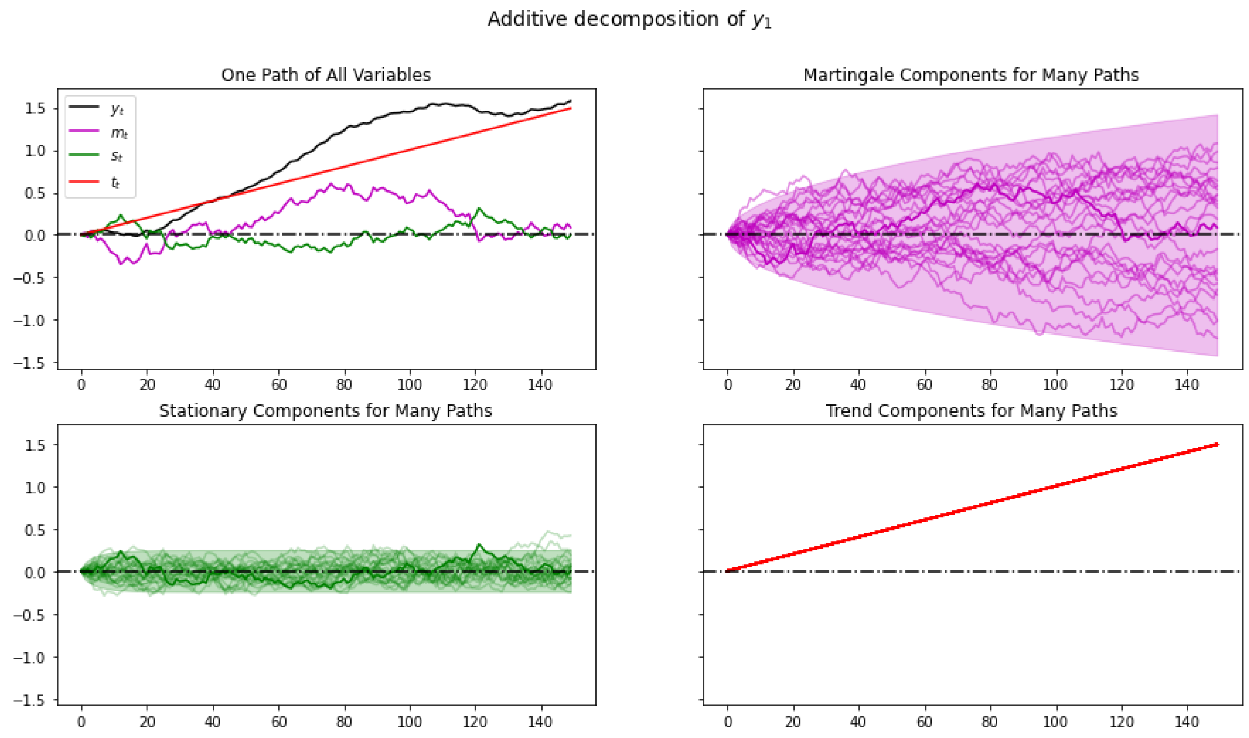
In fact, `AMF_LSS_VAR` does more because it allows us to study an associated multiplicative functional as well.

(A hint that it does more is the name of the class – here AMF stands for "additive and multiplicative functional" – the code computes and displays objects associated with multiplicative functionals too.)

Let's use this code (embedded above) to explore the *example process described above*.

If you run *the code that first simulated that example* again and then the method call you will generate (modulo randomness) the plot

```
plot_additive(amf, T)
plt.show()
```



When we plot multiple realizations of a component in the 2nd, 3rd, and 4th panels, we also plot the population 95% probability coverage sets computed using the LinearStateSpace class.

We have chosen to simulate many paths, all starting from the *same* non-random initial conditions x_0, y_0 (you can tell this from the shape of the 95% probability coverage shaded areas).

Notice tell-tale signs of these probability coverage shaded areas

- the purple one for the martingale component m_t grows with $\sqrt{t}$
- the green one for the stationary component s_t converges to a constant band

29.4.1 Associated Multiplicative Functional

Where $\{y_t\}$ is our additive functional, let $M_t = \exp(y_t)$.

As mentioned above, the process $\{M_t\}$ is called a **multiplicative functional**.

Corresponding to the additive decomposition described above we have a multiplicative decomposition of M_t

$$\frac{M_t}{M_0} = \exp(t\nu)\exp\Bigl(\sum_{j=1}^{t} H \cdot Z_j\Bigr)\exp\Bigl(D(I-A)^{-1}x_0 - D(I-A)^{-1}x_t\Bigr)$$

or

$$\frac{M_t}{M_0} = \exp(\tilde{\nu}t)\left(\frac{\widetilde{M}_t}{\widetilde{M}_0}\right)\left(\frac{\tilde{e}(X_0)}{\tilde{e}(x_t)}\right)$$

where

$$\tilde{\nu} = \nu + \frac{H \cdot H}{2}, \quad \widetilde{M}_t = \exp\biggl(\sum_{j=1}^{t}\Bigl(H \cdot z_j - \frac{H \cdot H}{2}\Bigr)\biggr), \quad \widetilde{M}_0 = 1$$

and

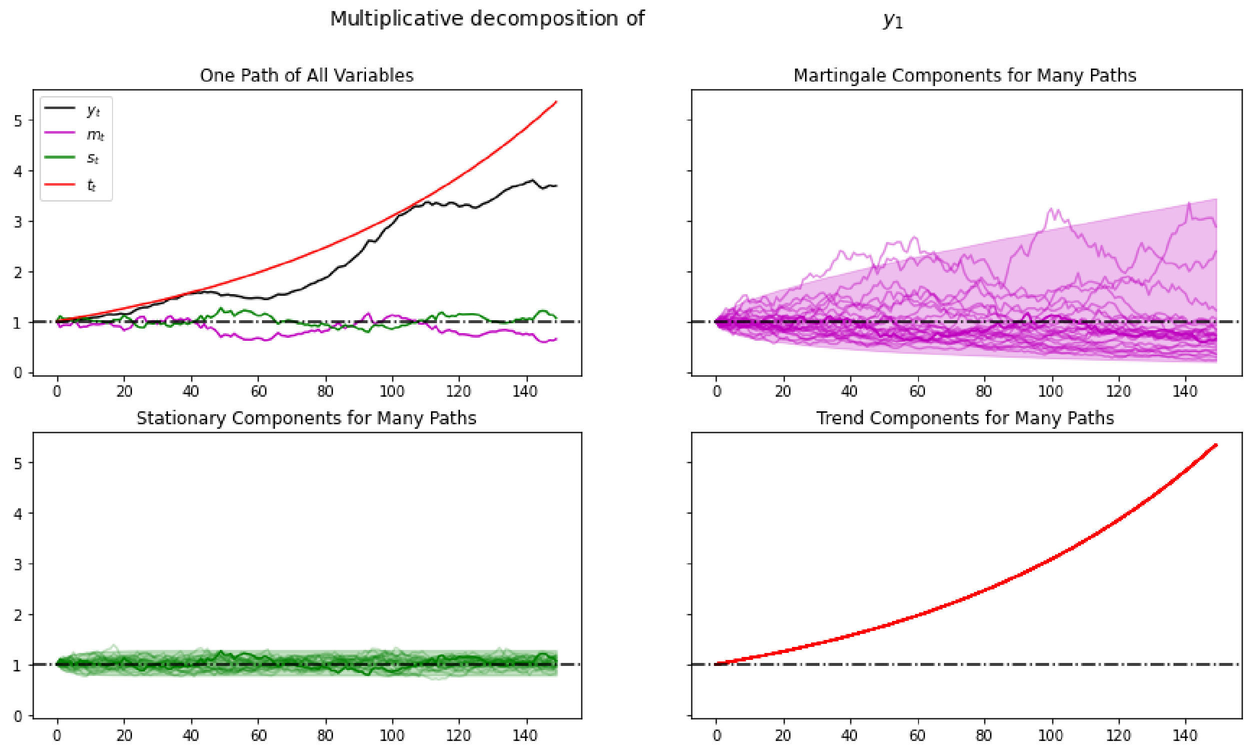
$$\tilde{e}(x) = \exp[g(x)] = \exp[D(I-A)^{-1}x]$$

An instance of class `AMF_LSS_VAR` (*above*) includes this associated multiplicative functional as an attribute.

Let's plot this multiplicative functional for our example.

If you run *the code that first simulated that example* again and then the method call in the cell below you'll obtain the graph in the next cell.

```
plot_multiplicative(amf, T)
plt.show()
```



As before, when we plotted multiple realizations of a component in the 2nd, 3rd, and 4th panels, we also plotted population 95% confidence bands computed using the LinearStateSpace class.

Comparing this figure and the last also helps show how geometric growth differs from arithmetic growth.

The top right panel of the above graph shows a panel of martingales associated with the panel of $M_t = \exp(y_t)$ that we have generated for a limited horizon T.

It is interesting to how the martingale behaves as $T \to +\infty$.

Let's see what happens when we set $T = 12000$ instead of 150.

29.4.2 Peculiar Large Sample Property

Hansen and Sargent [HS08b] (ch. 8) describe the following two properties of the martingale component $\widetilde{M}_t$ of the multiplicative decomposition

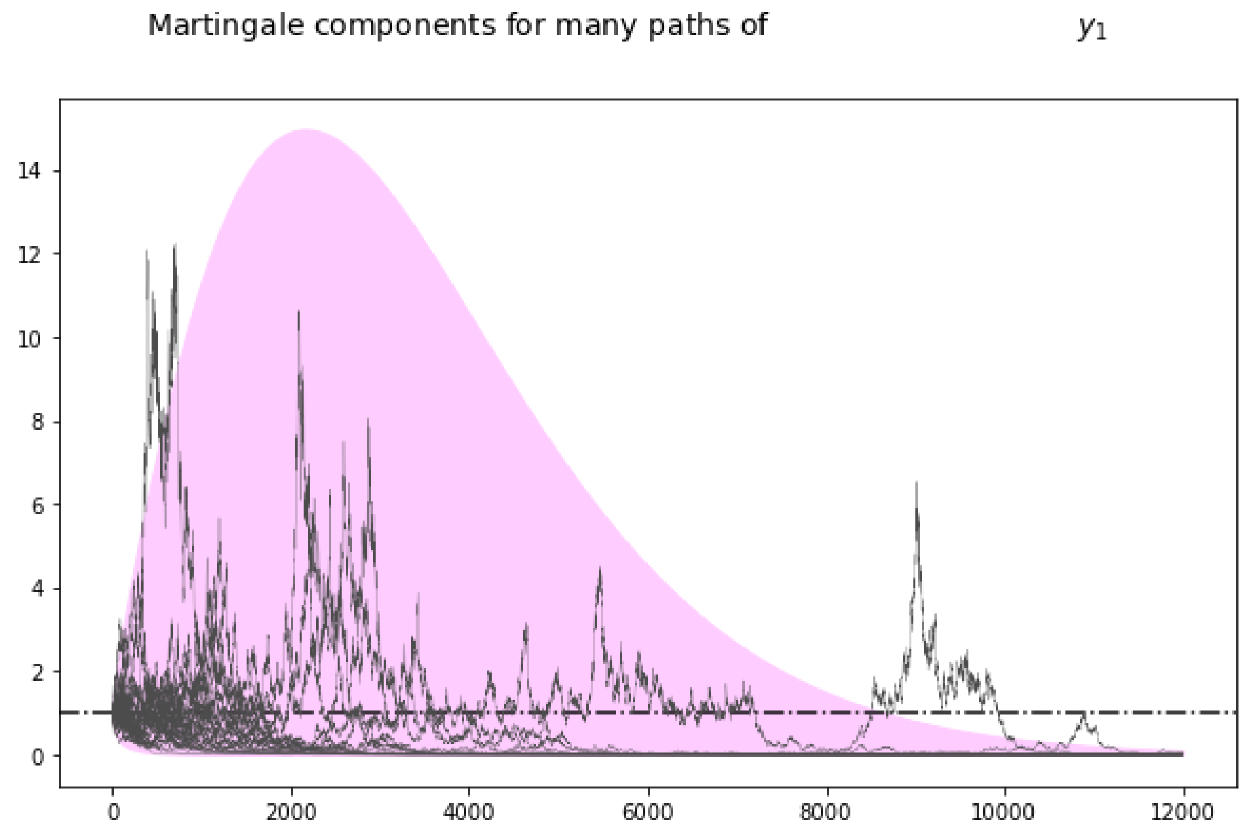
- while $E_0 \widetilde{M}_t = 1$ for all $t \geq 0$, nevertheless …
- as $t \to +\infty$, $\widetilde{M}_t$ converges to zero almost surely

The first property follows from the fact that $\widetilde{M}_t$ is a multiplicative martingale with initial condition $\widetilde{M}_0 = 1$.

The second is a **peculiar property** noted and proved by Hansen and Sargent [HS08b].

The following simulation of many paths of $\widetilde{M}_t$ illustrates both properties

```
np.random.seed(10021987)
plot_martingales(amf, 12000)
plt.show()
```



The dotted line in the above graph is the mean $E\tilde{M}_t = 1$ of the martingale.

It remains constant at unity, illustrating the first property.

The purple 95 percent frequency coverage interval collapses around zero, illustrating the second property.

29.5 More About the Multiplicative Martingale

Let's drill down and study probability distribution of the multiplicative martingale $\{\widetilde{M}_t\}_{t=0}^\infty$ in more detail.

As we have seen, it has representation

$$\widetilde{M}_t = \exp\left(\sum_{j=1}^t \left(H \cdot z_j - \frac{H \cdot H}{2}\right)\right), \quad \widetilde{M}_0 = 1$$

where $H = [F + D(I - A)^{-1}B]$.

It follows that $\log \widetilde{M}_t \sim \mathcal{N}(-\frac{tH \cdot H}{2}, tH \cdot H)$ and that consequently $\widetilde{M}_t$ is log normal.

29.5.1 Simulating a Multiplicative Martingale Again

Next, we want a program to simulate the likelihood ratio process $\{\tilde{M}_t\}_{t=0}^\infty$.

In particular, we want to simulate 5000 sample paths of length T for the case in which x is a scalar and $[A, B, D, F] = [0.8, 0.001, 1.0, 0.01]$ and $\nu = 0.005$.

After accomplishing this, we want to display and study histograms of $\tilde{M}_T^i$ for various values of T.

Here is code that accomplishes these tasks.

29.5.2 Sample Paths

Let's write a program to simulate sample paths of $\{x_t, y_t\}_{t=0}^\infty$.

We'll do this by formulating the additive functional as a linear state space model and putting the LinearStateSpace class to work.

```python
class AMF_LSS_VAR:
    """
    This class is written to transform a scalar additive functional
    into a linear state space system.
    """
    def __init__(self, A, B, D, F=0.0, ν=0.0):
        # Unpack required elements
        self.A, self.B, self.D, self.F, self.ν = A, B, D, F, ν

        # Create space for additive decomposition
        self.add_decomp = None
        self.mult_decomp = None

        # Construct BIG state space representation
        self.lss = self.construct_ss()

    def construct_ss(self):
```

(continues on next page)

(continued from previous page)

```
        """
        This creates the state space representation that can be passed
        into the quantecon LSS class.
        """
        # Pull out useful info
        A, B, D, F, ν = self.A, self.B, self.D, self.F, self.ν
        nx, nk, nm = 1, 1, 1
        if self.add_decomp:
            ν, H, g = self.add_decomp
        else:
            ν, H, g = self.additive_decomp()

        # Build A matrix for LSS
        # Order of states is: [1, t, xt, yt, mt]
        A1 = np.hstack([1, 0, 0, 0, 0])       # Transition for 1
        A2 = np.hstack([1, 1, 0, 0, 0])       # Transition for t
        A3 = np.hstack([0, 0, A, 0, 0])       # Transition for x_{t+1}
        A4 = np.hstack([ν, 0, D, 1, 0])       # Transition for y_{t+1}
        A5 = np.hstack([0, 0, 0, 0, 1])       # Transition for m_{t+1}
        Abar = np.vstack([A1, A2, A3, A4, A5])

        # Build B matrix for LSS
        Bbar = np.vstack([0, 0, B, F, H])

        # Build G matrix for LSS
        # Order of observation is: [xt, yt, mt, st, tt]
        G1 = np.hstack([0, 0, 1, 0, 0])            # Selector for x_{t}
        G2 = np.hstack([0, 0, 0, 1, 0])            # Selector for y_{t}
        G3 = np.hstack([0, 0, 0, 0, 1])            # Selector for martingale
        G4 = np.hstack([0, 0, -g, 0, 0])           # Selector for stationary
        G5 = np.hstack([0, ν, 0, 0, 0])            # Selector for trend
        Gbar = np.vstack([G1, G2, G3, G4, G5])

        # Build H matrix for LSS
        Hbar = np.zeros((1, 1))

        # Build LSS type
        x0 = np.hstack([1, 0, 0, 0, 0])
        S0 = np.zeros((5, 5))
        lss = qe.lss.LinearStateSpace(Abar, Bbar, Gbar, Hbar,
                                      mu_0=x0, Sigma_0=S0)

        return lss

    def additive_decomp(self):
        """
        Return values for the martingale decomposition (Proposition 4.3.3.)
            - ν     : unconditional mean difference in Y
            - H     : coefficient for the (linear) martingale component (kappa_a)
            - g     : coefficient for the stationary component g(x)
            - Y_0   : it should be the function of X_0 (for now set it to 0.0)
        """
        A_res = 1 / (1 - self.A)
        g = self.D * A_res
        H = self.F + self.D * A_res * self.B

        return self.ν, H, g
```

(continues on next page)

(continued from previous page)

```
    def multiplicative_decomp(self):
        """
        Return values for the multiplicative decomposition (Example 5.4.4.)
            - v_tilde  : eigenvalue
            - H        : vector for the Jensen term
        """
        ν, H, g = self.additive_decomp()
        ν_tilde = ν + (.5) * H**2

        return ν_tilde, H, g

    def loglikelihood_path(self, x, y):
        A, B, D, F = self.A, self.B, self.D, self.F
        T = y.T.size
        FF = F**2
        FFinv = 1 / FF
        temp = y[1:] - y[:-1] - D * x[:-1]
        obs = temp * FFinv * temp
        obssum = np.cumsum(obs)
        scalar = (np.log(FF) + np.log(2 * np.pi)) * np.arange(1, T)

        return (-0.5) * (obssum + scalar)

    def loglikelihood(self, x, y):
        llh = self.loglikelihood_path(x, y)

        return llh[-1]
```

The heavy lifting is done inside the `AMF_LSS_VAR` class.

The following code adds some simple functions that make it straightforward to generate sample paths from an instance of `AMF_LSS_VAR`.

```
def simulate_xy(amf, T):
    "Simulate individual paths."
    foo, bar = amf.lss.simulate(T)
    x = bar[0, :]
    y = bar[1, :]

    return x, y

def simulate_paths(amf, T=150, I=5000):
    "Simulate multiple independent paths."

    # Allocate space
    storeX = np.empty((I, T))
    storeY = np.empty((I, T))

    for i in range(I):
        # Do specific simulation
        x, y = simulate_xy(amf, T)

        # Fill in our storage matrices
        storeX[i, :] = x
        storeY[i, :] = y
```

(continues on next page)

(continued from previous page)

```
    return storeX, storeY

def population_means(amf, T=150):
    # Allocate Space
    xmean = np.empty(T)
    ymean = np.empty(T)

    # Pull out moment generator
    moment_generator = amf.lss.moment_sequence()

    for tt in range (T):
        tmoms = next(moment_generator)
        ymeans = tmoms[1]
        xmean[tt] = ymeans[0]
        ymean[tt] = ymeans[1]

    return xmean, ymean
```

Now that we have these functions in our toolkit, let's apply them to run some simulations.

```
def simulate_martingale_components(amf, T=1000, I=5000):
    # Get the multiplicative decomposition
    ν, H, g = amf.multiplicative_decomp()

    # Allocate space
    add_mart_comp = np.empty((I, T))

    # Simulate and pull out additive martingale component
    for i in range(I):
        foo, bar = amf.lss.simulate(T)

        # Martingale component is third component
        add_mart_comp[i, :] = bar[2, :]

    mul_mart_comp = np.exp(add_mart_comp - (np.arange(T) * H**2)/2)

    return add_mart_comp, mul_mart_comp


# Build model
amf_2 = AMF_LSS_VAR(0.8, 0.001, 1.0, 0.01,.005)

amc, mmc = simulate_martingale_components(amf_2, 1000, 5000)

amcT = amc[:, -1]
mmcT = mmc[:, -1]

print("The (min, mean, max) of additive Martingale component in period T is")
print(f"\t ({np.min(amcT)}, {np.mean(amcT)}, {np.max(amcT)})")

print("The (min, mean, max) of multiplicative Martingale component \
in period T is")
print(f"\t ({np.min(mmcT)}, {np.mean(mmcT)}, {np.max(mmcT)})")
```

```
The (min, mean, max) of additive Martingale component in period T is
         (-1.8379907335579106, 0.011040789361757435, 1.4697384727035145)
```

(continues on next page)

(continued from previous page)

```
The (min, mean, max) of multiplicative Martingale component in period T is
         (0.14222026893384476, 1.006753060146832, 3.8858858377907133)
```

Let's plot the probability density functions for $\log \widetilde{M}_t$ for $t = 100, 500, 1000, 10000, 100000$.

Then let's use the plots to investigate how these densities evolve through time.

We will plot the densities of $\log \widetilde{M}_t$ for different values of t.

Note: `scipy.stats.lognorm` expects you to pass the standard deviation first $(tH \cdot H)$ and then the exponent of the mean as a keyword argument `scale (scale=np.exp(-t * H2 / 2))`.

- See the documentation here.

This is peculiar, so make sure you are careful in working with the log normal distribution.

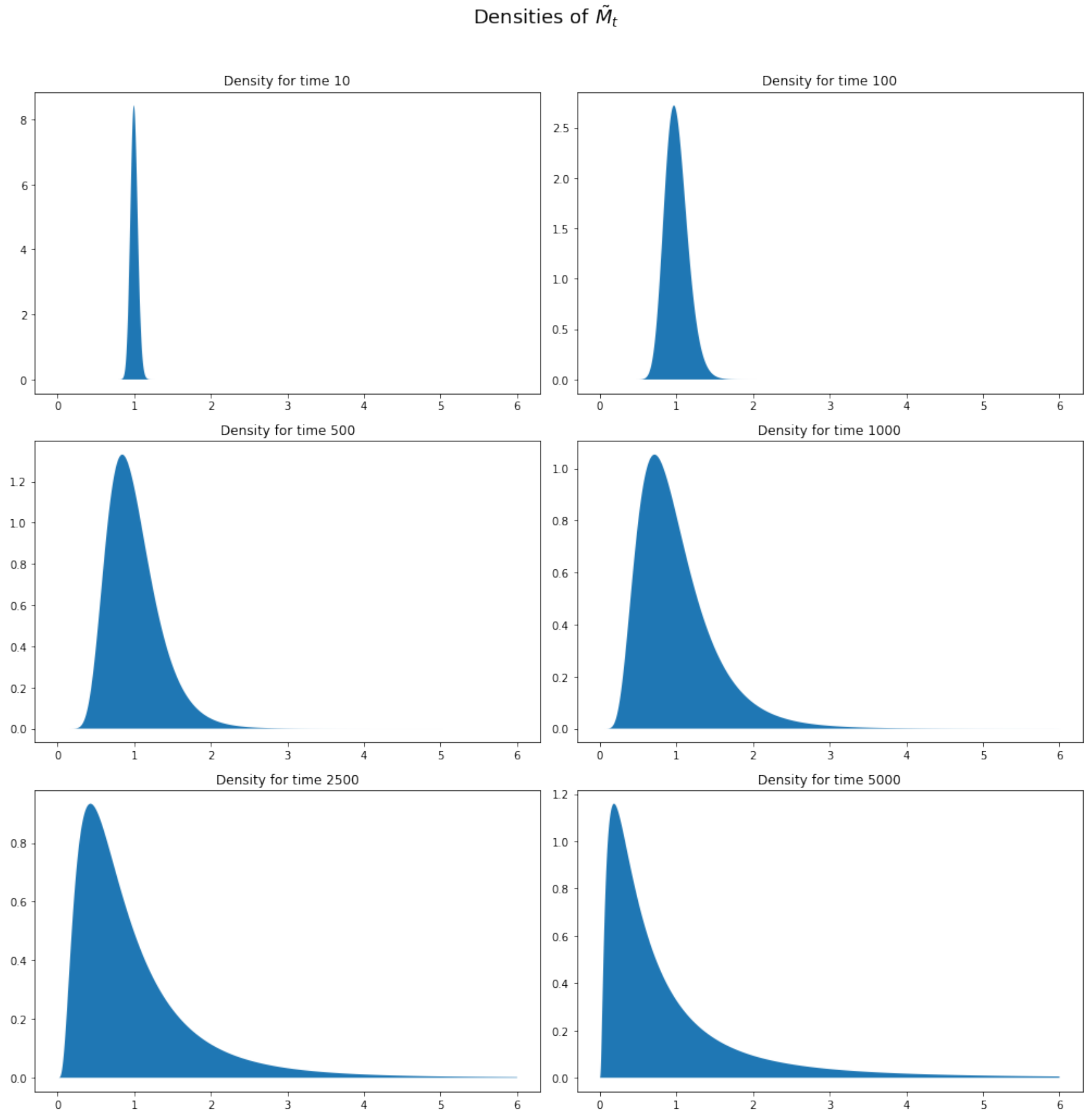
Here is some code that tackles these tasks

```python
def Mtilde_t_density(amf, t, xmin=1e-8, xmax=5.0, npts=5000):

    # Pull out the multiplicative decomposition
    vtilde, H, g = amf.multiplicative_decomp()
    H2 = H*H

    # The distribution
    mdist = lognorm(np.sqrt(t*H2), scale=np.exp(-t*H2/2))
    x = np.linspace(xmin, xmax, npts)
    pdf = mdist.pdf(x)

    return x, pdf


def logMtilde_t_density(amf, t, xmin=-15.0, xmax=15.0, npts=5000):

    # Pull out the multiplicative decomposition
    vtilde, H, g = amf.multiplicative_decomp()
    H2 = H*H

    # The distribution
    lmdist = norm(-t*H2/2, np.sqrt(t*H2))
    x = np.linspace(xmin, xmax, npts)
    pdf = lmdist.pdf(x)

    return x, pdf


times_to_plot = [10, 100, 500, 1000, 2500, 5000]
dens_to_plot = map(lambda t: Mtilde_t_density(amf_2, t, xmin=1e-8, xmax=6.0),
                   times_to_plot)
ldens_to_plot = map(lambda t: logMtilde_t_density(amf_2, t, xmin=-10.0,
                    xmax=10.0), times_to_plot)

fig, ax = plt.subplots(3, 2, figsize=(14, 14))
ax = ax.flatten()

fig.suptitle(r"Densities of $\tilde{M}_t$", fontsize=18, y=1.02)
for (it, dens_t) in enumerate(dens_to_plot):
    x, pdf = dens_t
    ax[it].set_title(f"Density for time {times_to_plot[it]}")
```

(continues on next page)

(continued from previous page)

```
    ax[it].fill_between(x, np.zeros_like(pdf), pdf)

plt.tight_layout()
plt.show()
```



These probability density functions help us understand mechanics underlying the **peculiar property** of our multiplicative martingale

- As T grows, most of the probability mass shifts leftward toward zero.
- For example, note that most mass is near 1 for $T = 10$ or $T = 100$ but most of it is near 0 for $T = 5000$.
- As T grows, the tail of the density of $\widetilde{M}_T$ lengthens toward the right.

- Enough mass moves toward the right tail to keep $E\widetilde{M}_T = 1$ even as most mass in the distribution of $\widetilde{M}_T$ collapses around 0.

29.5.3 Multiplicative Martingale as Likelihood Ratio Process

This lecture studies **likelihood processes** and **likelihood ratio processes**.

A **likelihood ratio process** is a multiplicative martingale with mean unity.

Likelihood ratio processes exhibit the peculiar property that naturally also appears here.

CHAPTER

THIRTY

CLASSICAL CONTROL WITH LINEAR ALGEBRA

Contents

30.1 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:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

30.1.1 References

Useful references include [Whi63], [HS80], [Orf88], [AP91], and [Mut60].

30.2 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

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

where $d_0, d_1, \ldots, d_m$ 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 $y_{-1}, y_{-2} \ldots, y_{-m}$.

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

30.2.1 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 [Sar87].

30.3 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 $y_0, y_1, \dots, y_N$, and
3. setting these derivatives to zero.

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

For $t = N - m + 1, \dots, N$, 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 $t = 0,\ 1,\ \dots,\ N - m$ 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

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

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

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

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

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

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.

30.3.1 Matrix Methods

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

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

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

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}.

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 $a_t, a_{t+1}, \dots, a_N$.

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:

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

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

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

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

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

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

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

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.

30.4 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

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

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

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

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, $z_1, z_2, \dots, z_{2m}$.

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 $z_{2m-j} = \beta z_{j+1}^{-1}, j = 0, 1, \dots, m-1$.

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 $j = 1, \dots, m$ 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

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

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

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

where

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

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

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 [Sar87].

To exhibit the solution in a form paralleling that of [Sar87], 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 [Sar87]).

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 $\phi = \max \{\beta \lambda_i, i = 1, \dots, m\}$.

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).

30.5 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

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

The factorization of the characteristic polynomial (16) becomes

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

where

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

The solution of the problem becomes

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

30.5.1 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 $\beta^t\,[d\,(L)y_t]^2 = [\tilde d\,(L)\tilde y_t]^2$ with $\tilde d\,(L) = \sum_{j=0}^m \tilde d_j\, L^j$ and $\tilde d_j = \beta^{j/2} d_j$.

Then the original criterion function (1) is equivalent to

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

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

The Euler equation for this problem is $[h + \tilde d\,(L^{-1})\,\tilde d\,(L)]\,\tilde y_t = \tilde a_t$.

The solution is

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

or

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

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

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

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

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

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.

30.6 Implementation

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

```python
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)
```

(continues on next page)

(continued from previous page)

```
        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 ϕ 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 ϕ_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
```

(continues on next page)

(continued from previous page)

```
        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)
        """
```

(continues on next page)

(continued from previous page)

```
        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)]
```

(continues on next page)

(continued from previous page)

```
        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
```

(continues on next page)

(continued from previous page)

```
        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
```

30.6.1 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 $a_t y_t - hy_t^2/2$ 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.

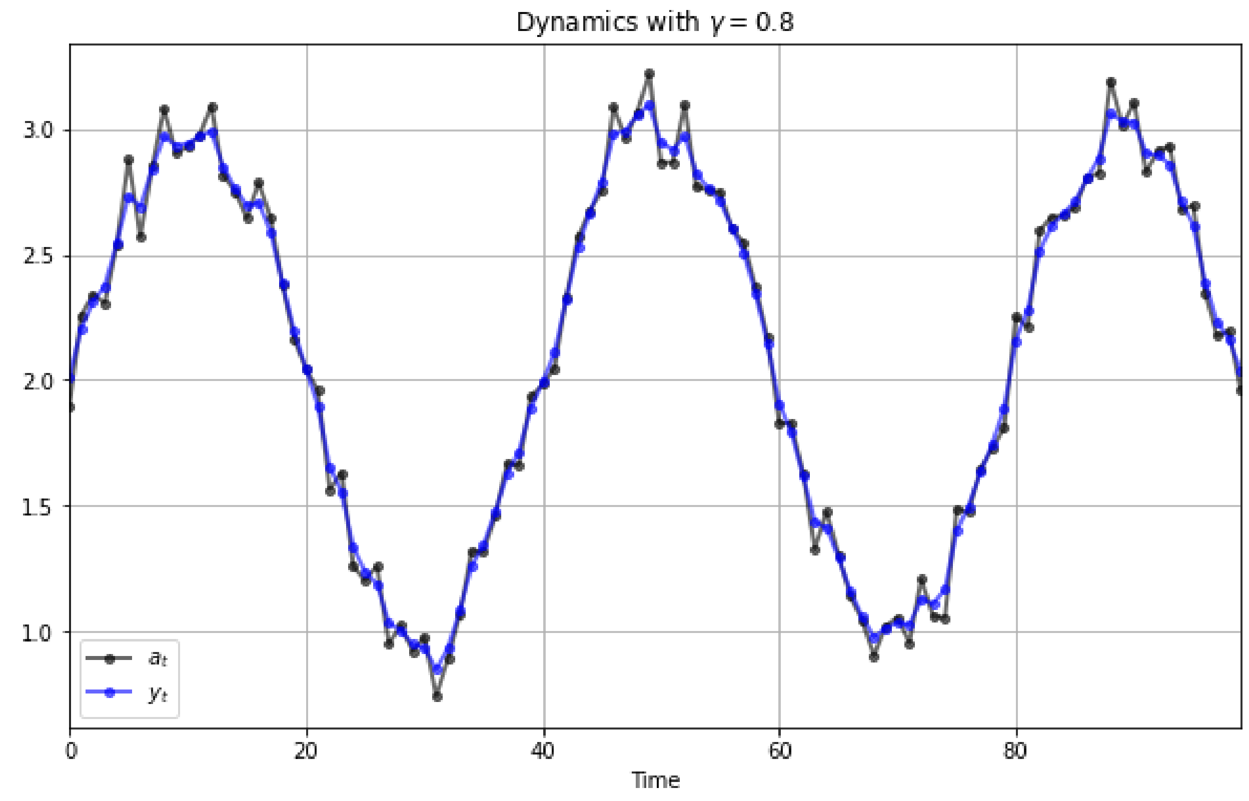
However, as we increase γ, the agent gives greater weight to a smooth time path.

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.

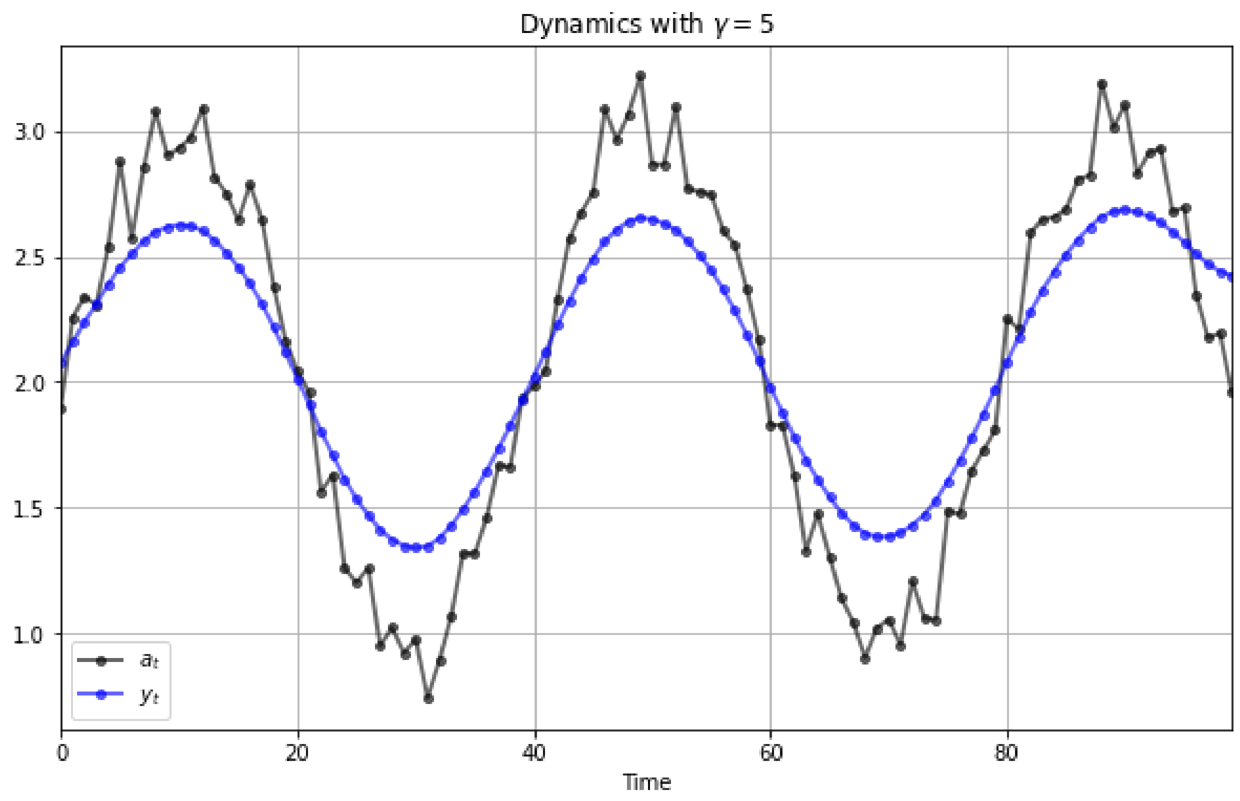
Here's some code that generates a plot when $\gamma = 0.8$

```python
# 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()
```



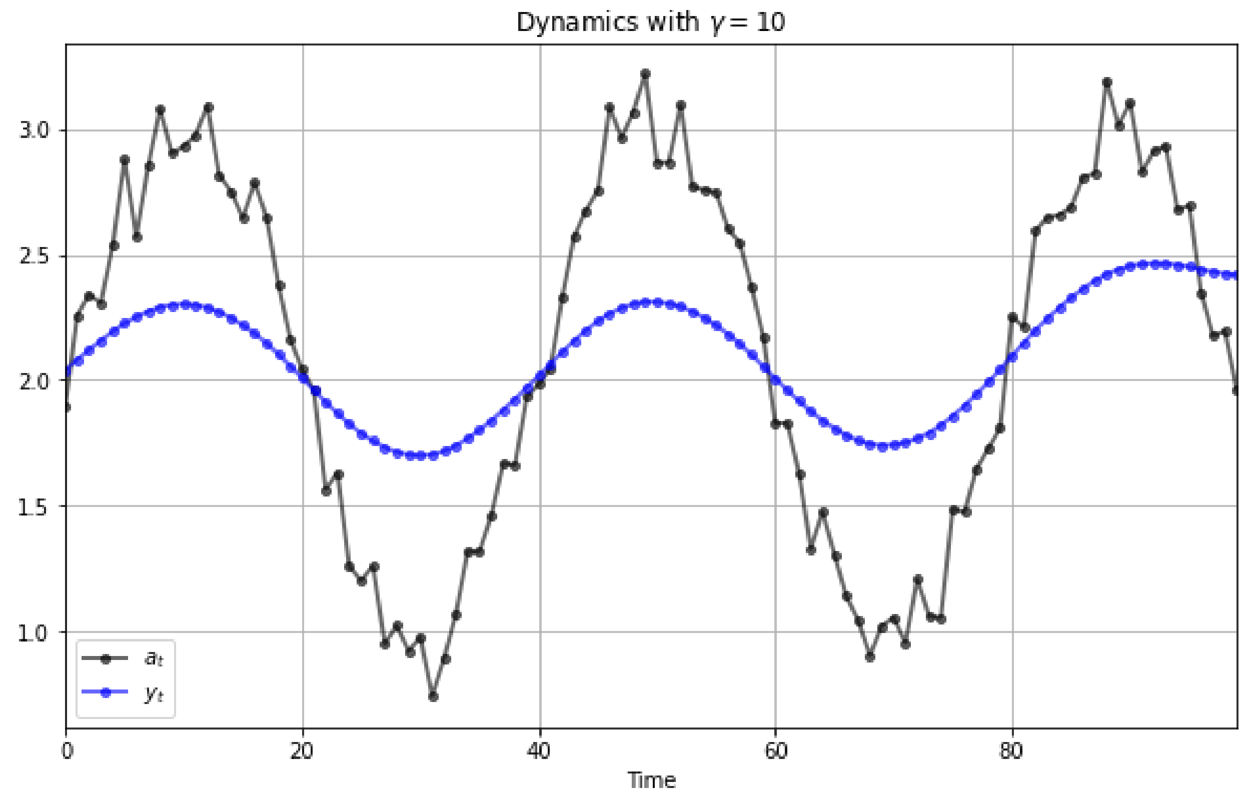
Here's what happens when we change γ to 5.0

```
plot_simulation(γ=5)
```



And here's $\gamma = 10$

```
plot_simulation(γ=10)
```



30.7 Exercises

30.7.1 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 $\lambda_j = \tilde{\lambda}_j \beta^{-1/2}$.

30.7.2 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.)

30.7.3 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

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

30.7.4 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 $y_t = .5y_{t-1}$.

CHAPTER

THIRTYONE

CLASSICAL PREDICTION AND FILTERING WITH LINEAR ALGEBRA

Contents

31.1 Overview

This is a sequel to the earlier lecture *Classical Control with Linear Algebra*.

That lecture used linear algebra – in particular, the LU decomposition – to formulate and solve a class of linear-quadratic optimal control problems.

In this lecture, we'll be using a closely related decomposition, the Cholesky decomposition, to solve linear prediction and filtering problems.

We exploit the useful fact that there is an intimate connection between two superficially different classes of problems:

- deterministic linear-quadratic (LQ) optimal control problems
- linear least squares prediction and filtering problems

The first class of problems involves no randomness, while the second is all about randomness.

Nevertheless, essentially the same mathematics solves both types of problem.

This connection, which is often termed "duality," is present whether one uses "classical" or "recursive" solution procedures.

In fact, we saw duality at work earlier when we formulated control and prediction problems recursively in lectures LQ dynamic programming problems, A first look at the Kalman filter, and The permanent income model.

A useful consequence of duality is that

- With every LQ control problem, there is implicitly affiliated a linear least squares prediction or filtering problem.
- With every linear least squares prediction or filtering problem there is implicitly affiliated a LQ control problem.

An understanding of these connections has repeatedly proved useful in cracking interesting applied problems.

For example, Sargent [Sar87] [chs. IX, XIV] and Hansen and Sargent [HS80] formulated and solved control and filtering problems using z-transform methods.

In this lecture, we begin to investigate these ideas by using mostly elementary linear algebra.

This is the main purpose and focus of the lecture.

However, after showing matrix algebra formulas, we'll summarize classic infinite-horizon formulas built on z-transform and lag operator methods.

And we'll occasionally refer to some of these formulas from the infinite dimensional problems as we present the finite time formulas and associated linear algebra.

We'll start with the following standard import:

```python
import numpy as np
```

31.1.1 References

Useful references include [Whi63], [HS80], [Orf88], [AP91], and [Mut60].

31.2 Finite Dimensional Prediction

Let $(x_1, x_2, \ldots, x_T)' = x$ be a $T \times 1$ vector of random variables with mean $\mathbb{E}x = 0$ and covariance matrix $\mathbb{E}xx' = V$.

Here V is a $T \times T$ positive definite matrix.

The i, j component $Ex_i x_j$ of V is the **inner product** between x_i and x_j.

We regard the random variables as being ordered in time so that x_t is thought of as the value of some economic variable at time t.

For example, x_t could be generated by the random process described by the Wold representation presented in equation (16) in the section below on infinite dimensional prediction and filtering.

In that case, V_{ij} is given by the coefficient on $z^{|i-j|}$ in the expansion of $g_x(z) = d(z)\, d(z^{-1}) + h$, which equals $h + \sum_{k=0}^{\infty} d_k d_{k+|i-j|}$.

We want to construct j step ahead linear least squares predictors of the form

$$\hat{\mathbb{E}}\left[x_T | x_{T-j}, x_{T-j+1}, \ldots, x_1\right]$$

where $\hat{\mathbb{E}}$ is the linear least squares projection operator.

(Sometimes $\hat{\mathbb{E}}$ is called the wide-sense expectations operator)

To find linear least squares predictors it is helpful first to construct a $T \times 1$ vector ε of random variables that form an orthonormal basis for the vector of random variables x.

The key insight here comes from noting that because the covariance matrix V of x is a positive definite and symmetric, there exists a (Cholesky) decomposition of V such that

$$V = L^{-1}(L^{-1})'$$

and

$$L\, V\, L' = I$$

where L and L^{-1} are both lower triangular.

Form the $T \times 1$ random vector $\varepsilon = Lx$.

The random vector ε is an orthonormal basis for x because

- L is nonsingular
- $\mathbb{E}\,\varepsilon\,\varepsilon' = L\mathbb{E}xx'L' = I$
- $x = L^{-1}\varepsilon$

It is enlightening to write out and interpret the equations $Lx = \varepsilon$ and $L^{-1}\varepsilon = x$.

First, we'll write $Lx = \varepsilon$

$$\begin{aligned} L_{11}x_1 &= \varepsilon_1 \\ L_{21}x_1 + L_{22}x_2 &= \varepsilon_2 \\ &\vdots \\ L_{T1}\,x_1 \,\ldots + L_{TT}x_T &= \varepsilon_T \end{aligned} \tag{1}$$

or

$$\sum_{j=0}^{t-1} L_{t,t-j}\,x_{t-j} = \varepsilon_t, \quad t = 1,\,2,\ldots T \tag{2}$$

Next, we write $L^{-1}\varepsilon = x$

$$\begin{aligned} x_1 &= L_{11}^{-1}\varepsilon_1 \\ x_2 &= L_{22}^{-1}\varepsilon_2 + L_{21}^{-1}\varepsilon_1 \\ &\vdots \\ x_T &= L_{TT}^{-1}\varepsilon_T + L_{T,T-1}^{-1}\varepsilon_{T-1}\,\ldots + L_{T,1}^{-1}\varepsilon_1 \end{aligned}, \tag{3}$$

or

$$x_t = \sum_{j=0}^{t-1} L_{t,t-j}^{-1}\,\varepsilon_{t-j} \tag{4}$$

where $L_{i,j}^{-1}$ denotes the i,j element of L^{-1}.

From (2), it follows that ε_t is in the linear subspace spanned by $x_t,\,x_{t-1},\ldots,\,x_1$.

From (4) it follows that that x_t is in the linear subspace spanned by $\varepsilon_t,\,\varepsilon_{t-1},\ldots,\varepsilon_1$.

Equation (2) forms a sequence of **autoregressions** that for $t = 1,\ldots,T$ express x_t as linear functions of $x_s, s = 1,\ldots,t-1$ and a random variable $(L_{t,t})^{-1}\varepsilon_t$ that is orthogonal to each componenent of $x_s, s = 1,\ldots,t-1$.

(Here $(L_{t,t})^{-1}$ denotes the reciprocal of $L_{t,t}$ while $L_{t,t}^{-1}$ denotes the t,t element of L^{-1}).

The equivalence of the subspaces spanned by $\varepsilon_t,\ldots,\varepsilon_1$ and $x_t,\ldots,x_1$ means that for $t-1 \geq m \geq 1$

$$\hat{\mathbb{E}}[x_t \mid x_{t-m},\,x_{t-m-1},\ldots,x_1] = \hat{\mathbb{E}}[x_t \mid \varepsilon_{t-m},\varepsilon_{t-m-1},\ldots,\varepsilon_1] \tag{5}$$

To proceed, it is useful to drill down and note that for $t-1 \geq m \geq 1$ we can rewrite (4) in the form of the **moving average representation**

$$x_t = \sum_{j=0}^{m-1} L_{t,t-j}^{-1}\,\varepsilon_{t-j} + \sum_{j=m}^{t-1} L_{t,t-j}^{-1}\,\varepsilon_{t-j} \tag{6}$$

Representation (6) is an orthogonal decomposition of x_t into a part $\sum_{j=m}^{t-1} L_{t,t-j}^{-1}\,\varepsilon_{t-j}$ that lies in the space spanned by $[x_{t-m},\, x_{t-m+1},\, \ldots, x_1]$ and an orthogonal component $\sum_{j=m}^{t-1} L_{t,t-j}^{-1}\,\varepsilon_{t-j}$ that does not lie in that space but instead in a linear space knowns as its **orthogonal complement**.

It follows that

$$\hat{\mathbb{E}}[x_t \mid x_{t-m},\, x_{t-m-1}, \ldots, x_1] = \sum_{j=0}^{m-1} L_{t,t-j}^{-1}\,\varepsilon_{t-j}$$

31.2.1 Implementation

Here's the code that computes solutions to LQ control and filtering problems using the methods described here and in *Classical Control with Linear Algebra*.

```python
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 ϕ 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) \
```

(continues on next page)

(continued from previous page)

```
                                                   @ 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 ϕ_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))
```

(continues on next page)

(continued from previous page)

```
        # (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
```

(continues on next page)

(continued from previous page)

```
        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()
```

(continues on next page)

(continued from previous page)

```
    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:
```

(continues on next page)

(continued from previous page)

```
                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
```

Let's use this code to tackle two interesting examples.

31.2.2 Example 1

Consider a stochastic process with moving average representation

$$x_t = (1 - 2L)\varepsilon_t$$

where ε_t is a serially uncorrelated random process with mean zero and variance unity.

If we were to use the tools associated with infinite dimensional prediction and filtering to be described below, we would use the Wiener-Kolmogorov formula (21) to compute the linear least squares forecasts $\mathbb{E}[x_{t+j} \mid x_t, x_{t-1}, ...]$, for $j = 1, 2$.

But we can do everything we want by instead using our finite dimensional tools and setting $d = r$, generating an instance of LQFilter, then invoking pertinent methods of LQFilter.

```
m = 1
y_m = np.asarray([.0]).reshape(m, 1)
d = np.asarray([1, -2])
```

(continues on next page)

(continued from previous page)

```
r = np.asarray([1, -2])
h = 0.0
example = LQFilter(d, h, y_m, r=d)
```

The Wold representation is computed by `example.coeffs_of_c()`.

Let's check that it "flips roots" as required

```
example.coeffs_of_c()
```

```
array([ 2., -1.])
```

```
example.roots_of_characteristic()
```

```
(array([2.]), -2.0, array([0.5]))
```

Now let's form the covariance matrix of a time series vector of length N and put it in V.

Then we'll take a Cholesky decomposition of $V = L^{-1}L^{-1}$ and use it to form the vector of "moving average representations" $x = L^{-1}\varepsilon$ and the vector of "autoregressive representations" $Lx = \varepsilon$.

```
V = example.construct_V(N=5)
print(V)
```

```
[[ 5. -2.  0.  0.  0.]
 [-2.  5. -2.  0.  0.]
 [ 0. -2.  5. -2.  0.]
 [ 0.  0. -2.  5. -2.]
 [ 0.  0.  0. -2.  5.]]
```

Notice how the lower rows of the "moving average representations" are converging to the appropriate infinite history Wold representation to be described below when we study infinite horizon-prediction and filtering

```
Li = np.linalg.cholesky(V)
print(Li)
```

```
[[ 2.23606798  0.          0.          0.          0.        ]
 [-0.89442719  2.04939015  0.          0.          0.        ]
 [ 0.         -0.97590007  2.01186954  0.          0.        ]
 [ 0.          0.         -0.99410024  2.00293902  0.        ]
 [ 0.          0.          0.         -0.99853265  2.000733  ]]
```

Notice how the lower rows of the "autoregressive representations" are converging to the appropriate infinite-history autoregressive representation to be described below when we study infinite horizon-prediction and filtering

```
L = np.linalg.inv(Li)
print(L)
```

```
[[0.4472136  0.         0.         0.         0.        ]
 [0.19518001 0.48795004 0.         0.         0.        ]
 [0.09467621 0.23669053 0.49705012 0.         0.        ]
 [0.04698977 0.11747443 0.2466963  0.49926632 0.        ]
 [0.02345182 0.05862954 0.12312203 0.24917554 0.49981682]]
```

31.2.3 Example 2

Consider a stochastic process X_t with moving average representation

$$X_t = (1 - \sqrt{2}L^2)\varepsilon_t$$

where ε_t is a serially uncorrelated random process with mean zero and variance unity.

Let's find a Wold moving average representation for x_t that will prevail in the infinite-history context to be studied in detail below.

To do this, we'll use the Wiener-Kolomogorov formula (21) presented below to compute the linear least squares forecasts $\hat{\mathbb{E}}\left[X_{t+j} \mid X_{t-1}, \ldots\right]$ for $j = 1, 2, 3$.

We proceed in the same way as in example 1

```
m = 2
y_m = np.asarray([.0, .0]).reshape(m, 1)
d = np.asarray([1, 0, -np.sqrt(2)])
r = np.asarray([1, 0, -np.sqrt(2)])
h = 0.0
example = LQFilter(d, h, y_m, r=d)
example.coeffs_of_c()
```

```
array([ 1.41421356, -0.        , -1.        ])
```

```
example.roots_of_characteristic()
```

```
(array([ 1.18920712, -1.18920712]),
 -1.4142135623731122,
 array([ 0.84089642, -0.84089642]))
```

```
V = example.construct_V(N=8)
print(V)
```

```
[[ 3.          0.         -1.41421356  0.          0.          0.
   0.          0.        ]
 [ 0.          3.          0.         -1.41421356  0.          0.
   0.          0.        ]
 [-1.41421356  0.          3.          0.         -1.41421356  0.
   0.          0.        ]
 [ 0.         -1.41421356  0.          3.          0.         -1.41421356
   0.          0.        ]
 [ 0.          0.         -1.41421356  0.          3.          0.
  -1.41421356  0.        ]
 [ 0.          0.          0.         -1.41421356  0.          3.
   0.         -1.41421356]
 [ 0.          0.          0.          0.         -1.41421356  0.
   3.          0.        ]
 [ 0.          0.          0.          0.          0.         -1.41421356
   0.          3.        ]]
```

```
Li = np.linalg.cholesky(V)
print(Li[-3:, :])
```

```
[[ 0.          0.          0.         -0.9258201   0.          1.46385011
   0.          0.        ]
 [ 0.          0.          0.          0.         -0.96609178  0.
   1.43759058  0.        ]
 [ 0.          0.          0.          0.          0.         -0.96609178
   0.          1.43759058]]
```

```python
L = np.linalg.inv(Li)
print(L)
```

```
[[0.57735027 0.         0.         0.         0.         0.
  0.         0.        ]
 [0.         0.57735027 0.         0.         0.         0.
  0.         0.        ]
 [0.3086067  0.         0.65465367 0.         0.         0.
  0.         0.        ]
 [0.         0.3086067  0.         0.65465367 0.         0.
  0.         0.        ]
 [0.19518001 0.         0.41403934 0.         0.68313005 0.
  0.         0.        ]
 [0.         0.19518001 0.         0.41403934 0.         0.68313005
  0.         0.        ]
 [0.13116517 0.         0.27824334 0.         0.45907809 0.
  0.69560834 0.        ]
 [0.         0.13116517 0.         0.27824334 0.         0.45907809
  0.         0.69560834]]
```

31.2.4 Prediction

It immediately follows from the "orthogonality principle" of least squares (see [AP91] or [Sar87] [ch. X]) that

$$\begin{aligned}\hat{\mathbb{E}}[x_t \mid x_{t-m}, x_{t-m+1}, \dots x_1] &= \sum_{j=m}^{t-1} L_{t,t-j}^{-1} \varepsilon_{t-j} \\ &= [L_{t,1}^{-1} \, L_{t,2}^{-1}, \dots, L_{t,t-m}^{-1} \; 0 \; 0 \dots 0] L \, x\end{aligned} \tag{7}$$

This can be interpreted as a finite-dimensional version of the Wiener-Kolmogorov m-step ahead prediction formula.

We can use (7) to represent the linear least squares projection of the vector x conditioned on the first s observations $[x_s, x_{s-1} \dots, x_1]$.

We have

$$\hat{\mathbb{E}}[x \mid x_s, x_{s-1}, \dots, x_1] = L^{-1} \begin{bmatrix} I_s & 0 \\ 0 & 0_{(t-s)} \end{bmatrix} Lx \tag{8}$$

This formula will be convenient in representing the solution of control problems under uncertainty.

Equation (4) can be recognized as a finite dimensional version of a moving average representation.

Equation (2) can be viewed as a finite dimension version of an autoregressive representation.

Notice that even if the x_t process is covariance stationary, so that V is such that V_{ij} depends only on $|i-j|$, the coefficients in the moving average representation are time-dependent, there being a different moving average for each t.

If x_t is a covariance stationary process, the last row of L^{-1} converges to the coefficients in the Wold moving average representation for $\{x_t\}$ as $T \to \infty$.

Further, if x_t is covariance stationary, for fixed k and $j > 0$, $L^{-1}_{T,T-j}$ converges to $L^{-1}_{T-k,T-k-j}$ as $T \to \infty$.

That is, the "bottom" rows of L^{-1} converge to each other and to the Wold moving average coefficients as $T \to \infty$.

This last observation gives one simple and widely-used practical way of forming a finite T approximation to a Wold moving average representation.

First, form the covariance matrix $\mathbb{E}xx' = V$, then obtain the Cholesky decomposition $L^{-1}L^{-1'}$ of V, which can be accomplished quickly on a computer.

The last row of L^{-1} gives the approximate Wold moving average coefficients.

This method can readily be generalized to multivariate systems.

31.3 Combined Finite Dimensional Control and Prediction

Consider the finite-dimensional control problem, maximize

$$\mathbb{E} \sum_{t=0}^{N} \left\{ a_t y_t - \frac{1}{2} h y_t^2 - \frac{1}{2} [d(L) y_t]^2 \right\}, \qquad h > 0$$

where $d(L) = d_0 + d_1 L + \ldots + d_m L^m$, L is the lag operator, $\bar{a} = [a_N, a_{N-1} \ldots, a_1, a_0]'$ a random vector with mean zero and $\mathbb{E}\, \bar{a}\bar{a}' = V$.

The variables $y_{-1}, \ldots, y_{-m}$ are given.

Maximization is over choices of $y_0, y_1 \ldots, y_N$, where y_t is required to be a linear function of $\{y_{t-s-1}, t + m - 1 \geq 0;\ a_{t-s}, t \geq s \geq 0\}$.

We saw in the lecture *Classical Control with Linear Algebra* that the solution of this problem under certainty could be represented in the feedback-feedforward form

$$U\bar{y} = L^{-1}\bar{a} + K \begin{bmatrix} y_{-1} \\ \vdots \\ y_{-m} \end{bmatrix}$$

for some $(N + 1) \times m$ matrix K.

Using a version of formula (7), we can express $\hat{\mathbb{E}}[\bar{a} \mid a_s,\ a_{s-1}, \ldots, a_0]$ as

$$\hat{\mathbb{E}}[\bar{a} \mid a_s,\ a_{s-1}, \ldots, a_0] = \tilde{U}^{-1} \begin{bmatrix} 0 & 0 \\ 0 & I_{(s+1)} \end{bmatrix} \tilde{U}\bar{a}$$

where $I_{(s+1)}$ is the $(s+1) \times (s+1)$ identity matrix, and $V = \tilde{U}^{-1}\tilde{U}^{-1'}$, where $\tilde{U}$ is the *upper* triangular Cholesky factor of the covariance matrix V.

(We have reversed the time axis in dating the a's relative to earlier)

The time axis can be reversed in representation (8) by replacing L with L^T.

The optimal decision rule to use at time $0 \leq t \leq N$ is then given by the $(N - t + 1)^{\text{th}}$ row of

$$U\bar{y} = L^{-1}\tilde{U}^{-1} \begin{bmatrix} 0 & 0 \\ 0 & I_{(t+1)} \end{bmatrix} \tilde{U}\bar{a} + K \begin{bmatrix} y_{-1} \\ \vdots \\ y_{-m} \end{bmatrix}$$

31.4 Infinite Horizon Prediction and Filtering Problems

It is instructive to compare the finite-horizon formulas based on linear algebra decompositions of finite-dimensional covariance matrices with classic formulas for infinite horizon and infinite history prediction and control problems.

These classic infinite horizon formulas used the mathematics of z-transforms and lag operators.

We'll meet interesting lag operator and z-transform counterparts to our finite horizon matrix formulas.

We pose two related prediction and filtering problems.

We let Y_t be a univariate m^{th} order moving average, covariance stationary stochastic process,

$$Y_t = d(L)u_t \tag{9}$$

where $d(L) = \sum_{j=0}^{m} d_j L^j$, and u_t is a serially uncorrelated stationary random process satisfying

$$\begin{aligned} \mathbb{E}u_t &= 0 \\ \mathbb{E}u_t u_s &= \begin{cases} 1 & \text{if } t = s \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{10}$$

We impose no conditions on the zeros of $d(z)$.

A second covariance stationary process is X_t given by

$$X_t = Y_t + \varepsilon_t \tag{11}$$

where ε_t is a serially uncorrelated stationary random process with $\mathbb{E}\varepsilon_t = 0$ and $\mathbb{E}\varepsilon_t \varepsilon_s$ = 0 for all distinct t and s.

We also assume that $\mathbb{E}\varepsilon_t u_s = 0$ for all t and s.

The **linear least squares prediction problem** is to find the L_2 random variable $\hat{X}_{t+j}$ among linear combinations of $\{X_t,\ X_{t-1}, ...\}$ that minimizes $\mathbb{E}(\hat{X}_{t+j} - X_{t+j})^2$.

That is, the problem is to find a $\gamma_j(L) = \sum_{k=0}^{\infty} \gamma_{jk} L^k$ such that $\sum_{k=0}^{\infty} |\gamma_{jk}|^2 < \infty$ and $\mathbb{E}[\gamma_j(L)X_t - X_{t+j}]^2$ is minimized.

The **linear least squares filtering problem** is to find a $b(L) = \sum_{j=0}^{\infty} b_j L^j$ such that $\sum_{j=0}^{\infty} |b_j|^2 < \infty$ and $\mathbb{E}[b(L)X_t - Y_t]^2$ is minimized.

Interesting versions of these problems related to the permanent income theory were studied by [Mut60].

31.4.1 Problem Formulation

These problems are solved as follows.

The covariograms of Y and X and their cross covariogram are, respectively,

$$\begin{aligned} C_X(\tau) &= \mathbb{E}X_t X_{t-\tau} \\ C_Y(\tau) &= \mathbb{E}Y_t Y_{t-\tau} \qquad \tau = 0, \pm 1, \pm 2, ... \\ C_{Y,X}(\tau) &= \mathbb{E}Y_t X_{t-\tau} \end{aligned} \tag{12}$$

The covariance and cross-covariance generating functions are defined as

$$
\begin{aligned}
g_X(z) &= \sum_{\tau=-\infty}^{\infty} C_X(\tau) z^{\tau} \\
g_Y(z) &= \sum_{\tau=-\infty}^{\infty} C_Y(\tau) z^{\tau} \\
g_{YX}(z) &= \sum_{\tau=-\infty}^{\infty} C_{YX}(\tau) z^{\tau}
\end{aligned}
\tag{13}
$$

The generating functions can be computed by using the following facts.

Let v_{1t} and v_{2t} be two mutually and serially uncorrelated white noises with unit variances.

That is, $\mathbb{E}v_{1t}^2 = \mathbb{E}v_{2t}^2 = 1, \mathbb{E}v_{1t} = \mathbb{E}v_{2t} = 0, \mathbb{E}v_{1t}v_{2s} = 0$ for all t and s, $\mathbb{E}v_{1t}v_{1t-j} = \mathbb{E}v_{2t}v_{2t-j} = 0$ for all $j \neq 0$.

Let x_t and y_t be two random processes given by

$$
\begin{aligned}
y_t &= A(L)v_{1t} + B(L)v_{2t} \\
x_t &= C(L)v_{1t} + D(L)v_{2t}
\end{aligned}
$$

Then, as shown for example in [Sar87] [ch. XI], it is true that

$$
\begin{aligned}
g_y(z) &= A(z)A(z^{-1}) + B(z)B(z^{-1}) \\
g_x(z) &= C(z)C(z^{-1}) + D(z)D(z^{-1}) \\
g_{yx}(z) &= A(z)C(z^{-1}) + B(z)D(z^{-1})
\end{aligned}
\tag{14}
$$

Applying these formulas to (9) – (12), we have

$$
\begin{aligned}
g_Y(z) &= d(z)d(z^{-1}) \\
g_X(z) &= d(z)d(z^{-1}) + h \\
g_{YX}(z) &= d(z)d(z^{-1})
\end{aligned}
\tag{15}
$$

The key step in obtaining solutions to our problems is to factor the covariance generating function $g_X(z)$ of X.

The solutions of our problems are given by formulas due to Wiener and Kolmogorov.

These formulas utilize the Wold moving average representation of the X_t process,

$$
X_t = c\,(L)\,\eta_t \tag{16}
$$

where $c(L) = \sum_{j=0}^{m} c_j\, L^j$, with

$$
c_0\eta_t = X_t - \hat{\mathbb{E}}[X_t | X_{t-1}, X_{t-2}, ...] \tag{17}
$$

Here $\hat{\mathbb{E}}$ is the linear least squares projection operator.

Equation (17) is the condition that $c_0\eta_t$ can be the one-step-ahead error in predicting X_t from its own past values.

Condition (17) requires that η_t lie in the closed linear space spanned by $[X_t,\ X_{t-1}, ...]$.

This will be true if and only if the zeros of $c(z)$ do not lie inside the unit circle.

It is an implication of (17) that η_t is a serially uncorrelated random process and that normalization can be imposed so that $\mathbb{E}\eta_t^2 = 1$.

Consequently, an implication of (16) is that the covariance generating function of X_t can be expressed as

$$
g_X(z) = c\,(z)\,c\,(z^{-1}) \tag{18}
$$

It remains to discuss how $c(L)$ is to be computed.

Combining (14) and (18) gives

$$d(z)\,d(z^{-1}) + h = c\,(z)\,c\,(z^{-1}) \tag{19}$$

Therefore, we have already shown constructively how to factor the covariance generating function $g_X(z) = d(z)\,d\,(z^{-1}) + h$.

We now introduce the **annihilation operator**:

$$\left[\sum_{j=-\infty}^{\infty} f_j\, L^j\right]_+ \equiv \sum_{j=0}^{\infty} f_j\, L^j \tag{20}$$

In words, $[\quad]_+$ means "ignore negative powers of L".

We have defined the solution of the prediction problem as $\hat{\mathbb{E}}[X_{t+j}|X_t,\, X_{t-1}, ...] = \gamma_j\,(L)X_t$.

Assuming that the roots of $c(z) = 0$ all lie outside the unit circle, the Wiener-Kolmogorov formula for $\gamma_j(L)$ holds:

$$\gamma_j\,(L) = \left[\frac{c(L)}{L^j}\right]_+ c\,(L)^{-1} \tag{21}$$

We have defined the solution of the filtering problem as $\hat{\mathbb{E}}[Y_t \mid X_t, X_{t-1}, ...] = b(L)X_t$.

The Wiener-Kolomogorov formula for $b(L)$ is

$$b(L) = \left[\frac{g_{YX}(L)}{c(L^{-1})}\right]_+ c(L)^{-1}$$

or

$$b(L) = \left[\frac{d(L)d(L^{-1})}{c(L^{-1})}\right]_+ c(L)^{-1} \tag{22}$$

Formulas (21) and (22) are discussed in detail in [Whi83] and [Sar87].

The interested reader can there find several examples of the use of these formulas in economics Some classic examples using these formulas are due to [Mut60].

As an example of the usefulness of formula (22), we let X_t be a stochastic process with Wold moving average representation

$$X_t = c(L)\eta_t$$

where $\mathbb{E}\eta_t^2 = 1,$ and $c_0\eta_t = X_t - \hat{\mathbb{E}}[X_t|X_{t-1}, ...], c(L) = \sum_{j=0}^m c_j L.$

Suppose that at time t, we wish to predict a geometric sum of future X's, namely

$$y_t \equiv \sum_{j=0}^{\infty} \delta^j X_{t+j} = \frac{1}{1-\delta L^{-1}} X_t$$

given knowledge of $X_t, X_{t-1}, ...$

We shall use (22) to obtain the answer.

Using the standard formulas (14), we have that

$$\begin{aligned} g_{yx}(z) &= (1-\delta z^{-1})c(z)c(z^{-1}) \\ g_x(z) &= c(z)c(z^{-1}) \end{aligned}$$

Then (22) becomes

$$b(L) = \left[\frac{c(L)}{1-\delta L^{-1}}\right]_+ c(L)^{-1} \tag{23}$$

In order to evaluate the term in the annihilation operator, we use the following result from [HS80].

Proposition Let

- $g(z) = \sum_{j=0}^{\infty} g_j\, z^j$ where $\sum_{j=0}^{\infty} |g_j|^2 < +\infty$.
- $h\left(z^{-1}\right) = (1-\delta_1 z^{-1}) \ldots (1-\delta_n z^{-1})$, where $|\delta_j| < 1$, for $j = 1, \ldots, n$.

Then

$$\left[\frac{g(z)}{h(z^{-1})}\right]_+ = \frac{g(z)}{h(z^{-1})} - \sum_{j=1}^{n} \frac{\delta_j g(\delta_j)}{\prod_{\substack{k=1\\k\neq j}}^{n}(\delta_j - \delta_k)} \left(\frac{1}{z-\delta_j}\right) \tag{24}$$

and, alternatively,

$$\left[\frac{g(z)}{h(z^{-1})}\right]_+ = \sum_{j=1}^{n} B_j \left(\frac{zg(z) - \delta_j g(\delta_j)}{z-\delta_j}\right) \tag{25}$$

where $B_j = 1/\prod_{\substack{k=1\\k+j}}^{n}(1-\delta_k/\delta_j)$.

Applying formula (25) of the proposition to evaluating (23) with $g(z) = c(z)$ and $h(z^{-1}) = 1 - \delta z^{-1}$ gives

$$b(L) = \left[\frac{Lc(L) - \delta c(\delta)}{L-\delta}\right] c(L)^{-1}$$

or

$$b(L) = \left[\frac{1 - \delta c(\delta) L^{-1} c(L)^{-1}}{1-\delta L^{-1}}\right]$$

Thus, we have

$$\hat{\mathbb{E}}\left[\sum_{j=0}^{\infty} \delta^j X_{t+j} | X_t,\, x_{t-1}, \ldots\right] = \left[\frac{1 - \delta c(\delta) L^{-1} c(L)^{-1}}{1-\delta L^{-1}}\right] X_t \tag{26}$$

This formula is useful in solving stochastic versions of problem 1 of lecture *Classical Control with Linear Algebra* in which the randomness emerges because $\{a_t\}$ is a stochastic process.

The problem is to maximize

$$\mathbb{E}_0 \lim_{N\to\infty} \sum_{t-0}^{N} \beta^t \left[a_t\, y_t - \frac{1}{2}\, h y_t^2 - \frac{1}{2}\, [d(L) y_t]^2\right] \tag{27}$$

where $\mathbb{E}_t$ is mathematical expectation conditioned on information known at t, and where $\{a_t\}$ is a covariance stationary stochastic process with Wold moving average representation

$$a_t = c(L)\, \eta_t$$

where

$$c(L) = \sum_{j=0}^{\tilde{n}} c_j L^j$$

and

$$\eta_t = a_t - \hat{\mathbb{E}}[a_t | a_{t-1}, ...]$$

The problem is to maximize (27) with respect to a contingency plan expressing y_t as a function of information known at t, which is assumed to be $(y_{t-1},\ y_{t-2}, \dots, a_t,\ a_{t-1}, \dots)$.

The solution of this problem can be achieved in two steps.

First, ignoring the uncertainty, we can solve the problem assuming that $\{a_t\}$ is a known sequence.

The solution is, from above,

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

or

$$(1 - \lambda_1 L) \dots (1 - \lambda_m L) y_t = \sum_{j=1}^m A_j \sum_{k=0}^\infty (\lambda_j \beta)^k \, a_{t+k} \tag{28}$$

Second, the solution of the problem under uncertainty is obtained by replacing the terms on the right-hand side of the above expressions with their linear least squares predictors.

Using (26) and (28), we have the following solution

$$(1 - \lambda_1 L) \dots (1 - \lambda_m L) y_t = \sum_{j=1}^m A_j \left[\frac{1 - \beta \lambda_j \, c(\beta \lambda_j) L^{-1} c(L)^{-1}}{1 - \beta \lambda_j L^{-1}} \right] a_t$$

Blaschke factors

The following is a useful piece of mathematics underlying "root flipping".

Let $\pi(z) = \sum_{j=0}^m \pi_j z^j$ and let $z_1, \dots, z_k$ be the zeros of $\pi(z)$ that are inside the unit circle, $k < m$.

Then define

$$\theta(z) = \pi(z) \Biggl(\frac{(z_1 z - 1)}{(z - z_1)} \Biggr) \Biggl(\frac{(z_2 z - 1)}{(z - z_2)} \Biggr) \dots \Biggl(\frac{(z_k z - 1)}{(z - z_k)} \Biggr)$$

The term multiplying $\pi(z)$ is termed a "Blaschke factor".

Then it can be proved directly that

$$\theta(z^{-1}) \theta(z) = \pi(z^{-1}) \pi(z)$$

and that the zeros of $\theta(z)$ are not inside the unit circle.

31.5 Exercises

31.5.1 Exercise 1

Let $Y_t = (1 - 2L) u_t$ where u_t is a mean zero white noise with $\mathbb{E} u_t^2 = 1$. Let

$$X_t = Y_t + \varepsilon_t$$

where ε_t is a serially uncorrelated white noise with $\mathbb{E} \varepsilon_t^2 = 9$, and $\mathbb{E} \varepsilon_t u_s = 0$ for all t and s.

Find the Wold moving average representation for X_t.

Find a formula for the A_{1j}'s in

$$\mathbb{E}\widehat{X}_{t+1} \mid X_t, X_{t-1}, \ldots = \sum_{j=0}^{\infty} A_{1j} X_{t-j}$$

Find a formula for the A_{2j}'s in

$$\hat{\mathbb{E}} X_{t+2} \mid X_t, X_{t-1}, \ldots = \sum_{j=0}^{\infty} A_{2j} X_{t-j}$$

31.5.2 Exercise 2

Multivariable Prediction: Let Y_t be an $(n \times 1)$ vector stochastic process with moving average representation

$$Y_t = D(L) U_t$$

where $D(L) = \sum_{j=0}^{m} D_j L^J$, D_j an $n \times n$ matrix, U_t an $(n \times 1)$ vector white noise with $\mathbb{E}U_t = 0$ for all t, $\mathbb{E}U_t U_s' = 0$ for all $s \neq t$, and $\mathbb{E}U_t U_t' = I$ for all t.

Let ε_t be an $n \times 1$ vector white noise with mean 0 and contemporaneous covariance matrix H, where H is a positive definite matrix.

Let $X_t = Y_t + \varepsilon_t$.

Define the covariograms as $C_X(\tau) = \mathbb{E}X_t X_{t-\tau}'$, $C_Y(\tau) = \mathbb{E}Y_t Y_{t-\tau}'$, $C_{YX}(\tau) = \mathbb{E}Y_t X_{t-\tau}'$.

Then define the matrix covariance generating function, as in (21), only interpret all the objects in (21) as matrices.

Show that the covariance generating functions are given by

$$\begin{aligned} g_y(z) &= D(z) D(z^{-1})' \\ g_X(z) &= D(z) D(z^{-1})' + H \\ g_{YX}(z) &= D(z) D(z^{-1})' \end{aligned}$$

A factorization of $g_X(z)$ can be found (see [Roz67] or [Whi83]) of the form

$$D(z) D(z^{-1})' + H = C(z) C(z^{-1})', \quad C(z) = \sum_{j=0}^{m} C_j z^j$$

where the zeros of $|C(z)|$ do not lie inside the unit circle.

A vector Wold moving average representation of X_t is then

$$X_t = C(L) \eta_t$$

where η_t is an $(n \times 1)$ vector white noise that is "fundamental" for X_t.

That is, $X_t - \hat{\mathbb{E}} [X_t \mid X_{t-1}, X_{t-2} \ldots] = C_0 \, \eta_t$.

The optimum predictor of X_{t+j} is

$$\hat{\mathbb{E}} \left[X_{t+j} \mid X_t, X_{t-1}, \ldots \right] = \left[\frac{C(L)}{L^j} \right]_+ \eta_t$$

If $C(L)$ is invertible, i.e., if the zeros of det $C(z)$ lie strictly outside the unit circle, then this formula can be written

$$\hat{\mathbb{E}} \left[X_{t+j} \mid X_t, X_{t-1}, \ldots \right] = \left[\frac{C(L)}{L^J} \right]_+ C(L)^{-1} \, X_t$$

CHAPTER

THIRTYTWO

KNOWING THE FORECASTS OF OTHERS

Contents

In addition to what's in Anaconda, this lecture will need the following libraries:

```
!pip install --upgrade quantecon
!conda install -y -c plotly plotly plotly-orca
```

32.1 Introduction

Robert E. Lucas, Jr. [REL75], Kenneth Kasa [Kas00], and Robert Townsend [Tow83] showed that giving decision makers incentives to infer persistent hidden state variables from equilibrium prices and quantities can elongate and amplify impulse responses to aggregate shocks in business cycle models.

Townsend [Tow83] noted that such incentives can naturally induce decision makers to want to forecast the forecast of others.

This theme has been pursued and extended in analyses in which decision makers' imperfect information forces them into pursuing an infinite recursion of forming beliefs about the beliefs of other (e.g., [AMS02]).

Lucas [REL75] side stepped having decision makers forecast the forecasts of other decision makers by assuming that they simply pool their information before forecasting.

A **pooling equilibrium** like Lucas's plays a prominent role in this lecture.

Because he didn't assume such pooling, [Tow83] confronted the forecasting the forecasts of others problem.

To formulate the problem recursively required that Townsend define decision maker's **state** vector.

Townsend concluded that his original model required an intractable infinite dimensional state space.

Therefore, he constructed a more manageable approximating model in which the hidden Markov component of the demand shock is revealed to all firms after a fixed and finite number of periods.

In this lecture, as yet another instance of the theme that **finding the state is an art**, we show how to formulate Townsend's original model in terms of a low-dimensional state space.

By doing so, we show that Townsend's model shares equilibrium prices and quantities with those that prevail in a pooling equilibrium.

That finding emerged from a line of research about Townsend's model that culminated in [PS05] that built on [PCL86].

However, rather than deploying the [PCL86] machinery here, we shall rely instead on a sneaky **guess-and-verify** tactic.

- We compute a pooling equilibrium and represent it as an instance of a linear state-space system provided by the Python class `quantecon.LinearStateSpace`.
- Leaving the state-transition equation for the pooling equilibrium unaltered, we alter the observation vector for a firm to what it in in Townsend's original model. So rather than directly observing the signal received by firms in the other industry, a firm sees the equilibrium price of the good produced by the other industry.
- We compute a population linear least squares regression of the noisy signal that firms in the other industry receive in a pooling equilibrium on time t information that a firm receives in Townsend's original model. The R^2 in this regression equals 1. That verifies that a firm's information set in Townsend's original model equals its information set in a pooling equilibrium. Therefore, equilibrium prices and quantities in Townsend's original model equal those in a pooling equilibrium.

32.1.1 A Sequence of Models

We proceed by describing a sequence of models of two industries that are linked in a single way: shocks to the demand curves for their products have a common component.

The models are simplified versions of Townsend's [Tow83].

Townsend's is a model of a rational expectations equilibrium in which firms confront the problem **forecasting the forecasts of others**.

In Townsend's model, firms condition their forecasts on observed endogenous variables whose equilibrium laws of motion are determined by their own forecasting functions.

We start with model components that we shall progressively assemble in ways that can help us to appreciate the structure of a **pooling equilibrium** that ultimately concerns us.

While keeping other aspects of the model the same, we shall study consequences of alternative assumptions about what decision makers observe.

Technically, this lecture deploys concepts and tools that appear in First Look at Kalman Filter and Rational Expectations Equilibrium.

32.2 The Setting

We cast all variables in terms of deviations from means.

Therefore, we omit constants from inverse demand curves and other functions.

Firms in each of two industries $i = 1, 2$ use a single factor of production, capital k_t^i, to produce output of a single good, y_t^i.

Firms bear quadratic costs of adjusting their capital stocks.

A representative firm in industry i has production function $y_t^i = f k_t^i$, $f > 0$, acts as a price taker with respect to output price P_t^i, and maximizes

$$E_0^i \sum_{t=0}^{\infty} \beta^t \left\{ P_t^i f k_t^i - .5h(k_{t+1}^i - k_t^i)^2 \right\}, \quad h > 0. \tag{1}$$

Demand in industry i is described by the inverse demand curve

$$P_t^i = -bY_t^i + \theta_t + \epsilon_t^i, \quad b > 0, \tag{2}$$

where P_t^i is the price of good i at t, $Y_t^i = fK_t^i$ is output in market i, θ_t is a persistent component of a demand shock that is common across the two industries, and ϵ_t^i is an industry specific component of the demand shock that is i.i.d. and whose time t marginal distributon is $\mathcal{N}(0, \sigma_\epsilon^2)$.

We assume that θ_t is governed by

$$\theta_{t+1} = \rho\theta_t + v_t \tag{3}$$

where $\{v_t\}$ is an i.i.d. sequence of Gaussian shocks each with mean zero and variance σ_v^2.

To simplify notation, we'll study a special case of the model by setting $h = f = 1$.

The presence of costs of adjusting their capital stocks imparts to firms an incentives to forecast the price of the good that they sell.

Throughout, we use the **rational expectations** equilibrium concept presented in this lecture Rational Expectations Equilibrium.

We let capital letters denote market wide objects and lower case letters denote objects chosen by a representative firm.

In each industry, a competitive equilibrium prevails.

To rationalize the big K, little k connection, we can think of there being a continua of each type of firm, each indexed by $\omega \in [0, 1]$ with $K^i = \int_0^1 k^i(\omega)d\omega$.

In equilibrium, $k_t^i = K_t^i$, but as usual we must distinguish between k_t^i and K_t^i when we pose the firm's optimization problem.

32.3 Tactics

We shall compute equilibrium laws of motion for capital in industry i under a sequence of assumptions about what a representative firm observes.

Successive members of this sequence make a representative firm's information more and more obscure.

We begin with the most information, then gradually withdraw information in a way that approaches and eventually reaches the information structure that that we are ultimately interested in.

Thus, we shall compute equilibria under the following alternative information structures:

- **Perfect foresight:** future values of θ_t, ϵ_t^i are observed in industry i.
- **Observed but stochastic** θ_t: $\{\theta_t, \epsilon_t^i\}$ are realizations from a stochastic process; current and past values of each are observed at time t but future values are not.
- **One noise-ridden observation on** θ_t: Values of $\{\theta_t, \epsilon_t^i\}$ separately are never observed. However, at time t, a history w^t of a scalar noise-ridden observations on θ_t is observed at time t.
- **Two noise-ridden observations on** θ_t: Values of $\{\theta_t, \epsilon_t^i\}$ separately are never observed. However, at time t, a history w^t of *two* noise-ridden observations on θ_t is observed at time t.

Successive computations build one on another.

We proceed by first finding an equilibrium under perfect foresight.

To compute an equilibrium with θ_t observed, we use a *certainty equivalence principle* to justify modifying the perfect foresight equilibrium by replacing future values of $\theta_s, \epsilon_s^i, s \geq t$ with mathematical expectations conditioned on θ_t.

This provides the equilibrium when θ_t is observed at t but future θ_{t+j} and ϵ_{t+j}^i are not observed.

To find an equilibrium when only a history w_t of a single noise ridden observations on θ_t is observed, we again apply a certainty equivalence principle and replace future values of the random variables $\theta_s, \epsilon_s^i, s \geq t$ with their mathematical expectations conditioned on w^t.

To find an equilibrium when only a history w_t of a *two* noisy signal on θ_t is observed, we replace future values of the random variables $\theta_s, \epsilon_s^i, s \geq t$ with their mathematical expectations conditioned on history w^t.

We call the equilibrium with two noise-ridden observations on θ_t a **pooling equilibrium**.

- It corresponds to an arrangement in which at the beginning of each period firms in industries 1 and 2 somehow get together and share information about current values of their noisy signals on θ.

We want ultimately to compare outcomes in a pooling equilibrium with an equilibrium under the following alternative information structure for a firm in industry i that interested [Tow83]:

- **Firm** i's **noise-ridden signal on** θ_t **and the price in industry** $-i$, a firm in industry i observes a history w^t of *one* noise-ridden signal on θ_t and a history of industry $-i$'s price is observed.

With this information structure, the representative firm i sees the price as well as the aggregate state variable Y_t^i in its own industry.

That allows it to infer the total demand shock $\theta_t + \epsilon_t^i$.

However, at time t, the firm sees only P_t^{-i} and does not see Y_t^{-i}, so that firm i does not directly observe $\theta_t + \epsilon_t^{-i}$.

Nevertheless, it will turn out that equilibrium prices and quantities in this equilibrium equal their counterparts in a pooling equilibrium because firms in industry i are able to infer the noisy signal about the demand shock received by firms in industry $-i$.

We shall eventually verify this assertion by using a guess and verify tactic.[1]

[1] [PS05] verified this assertion using a different tactic, namely, by constructing analytic formulas an equilibrium under the incomplete information structure and confirming that they match the pooling equilibrium formulas derived here.

32.4 Equilibrium conditions

It is convenient to solve the firm's problem without uncertainty by forming the Lagrangian:

$$J = \sum_{t=0}^{\infty} \beta^t \left\{ P_t^i k_t^i - .5(\mu_t^i)^2 + \phi_t^i \left[k_t^i + \mu_t^i - k_{t+1}^i \right] \right\}$$

where $\{\phi_t^i\}$ is a sequence of Lagrange multipliers on the transition law for k_{t+1}^i. First order conditions for the nonstochastic problem are

$$\begin{aligned} \phi_t^i &= \beta \phi_{t+1}^i + \beta P_{t+1}^i \\ \mu_t^i &= \phi_t^i. \end{aligned} \tag{4}$$

Substituting the demand function (2) for P_t^i, imposing the condition that the representative firm is representative ($k_t^i = K_t^i$), and using the definition below of g_t^i, the Euler equation (4), lagged by one period, can be expressed as $-bk_t^i + \theta_t + \epsilon_t^i + (k_{t+1}^i - k_t^i) - g_t^i = 0$ or

$$k_{t+1}^i = (b+1)k_t^i - \theta_t - \epsilon_t^i + g_t^i \tag{5}$$

where we define g_t^i by

$$g_t^i = \beta^{-1}(k_t^i - k_{t-1}^i) \tag{6}$$

We can write Euler equation (4) as:

$$g_t^i = P_t^i + \beta g_{t+1}^i \tag{7}$$

In addition, we have the law of motion for θ_t, (3), and the demand equation (2).

In summary, with perfect foresight, equilibrium conditions for industry i include the following system of difference equations:

$$\begin{aligned} k_{t+1}^i &= (1+b)k_t^i - \epsilon_t^i - \theta_t + g_t^i \\ \theta_{t+1} &= \rho \theta_t + v_t \\ g_{t+1}^i &= \beta^{-1}(g_t^i - P_t^i) \\ P_t^i &= -bk_t^i + \epsilon_t^i + \theta_t \end{aligned} \tag{8}$$

Without perfect foresight, the same system prevails except that the following equation replaces the third equation of (8):

$$g_{t+1,t}^i = \beta^{-1}(g_t^i - P_t^i)$$

where $x_{t+1,t}$ denotes the mathematical expectation of x_{t+1} conditional on information at time t.

32.4.1 Equilibrium under perfect foresight

Our first step is to compute the equilibrium law of motion for k_t^i under perfect foresight.

Let L be the lag operator.[2]

Equations (7) and (5) imply the second order difference equation in k_t^i:[3]

$$\left[(L^{-1} - (1+b))(1 - \beta L^{-1}) + b \right] k_t^i = \beta L^{-1} \epsilon_t^i + \beta L^{-1} \theta_t. \tag{9}$$

[2] See [Sar87], especially chapters IX and XIV, for the principles that guide solving some roots backwards and others forwards.

[3] As noted [Sar87], this difference equation is the Euler equation for the planning problem of maximizing the discounted sum of consumer plus producer surplus.

Factor the polynomial in L on the left side as:

$$-\beta[L^{-2} - (\beta^{-1} + (1+b))L^{-1} + \beta^{-1}] = \tilde{\lambda}^{-1}(L^{-1} - \tilde{\lambda})(1 - \tilde{\lambda}\beta L^{-1})$$

where $|\tilde{\lambda}| < 1$ is the smaller root and λ is the larger root of $(\lambda - 1)(\lambda - 1/\beta) = b\lambda$.

Therefore, (9) can be expressed as

$$\tilde{\lambda}^{-1}(L^{-1} - \tilde{\lambda})(1 - \tilde{\lambda}\beta L^{-1})k_t^i = \beta L^{-1}\epsilon_t^i + \beta L^{-1}\theta_t.$$

Solving the stable root backwards and the unstable root forwards gives

$$k_{t+1}^i = \tilde{\lambda}k_t^i + \frac{\tilde{\lambda}\beta}{1 - \tilde{\lambda}\beta L^{-1}}(\epsilon_{t+1}^i + \theta_{t+1}).$$

Recall that we have already set $k^i = K^i$ at the appropriate point in the argument (i.e., *after* having derived the first-order necessary conditions for a representative firm in industry i.

Thus, under perfect foresight the equilibrium capital stock in industry i satisfies

$$k_{t+1}^i = \tilde{\lambda}k_t^i + \sum_{j=1}^{\infty}(\tilde{\lambda}\beta)^j(\epsilon_{t+j}^i + \theta_{t+j}). \tag{10}$$

Next, we shall investigate consequences of replacing future values of $(\epsilon_{t+j}^i + \theta_{t+j})$ in equation (10) with alternative forecasting schemes.

In particular, we shall compute equilibrium laws of motion for capital under alternative assumptions about the information available to decision makers in market i.

32.5 Equilibrium with θ_t stochastic but observed at t

If future θ's are unknown at t, it is appropriate to replace all random variables on the right side of (10) with their conditional expectations based on the information available to decision makers in market i.

For now, we assume that this information set $I_t^p = \begin{bmatrix}\theta^t & \epsilon^{it}\end{bmatrix}$, where z^t represents the infinite history of variable z_s up to time t.

Later we shall give firms less information.

To obtain an appropriate counterpart to (10) under our current assumption about information, we apply a certainty equivalence principle.

In particular, it is appropriate to take (10) and replace each term $(\epsilon_{t+j}^i + \theta_{t+j})$ on the right side with $E[(\epsilon_{t+j}^i + \theta_{t+j})|\theta^t]$.

After using (3) and the i.i.d. assumption about $\{\epsilon_t^i\}$, this gives

$$k_{t+1}^i = \tilde{\lambda}k_t^i + \frac{\tilde{\lambda}\beta\rho}{1 - \tilde{\lambda}\beta\rho}\theta_t$$

or

$$k_{t+1}^i = \tilde{\lambda}k_t^i + \frac{\rho}{\lambda - \rho}\theta_t \tag{11}$$

where $\lambda \equiv (\beta\tilde{\lambda})^{-1}$.

For future purposes, it is useful to represent the equilibrium $\{k_t^i\}_t$ process recursively as

$$\begin{aligned} k_{t+1}^i &= \tilde{\lambda}k_t^i + \frac{1}{\lambda - \rho}\hat{\theta}_{t+1} \\ \hat{\theta}_{t+1} &= \rho\theta_t \\ \theta_{t+1} &= \rho\theta_t + v_t. \end{aligned} \tag{12}$$

32.5.1 Filtering

One noisy signal

We get closer to a model that we ultimately want to study by now assuming that firms in market i do not observe θ_t, but instead observe a history w^t of noisy signals at time t.

In particular, assume that

$$\begin{aligned} w_t &= \theta_t + e_t \\ \theta_{t+1} &= \rho\theta_t + v_t \end{aligned} \tag{13}$$

where e_t and v_t are mutually independent i.i.d. Gaussian shock processes with means of zero and variances σ_e^2 and σ_v^2, respectively.

Define

$$\hat{\theta}_{t+1} = E(\theta_{t+1}|w^t)$$

where $w^t = [w_t, w_{t-1}, \ldots, w_0]$ denotes the history of the w_s process up to and including t.

Associated with the state-space representation (13) is the *innovations representation*

$$\begin{aligned} \hat{\theta}_{t+1} &= \rho\hat{\theta}_t + ka_t \\ w_t &= \hat{\theta}_t + a_t \end{aligned} \tag{14}$$

where $a_t \equiv w_t - E(w_t|w^{t-1})$ is the *innovations* process in w_t and the Kalman gain k is

$$k = \frac{\rho p}{p + \sigma_e^2} \tag{15}$$

and where p satisfies the Riccati equation

$$p = \sigma_v^2 + \frac{p\rho^2\sigma_e^2}{\sigma_e^2 + p}. \tag{16}$$

θ-reconstruction error:

Define the state *reconstruction error* $\tilde{\theta}_t$ by

$$\tilde{\theta}_t = \theta_t - \hat{\theta}_t.$$

Then $p = E\tilde{\theta}_t^2$.

Equations (13) and (14) imply

$$\tilde{\theta}_{t+1} = (\rho - k)\tilde{\theta}_t + v_t - ke_t. \tag{17}$$

Now notice that we can express $\hat{\theta}_{t+1}$ as

$$\hat{\theta}_{t+1} = [\rho\theta_t + v_t] + [ke_t - (\rho - k)\tilde{\theta}_t - v_t], \tag{18}$$

where the first term in braces equals θ_{t+1} and the second term in braces equals $-\tilde{\theta}_{t+1}$.

We can express (11) as

$$k_{t+1}^i = \tilde{\lambda}k_t^i + \frac{1}{\lambda - \rho}E\theta_{t+1}|\theta^t. \tag{19}$$

An application of a certainty equivalence principle asserts that when only w^t is observed, the appropriate solution is found by replacing the information set θ^t with w^t in (19).

Making this substitution and using (18) leads to

$$k_{t+1}^i = \tilde{\lambda} k_t^i + \frac{\rho}{\lambda - \rho}\theta_t + \frac{k}{\lambda - \rho} e_t - \frac{\rho - k}{\lambda - \rho}\tilde{\theta}_t. \tag{20}$$

Simplifying equation (18), we also have

$$\hat{\theta}_{t+1} = \rho\theta_t + k e_t - (\rho - k)\tilde{\theta}_t. \tag{21}$$

Equations (20), (21) describe the equilibrium when w^t is observed.

Relative to (11), the equilibrium acquires a new state variable, namely, the θ–reconstruction error, $\tilde{\theta}_t$.

For future purposes, by using (15), it is useful to write (20) as

$$k_{t+1}^i = \tilde{\lambda} k_t^i + \frac{\rho}{\lambda - \rho}\theta_t + \frac{1}{\lambda - \rho}\frac{p\rho}{p + \sigma_e^2} e_t - \frac{1}{\lambda - \rho}\frac{\rho\sigma_e^2}{p + \sigma_e^2}\tilde{\theta}_t \tag{22}$$

In summary, when decision makers in market i observe a noisy signal w_t on θ_t at t, we can represent an equilibrium law of motion for k_t^i as

$$\begin{aligned}
k_{t+1}^i &= \tilde{\lambda} k_t^i + \frac{1}{\lambda - \rho}\hat{\theta}_{t+1} \\
\hat{\theta}_{t+1} &= \rho\theta_t + \frac{\rho p}{p + \sigma_e^2} e_t - \frac{\rho\sigma_e^2}{p + \sigma_e^2}\tilde{\theta}_t \\
\tilde{\theta}_{t+1} &= \frac{\rho\sigma_e^2}{p + \sigma_e^2}\tilde{\theta}_t - \frac{p\rho}{p + \sigma_e^2} e_t + v_t \\
\theta_{t+1} &= \rho\theta_t + v_t.
\end{aligned} \tag{23}$$

32.5.2 Two noisy signals

We now construct a **pooling equilibrium** by assuming that a firm in industry i receives a vector w_t of *two* noisy signals on θ_t:

$$\begin{aligned}
\theta_{t+1} &= \rho\theta_t + v_t \\
w_t &= \begin{bmatrix} 1 \\ 1 \end{bmatrix}\theta_t + \begin{bmatrix} e_{1t} \\ e_{2t} \end{bmatrix}
\end{aligned}$$

To justify that we are constructing is a **pooling equilibrium** we can assume that

$$\begin{bmatrix} e_{1t} \\ e_{2t} \end{bmatrix} = \begin{bmatrix} \epsilon_t^1 \\ \epsilon_t^2 \end{bmatrix}$$

so that a firm in industry i observes the noisy signals on that θ_t presented to firms in both industries i and $-i$.

The appropriate innovations representation becomes

$$\begin{aligned}
\hat{\theta}_{t+1} &= \rho\hat{\theta}_t + k a_t \\
w_t &= \begin{bmatrix} 1 \\ 1 \end{bmatrix}\hat{\theta}_t + a_t
\end{aligned} \tag{24}$$

where $a_t \equiv w_t - E[w_t | w^{t-1}]$ is a (2×1) vector of innovations in w_t and k is now a (1×2) vector of Kalman gains.

Formulas for the Kalman filter imply that

$$k = \frac{\rho p}{2p + \sigma_e^2} \begin{bmatrix} 1 & 1 \end{bmatrix} \tag{25}$$

where $p = E\tilde{\theta}_t \tilde{\theta}_t^T$ now satisfies the Riccati equation

$$p = \sigma_v^2 + \frac{p\rho^2\sigma_e^2}{2p + \sigma_e^2}. \tag{26}$$

Thus, when a representative firm in industry i observes *two* noisy signals on θ_t, we can express the equilibrium law of motion for capital recursively as

$$\begin{aligned}
k_{t+1}^i &= \tilde{\lambda} k_t^i + \frac{1}{\lambda - \rho}\hat{\theta}_{t+1} \\
\hat{\theta}_{t+1} &= \rho\theta_t + \frac{\rho p}{2p + \sigma_e^2}(e_{1t} + e_{2t}) - \frac{\rho\sigma_e^2}{2p + \sigma_e^2}\tilde{\theta}_t \\
\tilde{\theta}_{t+1} &= \frac{\rho\sigma_e^2}{2p + \sigma_e^2}\tilde{\theta}_t - \frac{p\rho}{2p + \sigma_e^2}(e_{1t} + e_{2t}) + v_t \\
\theta_{t+1} &= \rho\theta_t + v_t.
\end{aligned} \tag{27}$$

Below, by using a guess-and-verify tactic, we shall show that outcomes in this **pooling equilibrium** equal those in an equilibrium under the alternative information structure that interested [Tow83].[4]

32.6 Guess-and-verify tactic

As a preliminary step we shall take our recursive representation (23) of an equilibrium in industry i with one noisy signal on θ_t and perform the following steps:

- Compute λ and $\tilde{\lambda}$ by posing a root-finding problem and then solving it using `numpy.roots`
- Compute p by forming the appropriate discrete Riccati equation and then solving it using `quantecon.solve_discrete_riccati`
- Add a *measurement equation* for $P_t^i = bk_t^i + \theta_t + e_t$, $\theta_t + e_t$, and e_t to system (23). Write the resulting system in state-space form and encode it using `quantecon.LinearStateSpace`
- Use methods of the `quantecon.LinearStateSpace` to compute impulse response functions of k_t^i with respect to shocks v_t, e_t.

After analyzing the one-noisy-signal structure in this way, by making appropriate modifications we shall analyze the two-noisy-signal structure.

We proceed to analyze first the one-noisy-signal structure and then the two-noisy-signal structure.

32.7 Equilibrium with one signal on θ_t

32.7.1 Step 1: Solve for $\tilde{\lambda}$ and λ

1. Cast $(\lambda - 1)\left(\lambda - \frac{1}{\beta}\right) = b\lambda$ as $p(\lambda) = 0$ where p is a polynomial function of λ.
2. Use `numpy.roots` to solve for the roots of p

[4] [PS05] verify the same claim by applying machinery of [PCL86].

3. Verify $\lambda \approx \frac{1}{\beta\tilde{\lambda}}$

Note that $p\left(\lambda\right)=\lambda^{2}-\left(1+b+\frac{1}{\beta}\right)\lambda+\frac{1}{\beta}$.

32.7.2 Step 2: Solve for p

1. Cast $p=\sigma_{v}^{2}+\frac{p\rho^{2}\sigma_{e}^{2}}{2p+\sigma_{e}^{2}}$ as a discrete matrix Riccati equation.
2. Use `quantecon.solve_discrete_riccati` to solve for p
3. Verify $p\approx\sigma_{v}^{2}+\frac{p\rho^{2}\sigma_{e}^{2}}{2p+\sigma_{e}^{2}}$

Note that:

$$\begin{aligned}
A= & \left[\begin{array}{c}\rho\end{array}\right]\\
B= & \left[\begin{array}{c}\sqrt{2}\end{array}\right]\\
R= & \left[\begin{array}{c}\sigma_{e}^{2}\end{array}\right]\\
Q= & \left[\begin{array}{c}\sigma_{v}^{2}\end{array}\right]\\
N= & \left[\begin{array}{c}0\end{array}\right]
\end{aligned}$$

32.7.3 Step 3: Represent the system using `quantecon.LinearStateSpace`

We use the following representation for constructing the `quantecon.LinearStateSpace` instance.

$$\underbrace{\left[\begin{array}{c}e_{t+1}\\k_{t+1}^{i}\\\tilde{\theta}_{t+1}\\P_{t+1}\\\theta_{t+1}\\v_{t+1}\end{array}\right]}_{x_{t+1}}=\underbrace{\left[\begin{array}{cccccc}0 & 0 & 0 & 0 & 0 & 0\\\frac{\kappa}{\lambda-\rho} & \tilde{\lambda} & \frac{-1}{\lambda-\rho}\frac{\kappa\sigma_{e}^{2}}{p} & 0 & \frac{\rho}{\lambda-\rho} & 0\\-\kappa & 0 & \frac{\kappa\sigma_{e}^{2}}{p} & 0 & 0 & 1\\\frac{b\kappa}{\lambda-\rho} & b\tilde{\lambda} & \frac{-b}{\lambda-\rho}\frac{\kappa\sigma_{e}^{2}}{p} & 0 & \frac{b\rho}{\lambda-\rho}+\rho & 1\\0 & 0 & 0 & 0 & \rho & 1\\0 & 0 & 0 & 0 & 0 & 0\end{array}\right]}_{A}\underbrace{\left[\begin{array}{c}e_{t}\\k_{t}^{i}\\\tilde{\theta}_{t}\\P_{t}\\\theta_{t}\\v_{t}\end{array}\right]}_{x_{t}}+\underbrace{\left[\begin{array}{cc}\sigma_{e} & 0\\0 & 0\\0 & 0\\\sigma_{e} & 0\\0 & 0\\0 & \sigma_{v}\end{array}\right]}_{C}\left[\begin{array}{c}z_{1,t+1}\\z_{2,t+1}\end{array}\right]$$

$$\underbrace{\left[\begin{array}{c}P_{t}\\e_{t}+\theta_{t}\\e_{t}\end{array}\right]}_{y_{t}}=\underbrace{\left[\begin{array}{cccccc}0 & 0 & 0 & 1 & 0 & 0\\1 & 0 & 0 & 0 & 1 & 0\\1 & 0 & 0 & 0 & 0 & 0\end{array}\right]}_{G}\underbrace{\left[\begin{array}{c}e_{t}\\k_{t}^{i}\\\tilde{\theta}_{t}\\P_{t}\\\theta_{t}\\v_{t}\end{array}\right]}_{x_{t}}+\underbrace{\left[\begin{array}{c}0\\0\\0\end{array}\right]}_{H}w_{t+1}$$

$$\left[\begin{array}{c}z_{1,t+1}\\z_{2,t+1}\\w_{t+1}\end{array}\right]\sim\mathcal{N}\left(0,I\right)$$

$$\kappa=\frac{\rho p}{p+\sigma_{e}^{2}}$$

This representation includes extraneous variables such as P_t in the state vector.

We formulate things in this way because it allows us easily to compute covariances of these variables with other components of the state vector (step 5 above) by using the `stationary_distributions` method of the `LinearStateSpace` class.

```
import numpy as np
import quantecon as qe
from plotly.subplots import make_subplots
import plotly.graph_objects as go
import plotly.express as px
import plotly.offline as pyo
from statsmodels.regression.linear_model import OLS
from IPython.display import display, Latex, Image

pyo.init_notebook_mode(connected=True)
```

```
β = 0.9   # Discount factor
ρ = 0.8   # Persistence parameter for the hidden state
b = 1.5   # Demand curve parameter
σ_v = 0.5   # Standard deviation of shock to θ_t
σ_e = 0.6   # Standard deviation of shocks to w_t
```

```
# Compute λ
poly = np.array([1, -(1 + β + b) / β, 1 / β])
roots_poly = np.roots(poly)
λ_tilde = roots_poly.min()
λ = roots_poly.max()
```

```
# Verify that λ = (βλ_tilde) ^ (-1)
tol = 1e-12
np.max(np.abs(λ - 1 / (β * λ_tilde))) < tol
```

```
True
```

```
A_ricc = np.array([[ρ]])
B_ricc = np.array([[1.]])
R_ricc = np.array([[σ_e ** 2]])
Q_ricc = np.array([[σ_v ** 2]])
N_ricc = np.zeros((1, 1))
p = qe.solve_discrete_riccati(A_ricc, B_ricc, Q_ricc, R_ricc, N_ricc).item()

p_one = p  # Save for comparison later
```

```
# Verify that p = σ_v ^ 2 + p * ρ ^ 2 - (ρ * p) ^ 2 / (p + σ_e ** 2)
tol = 1e-12
np.abs(p - (σ_v ** 2 + p * ρ ** 2 - (ρ * p) ** 2 / (p + σ_e ** 2))) < tol
```

```
True
```

```
κ = ρ * p / (p + σ_e ** 2)
κ_prod = κ * σ_e ** 2 / p

κ_one = κ  # Save for comparison later

A_lss = np.array([[0., 0., 0., 0., 0., 0.],
                 [κ / (λ - ρ), λ_tilde, -κ_prod / (λ - ρ), 0., ρ / (λ - ρ), 0.],
                 [-κ, 0., κ_prod, 0., 0., 1.],
                 [b * κ / (λ - ρ) , b * λ_tilde, -b * κ_prod / (λ - ρ), 0., b * ρ /␣
↪(λ - ρ) + ρ, 1.],
```

(continues on next page)

(continued from previous page)

```
                 [0., 0., 0., 0., ρ, 1.],
                 [0., 0., 0., 0., 0., 0.]])

C_lss = np.array([[σ_e, 0.],
                 [0.,  0.],
                 [0.,  0.],
                 [σ_e, 0.],
                 [0., 0.],
                 [0., σ_v]])

G_lss = np.array([[0., 0., 0., 1., 0., 0.],
                 [1., 0., 0., 0., 1., 0.],
                 [1., 0., 0., 0., 0., 0.]])
```

```
mu_0 = np.array([0., 0., 0., 0., 0., 0.])

lss = qe.LinearStateSpace(A_lss, C_lss, G_lss, mu_0=mu_0)
```

```
ts_length = 100_000
x, y = lss.simulate(ts_length, random_state=1)
```

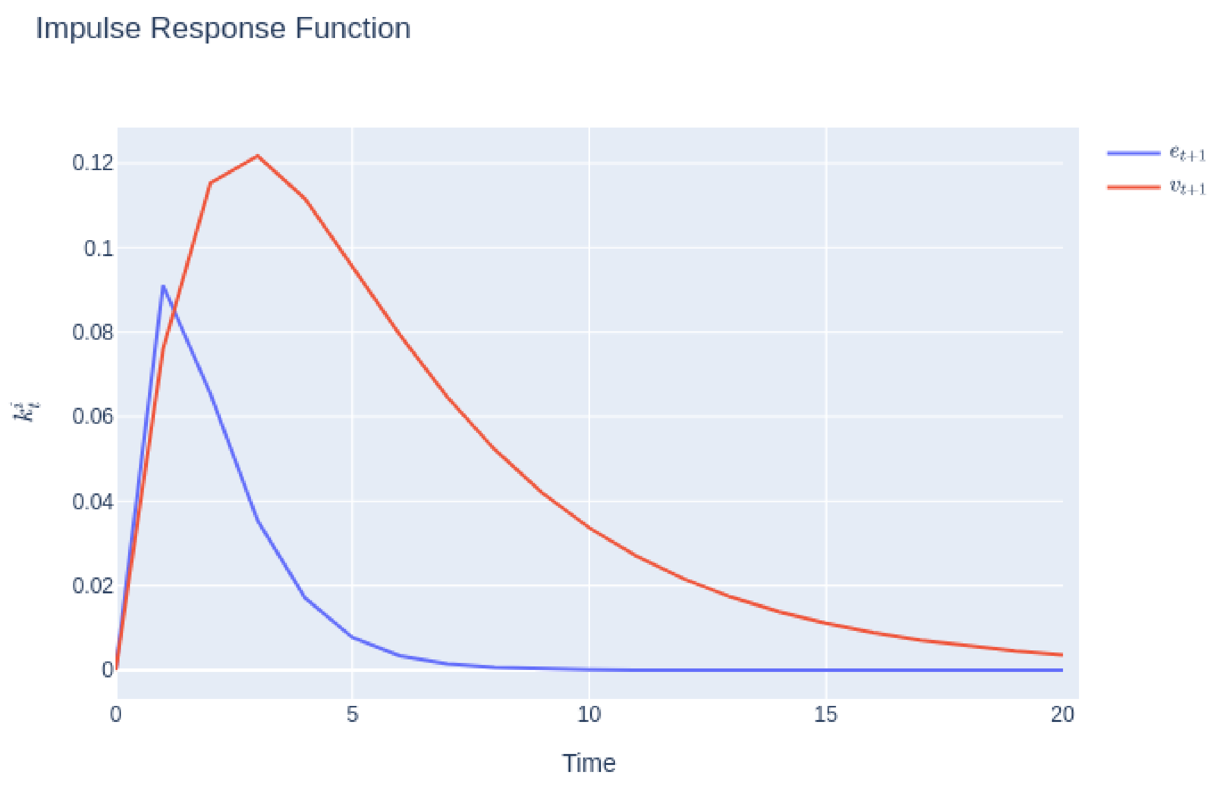
```
# Verify that two ways of computing P_t match
np.max(np.abs(np.array([[1., b, 0., 0., 1., 0.]]) @ x - x[3])) < 1e-12
```

```
True
```

32.7.4 Step 4: Compute impulse response functions

To compute impulse response functions of k_t^i, we use the `impulse_response` method of the `quantecon.LinearStateSpace` class and plot the result.

```
xcoef, ycoef = lss.impulse_response(j=21)
data = np.array([xcoef])[0, :, 1, :]

fig = go.Figure(data=go.Scatter(y=data[:-1, 0], name=r'$e_{t+1}$'))
fig.add_trace(go.Scatter(y=data[1:, 1], name=r'$v_{t+1}$'))
fig.update_layout(title=r'Impulse Response Function',
                  xaxis_title='Time',
                  yaxis_title=r'$k^{i}_{t}$')
fig1 = fig
# Export to PNG file
Image(fig1.to_image(format="png"))
# fig1.show() will provide interactive plot when running
# notebook locally
```



32.7.5 Step 5: Compute stationary covariance matrices and population regressions

We compute stationary covariance matrices by calling the `stationary_distributions` method of the `quantecon.LinearStateSpace` class.

By appropriately decomposing the covariance matrix of the state vector, we obtain ingredients of some population regression coefficients.

$$\Sigma_x = \left[\begin{array}{cc} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{array} \right]$$

where Σ_{11} is the covariance matrix of dependent variables and Σ_{22} is the covariance matrix of independent variables.

Regression coefficients are $\beta = \Sigma_{21}\Sigma_{22}^{-1}$.

To verify an instance of a law of large numbers computation, we construct a long simulation of the state vector and for the resulting sample compute the ordinary least-squares estimator of β that we shall compare to the corresponding population regression coefficients.

```
_, _, Σ_x, Σ_y, Σ_yx = lss.stationary_distributions()

Σ_11 = Σ_x[0, 0]
Σ_12 = Σ_x[0, 1:4]
Σ_21 = Σ_x[1:4, 0]
Σ_22 = Σ_x[1:4, 1:4]
```

(continues on next page)

(continued from previous page)

```
reg_coeffs = Σ_12 @ np.linalg.inv(Σ_22)

print('Regression coefficients (e_t on k_t, P_t, \\tilde{\\theta_t})')
print('------------------------------')
print(r'k_t:', reg_coeffs[0])
print(r'\tilde{\theta_t}:', reg_coeffs[1])
print(r'P_t:', reg_coeffs[2])
```

```
Regression coefficients (e_t on k_t, P_t, \tilde{\theta_t})
------------------------------
k_t: -3.2755568452197705
\tilde{\theta_t}: -0.964946117047546
P_t: 0.9649461170475461
```

```
# Compute R squared
R_squared = reg_coeffs @ Σ_x[1:4, 1:4] @ reg_coeffs  / Σ_x[0, 0]
R_squared
```

```
0.9649461170475461
```

```
# Verify that the computed coefficients are close to least squares estimates
model = OLS(x[0], x[1:4].T)
reg_res = model.fit()
np.max(np.abs(reg_coeffs - reg_res.params)) < 1e-2
```

```
True
```

```
# Verify that R_squared matches least squares estimate
np.abs(reg_res.rsquared - R_squared) < 1e-2
```

```
True
```

```
# Verify that θ_t + e_t can be recovered
model = OLS(y[1], x[1:4].T)
reg_res = model.fit()
np.abs(reg_res.rsquared - 1.) < 1e-6
```

```
True
```

32.8 Equilibrium with two noisy signals on θ_t

Steps 1, 4, and 5 are identical to those for the one-noisy-signal structure.

Step 2 requires only a straightforward modification.

For step 3, we use construct the following state-space representation so that we can get our hands on all of the random processes that we require in order to compute a regression of the noisy signal about θ from the other industry that a firm receives directly in a pooling equilibrium on the information that a firm receives in Townsend's original model.

For this purpose, we include equilibrium goods prices from both industries appear in the state vector:

$$
\underbrace{\begin{bmatrix} e_{1,t+1} \\ e_{2,t+1} \\ k_{t+1}^{i} \\ \tilde{\theta}_{t+1} \\ P_{t+1}^{1} \\ P_{t+1}^{2} \\ \theta_{t+1} \\ v_{t+1} \end{bmatrix}}_{x_{t+1}} = \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{\kappa}{\lambda-\rho} & \frac{\kappa}{\lambda-\rho} & \tilde{\lambda} & \frac{-1}{\lambda-\rho}\frac{\kappa\sigma_e^2}{p} & 0 & 0 & \frac{\rho}{\lambda-\rho} & 0 \\ -\kappa & -\kappa & 0 & \frac{\kappa\sigma_e^2}{p} & 0 & 0 & 0 & 1 \\ \frac{b\kappa}{\lambda-\rho} & \frac{b\kappa}{\lambda-\rho} & b\tilde{\lambda} & \frac{-b}{\lambda-\rho}\frac{\kappa\sigma_e^2}{p} & 0 & 0 & \frac{b\rho}{\lambda-\rho}+\rho & 1 \\ \frac{b\kappa}{\lambda-\rho} & \frac{b\kappa}{\lambda-\rho} & b\tilde{\lambda} & \frac{-b}{\lambda-\rho}\frac{\kappa\sigma_e^2}{p} & 0 & 0 & \frac{b\rho}{\lambda-\rho}+\rho & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & \rho & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} e_{1,t} \\ e_{2,t} \\ k_{t}^{i} \\ \tilde{\theta}_{t} \\ P_{t}^{1} \\ P_{t}^{2} \\ \theta_{t} \\ v_{t} \end{bmatrix}}_{x_{t}} + \underbrace{\begin{bmatrix} \sigma_e & 0 & 0 \\ 0 & \sigma_e & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ \sigma_e & 0 & 0 \\ 0 & \sigma_e & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \sigma_v \end{bmatrix}}_{C} \begin{bmatrix} z_{1,t+1} \\ z_{2,t+1} \\ z_{3,t+1} \end{bmatrix}
$$

$$
\underbrace{\begin{bmatrix} P_t^1 \\ P_t^2 \\ e_{1,t}+\theta_t \\ e_{2,t}+\theta_t \\ e_{1,t} \\ e_{2,t} \end{bmatrix}}_{y_t} = \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{G} \underbrace{\begin{bmatrix} e_{1,t} \\ e_{2,t} \\ k_{t}^{i} \\ \tilde{\theta}_{t} \\ P_{t}^{1} \\ P_{t}^{2} \\ \theta_{t} \\ v_{t} \end{bmatrix}}_{x_{t}} + \underbrace{\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}}_{H} w_{t+1}
$$

$$
\begin{bmatrix} z_{1,t+1} \\ z_{2,t+1} \\ z_{3,t+1} \\ w_{t+1} \end{bmatrix} \sim \mathcal{N}(0, I)
$$

$$
\kappa = \frac{\rho p}{2p + \sigma_e^2}
$$

```
A_ricc = np.array([[ρ]])
B_ricc = np.array([[np.sqrt(2)]])
R_ricc = np.array([[σ_e ** 2]])
Q_ricc = np.array([[σ_v ** 2]])
N_ricc = np.zeros((1, 1))
p = qe.solve_discrete_riccati(A_ricc, B_ricc, Q_ricc, R_ricc, N_ricc).item()

p_two = p  # Save for comparison later
```

```
# Verify that p = σ_v^2 + (pρ^2σ_e^2) / (2p + σ_e^2)
tol = 1e-12
np.abs(p - (σ_v ** 2 + p * ρ ** 2 * σ_e ** 2 / (2 * p + σ_e ** 2))) < tol
```

```
True
```

```
κ = ρ * p / (2 * p + σ_e ** 2)
κ_prod = κ * σ_e ** 2 / p

κ_two = κ  # Save for comparison later

A_lss = np.array([[0., 0., 0., 0., 0., 0., 0., 0.],
                  [0., 0., 0., 0., 0., 0., 0., 0.],
                  [κ / (λ - ρ), κ / (λ - ρ), λ_tilde, -κ_prod / (λ - ρ), 0., 0., ρ /
 ↪(λ - ρ), 0.],
                  [-κ, -κ, 0., κ_prod, 0., 0., 0., 1.],
```

(continues on next page)

(continued from previous page)

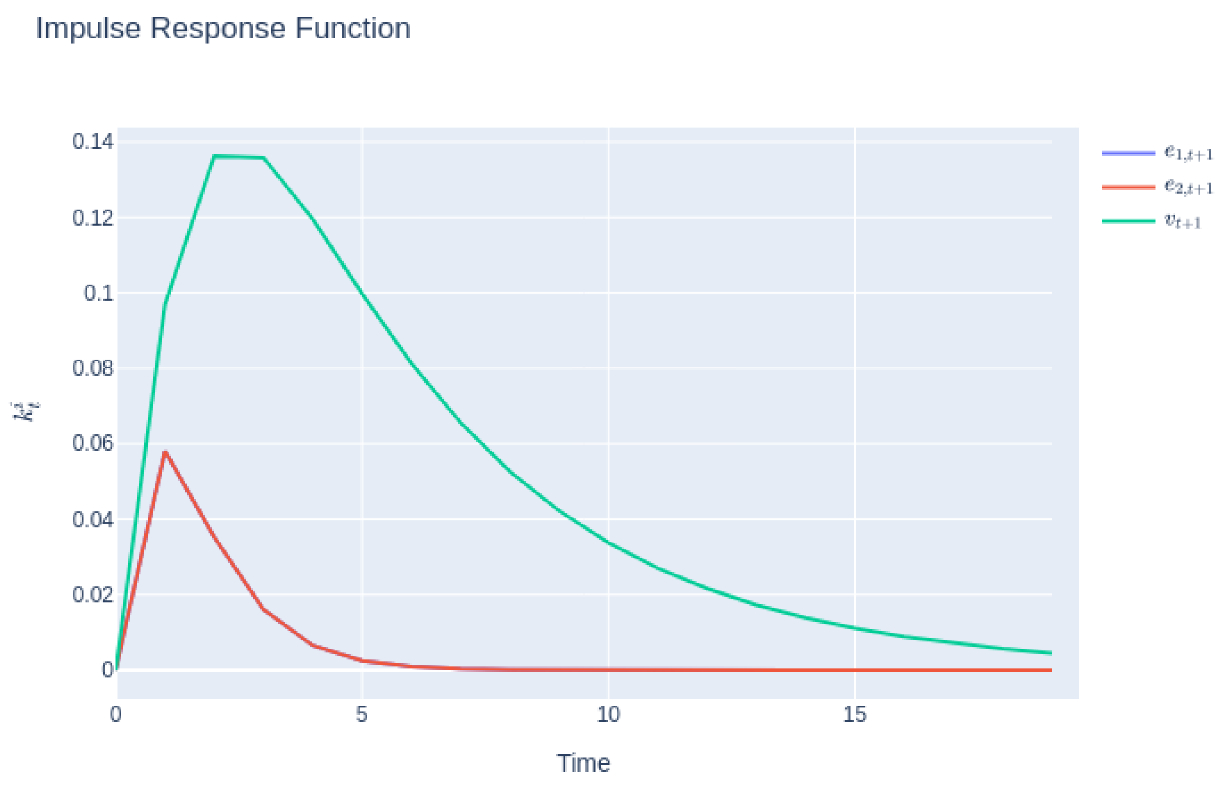
```
                 [b * κ / (λ - ρ), b * κ / (λ - ρ), b * λ_tilde, -b * κ_prod / (λ -␣
↪ρ), 0., 0., b * ρ / (λ - ρ) + ρ, 1.],
                 [b * κ / (λ - ρ), b * κ / (λ - ρ), b * λ_tilde, -b * κ_prod / (λ -␣
↪ρ), 0., 0., b * ρ / (λ - ρ) + ρ, 1.],
                 [0., 0., 0., 0., 0., 0., ρ, 1.],
                 [0., 0., 0., 0., 0., 0., 0., 0.]])

C_lss = np.array([[σ_e, 0., 0.],
                 [0., σ_e, 0.],
                 [0., 0.,  0.],
                 [0., 0.,  0.],
                 [σ_e, 0., 0.],
                 [0., σ_e, 0.],
                 [0., 0., 0.],
                 [0., 0., σ_v]])

G_lss = np.array([[0., 0., 0., 0., 1., 0., 0., 0.],
                 [0., 0, 0, 0., 0., 1., 0., 0.],
                 [1., 0., 0., 0., 0., 0., 1., 0.],
                 [0., 1., 0., 0., 0., 0., 1., 0.],
                 [1., 0., 0., 0., 0., 0., 0., 0.],
                 [0., 1., 0., 0., 0., 0., 0., 0.]])
```

```
mu_0 = np.array([0., 0., 0., 0., 0., 0., 0., 0.])

lss = qe.LinearStateSpace(A_lss, C_lss, G_lss, mu_0=mu_0)
```

```
ts_length = 100_000
x, y = lss.simulate(ts_length, random_state=1)
```

```
xcoef, ycoef = lss.impulse_response(j=20)
```

```
data = np.array([xcoef])[0, :, 2, :]

fig = go.Figure(data=go.Scatter(y=data[:-1, 0], name=r'$e_{1,t+1}$'))
fig.add_trace(go.Scatter(y=data[:-1, 1], name=r'$e_{2,t+1}$'))
fig.add_trace(go.Scatter(y=data[1:, 2], name=r'$v_{t+1}$'))
fig.update_layout(title=r'Impulse Response Function',
                   xaxis_title='Time',
                   yaxis_title=r'$k^{i}_{t}$')
fig2=fig
# Export to PNG file
Image(fig2.to_image(format="png"))
# fig2.show() will provide interactive plot when running
# notebook locally
```



```
_, _, Σ_x, Σ_y, Σ_yx = lss.stationary_distributions()

Σ_11 = Σ_x[1, 1]
Σ_12 = Σ_x[1, 2:5]
Σ_21 = Σ_x[2:5, 1]
Σ_22 = Σ_x[2:5, 2:5]

reg_coeffs = Σ_12 @ np.linalg.inv(Σ_22)

print('Regression coefficients (e_{2,t} on k_t, P^{1}_t, \\tilde{\\theta_t})')
print('------------------------------')
print(r'k_t:', reg_coeffs[0])
print(r'\tilde{\theta_t}:', reg_coeffs[1])
print(r'P_t:', reg_coeffs[2])
```

```
Regression coefficients (e_{2,t} on k_t, P^{1}_t, \tilde{\theta_t})
------------------------------
k_t: 0.0
\tilde{\theta_t}: 0.0
P_t: 0.0
```

```
# Compute R squared
R_squared = reg_coeffs @ Σ_x[2:5, 2:5] @ reg_coeffs  / Σ_x[1, 1]
R_squared
```

```
0.0
```

```
# Verify that the computed coefficients are close to least squares estimates
model = OLS(x[1], x[2:5].T)
reg_res = model.fit()
np.max(np.abs(reg_coeffs - reg_res.params)) < 1e-2
```

```
True
```

```
# Verify that R_squared matches least squares estimate
np.abs(reg_res.rsquared - R_squared) < 1e-2
```

```
True
```

```
_, _, Σ_x, Σ_y, Σ_yx = lss.stationary_distributions()

Σ_11 = Σ_x[1, 1]
Σ_12 = Σ_x[1, 2:6]
Σ_21 = Σ_x[2:6, 1]
Σ_22 = Σ_x[2:6, 2:6]

reg_coeffs = Σ_12 @ np.linalg.inv(Σ_22)

print('Regression coefficients (e_{2,t} on k_t, P^{1}_t, P^{2}_t, \\tilde{\\theta_t})
↪')
print('------------------------------')
print(r'k_t:', reg_coeffs[0])
print(r'\tilde{\theta_t}:', reg_coeffs[1])
print(r'P^{1}_t:', reg_coeffs[2])
print(r'P^{2}_t:', reg_coeffs[3])
```

```
Regression coefficients (e_{2,t} on k_t, P^{1}_t, P^{2}_t, \tilde{\theta_t})
------------------------------
k_t: -3.1373589171035654
\tilde{\theta_t}: -0.924234396744368
P^{1}_t: -0.037882801627815835
P^{2}_t: 0.9621171983721839
```

```
# Compute R squared
R_squared = reg_coeffs @ Σ_x[2:6, 2:6] @ reg_coeffs  / Σ_x[1, 1]
R_squared
```

```
0.9621171983721838
```

32.9 Key step

Now we come to the key step of verifying that equilibrium outcomes for prices and quantities are identical in the pooling equilibrium and Townsend's original model.

We accomplish this by compute a population linear least squares regression of the noisy signal that firms in the other industry receive in a pooling equilibrium on time t information that a firm receives in Townsend's original model.

Let's compute the regression and stare at the R^2:

```
# Verify that ϑ_t + e^{2}_t can be recovered

# ϑ_t + e^{2}_t on k^{i}_t, P^{1}_t, P^{2}_t, \\tilde{\\theta_t}


model = OLS(y[1], x[2:6].T)
reg_res = model.fit()
np.abs(reg_res.rsquared - 1.) < 1e-6
```

```
True
```

```
reg_res.rsquared
```

```
1.0
```

The R^2 in this regression equals 1.

That verifies that a firm's information set in Townsend's original model equals its information set in a pooling equilibrium.

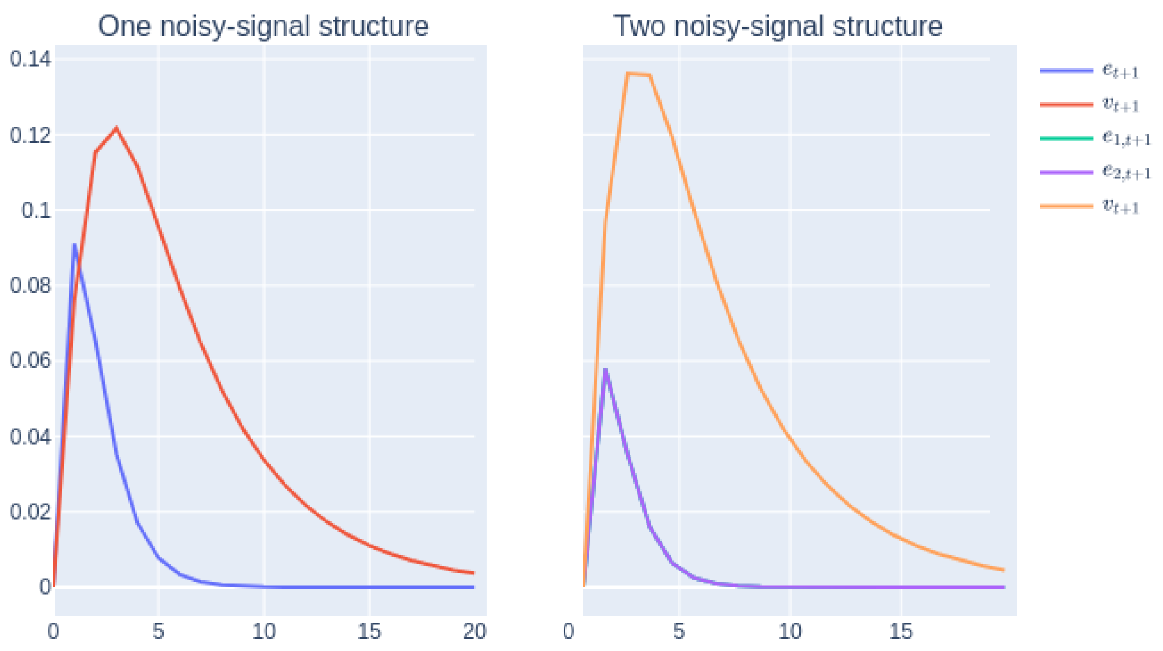
Therefore, equilibrium prices and quantities in Townsend's original model equal those in a pooling equilibrium.

32.10 Comparison of the two signal structures

It is enlightening side by side to plot impulse response functions for capital in an industry for the two information noisy-signal information structures.

Please remember that the two-signal structure corresponds to the **pooling equilibrium** and also **Townsend's original model**.

```
fig_comb = go.Figure(data=[*fig1.data,
                           *fig2.update_traces(xaxis='x2', yaxis='y2').data]).set_
↪subplots(1, 2,
                                                                                    ␣
↪   subplot_titles=("One noisy-signal structure", "Two noisy-signal structure"),
                                                                                    ␣
↪   horizontal_spacing=0.1,
                                                                                    ␣
↪   shared_yaxes=True)
# Export to PNG file
Image(fig_comb.to_image(format="png"))
# fig_comb.show() will provide interactive plot when running
# notebook locally
```



The graphs above show that

- the response of k_t^i to shocks v_t to the hidden Markov demand state θ_t process is **larger** in the two-noisy=signal structure
- the response of k_t^i to idiosyncratic *own-market* noise-shocks e_t is **smaller** in the two-noisy-signal structure

Taken together, these findings in turn can be shown to imply that time series correlations and coherences between outputs in the two industries are higher in the two-noisy-signals or **pooling** model.

The enhanced influence of the shocks v_t to the hidden Markov demand state θ_t process that emerges from the two-noisy-signal model relative to the one-noisy-signal model is a symptom of a lower equilibrium hidden-state reconstruction error variance in the two-signal model:

```
display(Latex('$\\textbf{Reconstruction error variances}$'))
display(Latex(f'One-noise structure: {round(p_one, 6)}'))
display(Latex(f'Two-noise structure: {round(p_two, 6)}'))
```

$$\textbf{Reconstruction error variances}$$

$$One-noisestructure: 0.36618$$

$$Two-noisestructure: 0.324062$$

Kalman gains for the two structures are

```
display(Latex('$\\textbf{Kalman Gains}$'))
display(Latex(f'One noisy-signal structure: {round(κ_one, 6)}'))
display(Latex(f'Two noisy-signals structure: {round(κ_two, 6)}'))
```

Kalman Gains

$$Onenoisy - signalstructure : 0.403404$$

$$Twonoisy - signalsstructure : 0.25716$$

32.11 Notes on History of the Problem

To truncate what he saw as an intractable, infinite dimensional state space, Townsend constructed an approximating model in which the common hidden Markov demand shock is revealed to all firms after a fixed number of periods.

Thus,

- Townsend wanted to assume that at time t firms in industry i observe $k_t^i, Y_t^i, P_t^i, (P^{-i})^t$, where $(P^{-i})^t$ is the history of prices in the other market up to time t.
- Because that turned out to be too challenging, Townsend made an alternative assumption that eased his calculations: that after a large number S of periods, firms in industry i observe the hidden Markov component of the demand shock θ_{t-S}.

Townsend argued that the more manageable model could do a good job of approximating the intractable model in which the Markov component of the demand shock remains unobserved for ever.

By applying technical machinery of [PCL86], [PS05] showed that there is a recursive representation of the equilibrium of the perpetually and symmetrically uninformed model formulated but not completely solved in section 8 of [Tow83].

A reader of [PS05] will notice that their representation of the equilibrium of Townsend's model exactly matches that of the **pooling equilibrium** presented here.

We have structured our notation in this lecture to faciliate comparison of the **pooling equilibrium** constructed here with the equilibrium of Townsend's model reported in [PS05].

The computational method of [PS05] is recursive: it enlists the Kalman filter and invariant subspace methods for solving systems of Euler equations[5] .

As [Sin87], [Kas00], and [Sar91] also found, the equilibrium is fully revealing: observed prices tell participants in industry i all of the information held by participants in market $-i$ ($-i$ means not i).

This means that higher-order beliefs play no role: seeing equilibrium prices in effect lets decision makers pool their information sets[6] .

The disappearance of higher order beliefs means that decision makers in this model do not really face a problem of forecasting the forecasts of others.

They know those forecasts because they are the same as their own.

[5] See [AHMS96] for an account of invariant subspace methods.

[6] See [AHMS96] for a discussion of the information assumptions needed to create a situation in which higher order beliefs appear in equilibrium decision rules. The way to read our findings in light of [AMS02] is that Townsend's section 8 model has too few sources of random shocks relative to sources of signals to permit higher order beliefs to play a role.

32.11.1 Further historical remarks

[Sar91] proposed a way to compute an equilibrium without making Townsend's approximation.

Extending the reasoning of [Mut60], Sargent noticed that it is possible to summarize the relevant history with a low dimensional object, namely, a small number of current and lagged forecasting errors.

Positing an equilibrium in a space of perceived laws of motion for endogenous variables that takes the form of a vector autoregressive, moving average, Sargent described an equilibrium as a fixed point of a mapping from the perceived law of motion to the actual law of motion of that form.

Sargent worked in the time domain and had to guess and verify the appropriate orders of the autoregressive and moving average pieces of the equilibrium representation.

By working in the frequency domain [Kas00] showed how to discover the appropriate orders of the autoregressive and moving average parts, and also how to compute an equilibrium.

The [PS05] recursive computational method, which stays in the time domain, also discovered appropriate orders of the autoregressive and moving average pieces.

In addition, by displaying equilibrium representations in the form of [PCL86], [PS05] showed how the moving average piece is linked to the innovation process of the hidden persistent component of the demand shock.

That scalar innovation process is the additional state variable contributed by the problem of extracting a signal from equilibrium prices that decision makers face in Townsend's model.