Part I

Tools and Techniques

CHAPTER

ONE

ORTHOGONAL PROJECTIONS AND THEIR APPLICATIONS

Contents

- *Orthogonal Projections and Their Applications*
  - *Overview*
  - *Key Definitions*
  - *The Orthogonal Projection Theorem*
  - *Orthonormal Basis*
  - *Projection Via Matrix Algebra*
  - *Least Squares Regression*
  - *Orthogonalization and Decomposition*
  - *Exercises*
  - *Solutions*

1.1 Overview

Orthogonal projection is a cornerstone of vector space methods, with many diverse applications.

These include, but are not limited to,

- Least squares projection, also known as linear regression
- Conditional expectations for multivariate normal (Gaussian) distributions
- Gram–Schmidt orthogonalization
- QR decomposition
- Orthogonal polynomials
- etc

In this lecture, we focus on

- key ideas
- least squares regression

We'll require the following imports:

```python
import numpy as np
from scipy.linalg import qr
```

1.1.1 Further Reading

For background and foundational concepts, see our lecture on linear algebra.

For more proofs and greater theoretical detail, see A Primer in Econometric Theory.

For a complete set of proofs in a general setting, see, for example, [Rom05].

For an advanced treatment of projection in the context of least squares prediction, see this book chapter.

1.2 Key Definitions

Assume $x, z \in \mathbb{R}^n$.

Define $\langle x, z \rangle = \sum_i x_i z_i$.

Recall $\|x\|^2 = \langle x, x \rangle$.

The **law of cosines** states that $\langle x, z \rangle = \|x\| \|z\| \cos(\theta)$ where θ is the angle between the vectors x and z.

When $\langle x, z \rangle = 0$, then $\cos(\theta) = 0$ and x and z are said to be **orthogonal** and we write $x \perp z$.

For a linear subspace $S \subset \mathbb{R}^n$, we call $x \in \mathbb{R}^n$ **orthogonal to** S if $x \perp z$ for all $z \in S$, and write $x \perp S$.

The **orthogonal complement** of linear subspace $S \subset \mathbb{R}^n$ is the set $S^\perp := \{x \in \mathbb{R}^n \,:\, x \perp S\}$.

$S^\perp$ is a linear subspace of $\mathbb{R}^n$

- To see this, fix $x, y \in S^\perp$ and $\alpha, \beta \in \mathbb{R}$.
- Observe that if $z \in S$, then

$$\langle \alpha x + \beta y, z \rangle = \alpha \langle x, z \rangle + \beta \langle y, z \rangle = \alpha \times 0 + \beta \times 0 = 0$$

- Hence $\alpha x + \beta y \in S^\perp$, as was to be shown

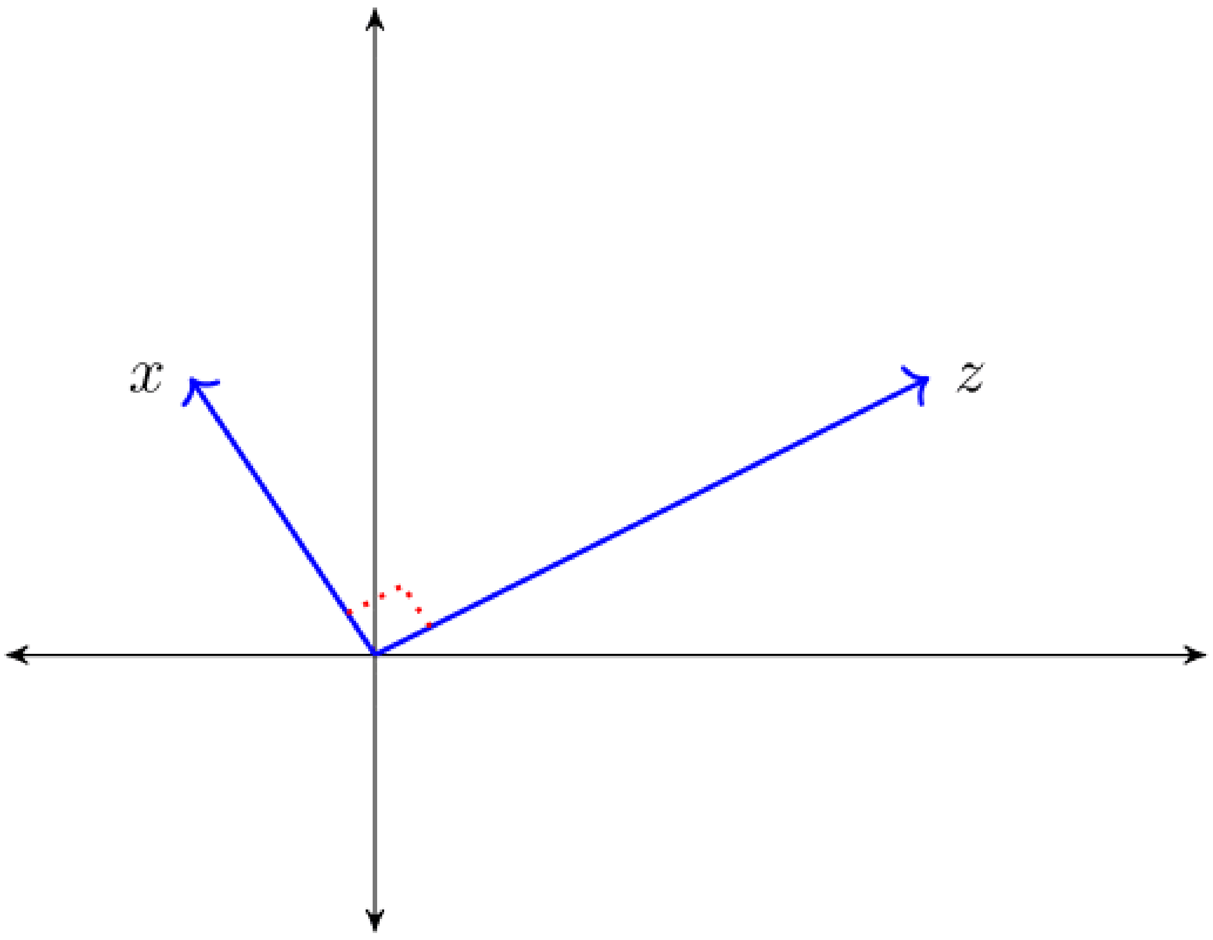
A set of vectors $\{x_1, \dots, x_k\} \subset \mathbb{R}^n$ is called an **orthogonal set** if $x_i \perp x_j$ whenever $i \neq j$.

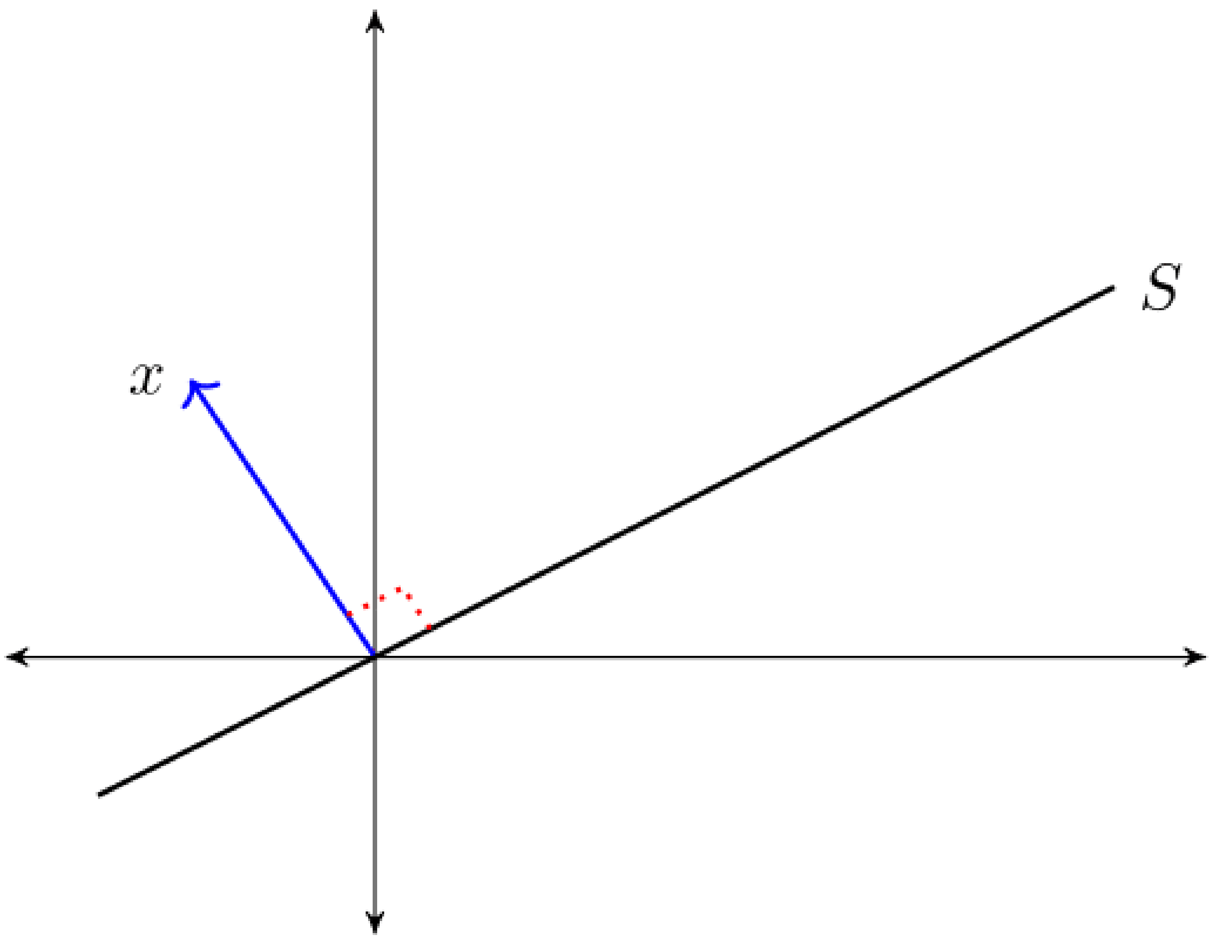
If $\{x_1, \dots, x_k\}$ is an orthogonal set, then the **Pythagorean Law** states that

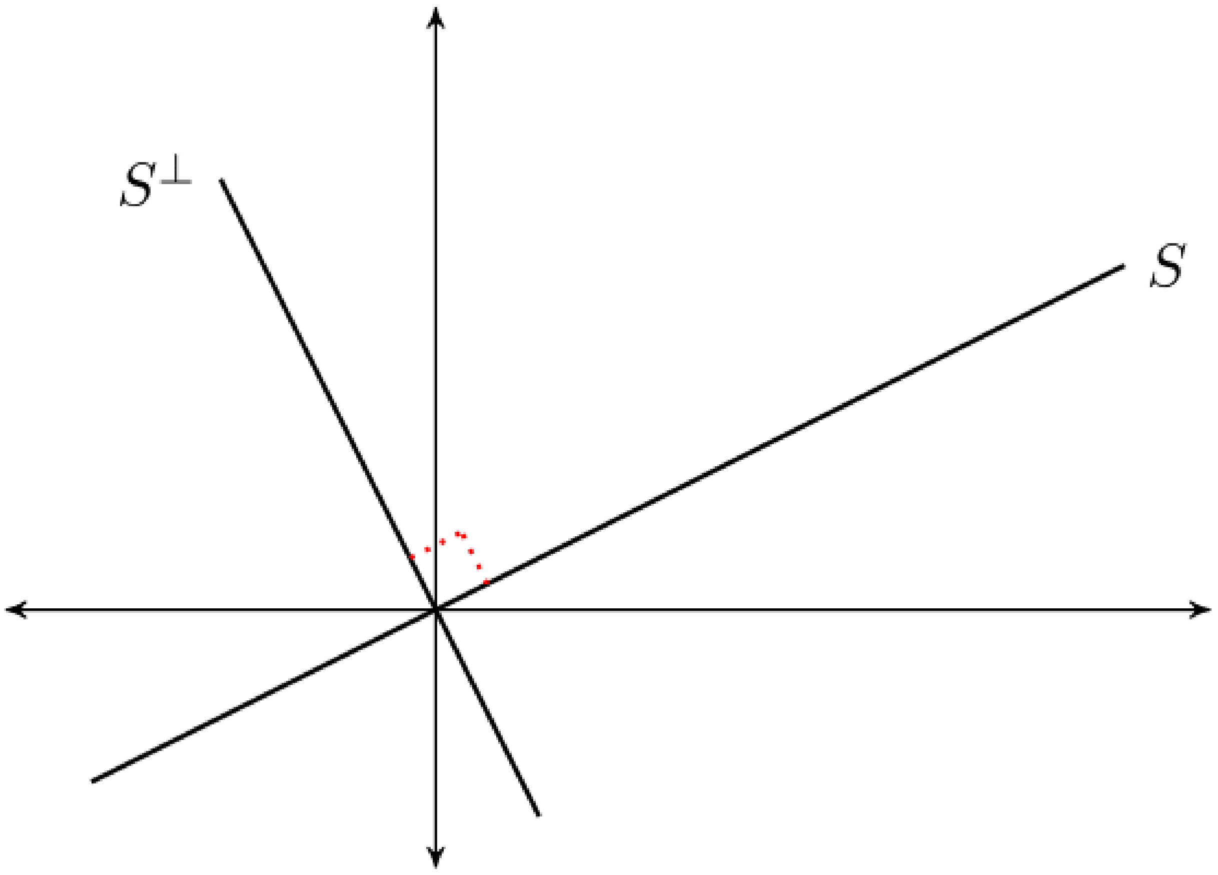
$$\|x_1 + \cdots + x_k\|^2 = \|x_1\|^2 + \cdots + \|x_k\|^2$$

For example, when $k = 2$, $x_1 \perp x_2$ implies

$$\|x_1 + x_2\|^2 = \langle x_1 + x_2, x_1 + x_2 \rangle = \langle x_1, x_1 \rangle + 2 \langle x_2, x_1 \rangle + \langle x_2, x_2 \rangle = \|x_1\|^2 + \|x_2\|^2$$


x
z


x
S


$S^{\perp}$
S

1.2.1 Linear Independence vs Orthogonality

If $X \subset \mathbb{R}^n$ is an orthogonal set and $0 \notin X$, then X is linearly independent.

Proving this is a nice exercise.

While the converse is not true, a kind of partial converse holds, as we'll *see below*.

1.3 The Orthogonal Projection Theorem

What vector within a linear subspace of $\mathbb{R}^n$ best approximates a given vector in $\mathbb{R}^n$?

The next theorem answers this question.

Theorem (OPT) Given $y \in \mathbb{R}^n$ and linear subspace $S \subset \mathbb{R}^n$, there exists a unique solution to the minimization problem

$$\hat{y} := \arg\min_{z \in S} \|y - z\|$$

The minimizer $\hat{y}$ is the unique vector in $\mathbb{R}^n$ that satisfies

- $\hat{y} \in S$
- $y - \hat{y} \perp S$

The vector $\hat{y}$ is called the **orthogonal projection** of y onto S.

The next figure provides some intuition

1.3.1 Proof of Sufficiency

We'll omit the full proof.

But we will prove sufficiency of the asserted conditions.

To this end, let $y \in \mathbb{R}^n$ and let S be a linear subspace of $\mathbb{R}^n$.

Let $\hat{y}$ be a vector in $\mathbb{R}^n$ such that $\hat{y} \in S$ and $y - \hat{y} \perp S$.

Let z be any other point in S and use the fact that S is a linear subspace to deduce

$$\|y - z\|^2 = \|(y - \hat{y}) + (\hat{y} - z)\|^2 = \|y - \hat{y}\|^2 + \|\hat{y} - z\|^2$$

Hence $\|y - z\| \geq \|y - \hat{y}\|$, which completes the proof.

1.3.2 Orthogonal Projection as a Mapping

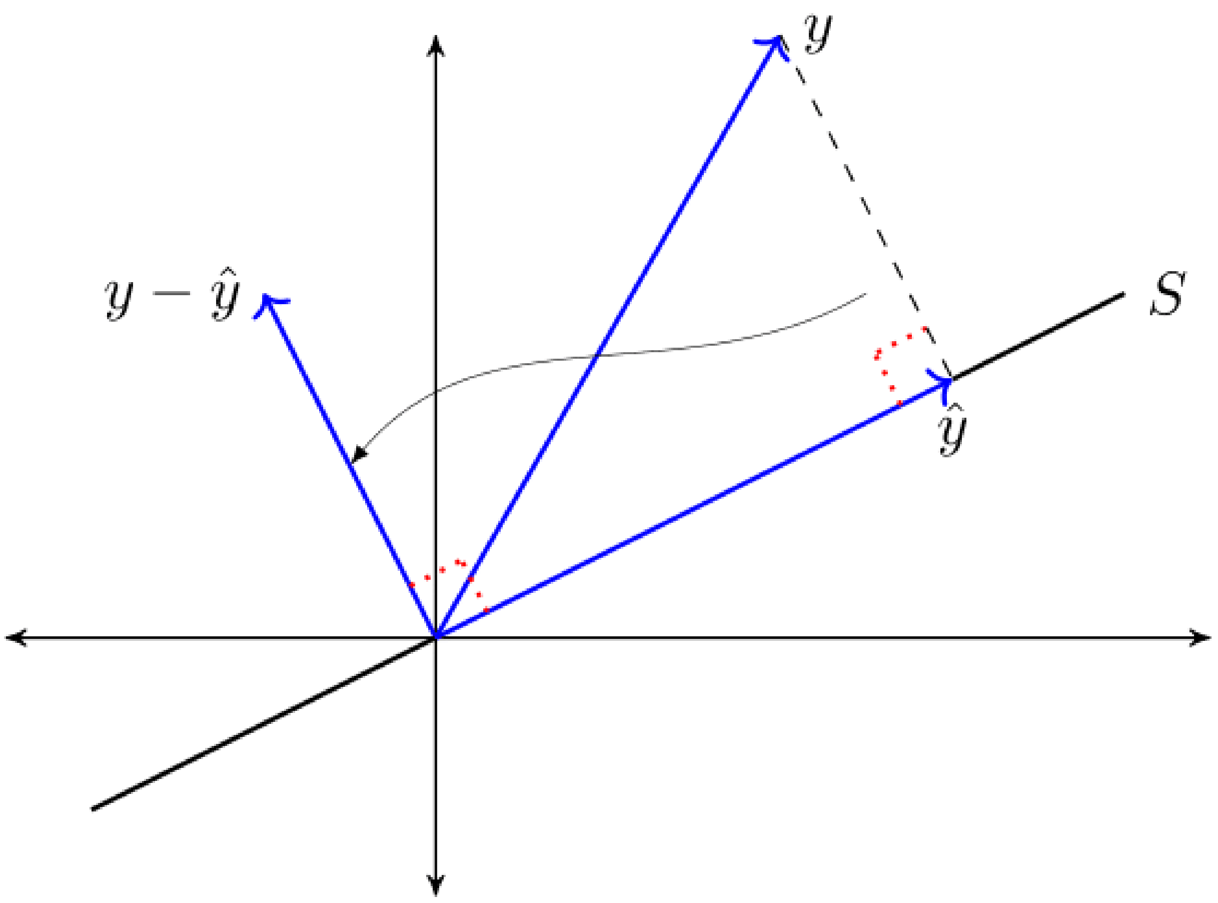
For a linear space Y and a fixed linear subspace S, we have a functional relationship

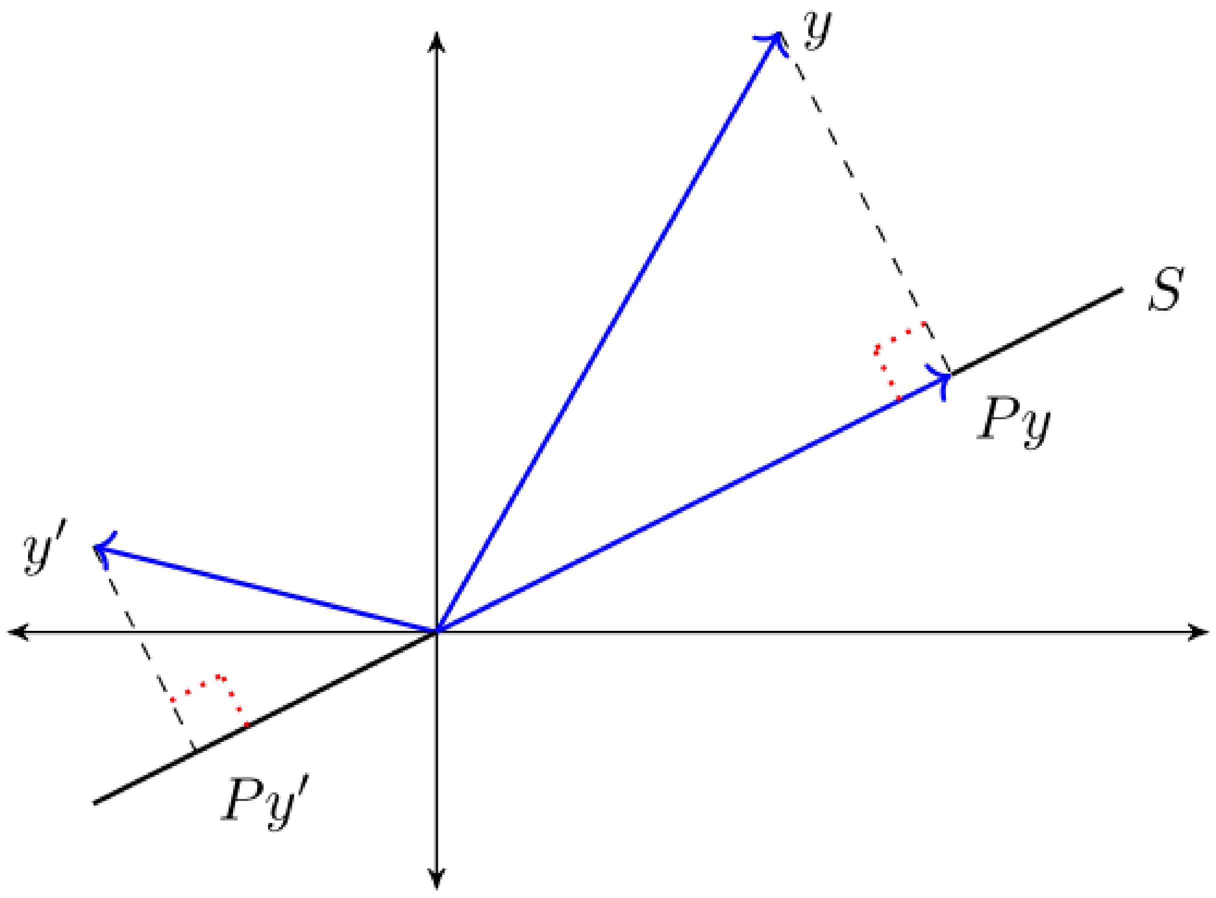
$$y \in Y \; \mapsto \text{ its orthogonal projection } \hat{y} \in S$$

By the OPT, this is a well-defined mapping or *operator* from $\mathbb{R}^n$ to $\mathbb{R}^n$.

In what follows we denote this operator by a matrix P

- Py represents the projection $\hat{y}$.
- This is sometimes expressed as $\hat{E}_S y = Py$, where $\hat{E}$ denotes a **wide-sense expectations operator** and the subscript S indicates that we are projecting y onto the linear subspace S.


y
$y - \hat{y}$
S
$\hat{y}$


y
S
Py
y'
Py'

The operator P is called the **orthogonal projection mapping onto** S.

It is immediate from the OPT that for any $y \in \mathbb{R}^n$

1. $Py \in S$ and
2. $y - Py \perp S$

From this, we can deduce additional useful properties, such as

1. $\|y\|^2 = \|Py\|^2 + \|y - Py\|^2$ and
2. $\|Py\| \leq \|y\|$

For example, to prove 1, observe that $y = Py + y - Py$ and apply the Pythagorean law.

Orthogonal Complement

Let $S \subset \mathbb{R}^n$.

The **orthogonal complement** of S is the linear subspace $S^{\perp}$ that satisfies $x_1 \perp x_2$ for every $x_1 \in S$ and $x_2 \in S^{\perp}$.

Let Y be a linear space with linear subspace S and its orthogonal complement $S^{\perp}$.

We write

$$Y = S \oplus S^{\perp}$$

to indicate that for every $y \in Y$ there is unique $x_1 \in S$ and a unique $x_2 \in S^{\perp}$ such that $y = x_1 + x_2$.

Moreover, $x_1 = \hat{E}_S y$ and $x_2 = y - \hat{E}_S y$.

This amounts to another version of the OPT:

Theorem. If S is a linear subspace of $\mathbb{R}^n$, $\hat{E}_S y = Py$ and $\hat{E}_{S^{\perp}} y = My$, then

$$Py \perp My \quad \text{and} \quad y = Py + My \quad \text{for all } \; y \in \mathbb{R}^n$$

The next figure illustrates

1.4 Orthonormal Basis

An orthogonal set of vectors $O \subset \mathbb{R}^n$ is called an **orthonormal set** if $\|u\| = 1$ for all $u \in O$.

Let S be a linear subspace of $\mathbb{R}^n$ and let $O \subset S$.

If O is orthonormal and span $O = S$, then O is called an **orthonormal basis** of S.

O is necessarily a basis of S (being independent by orthogonality and the fact that no element is the zero vector).

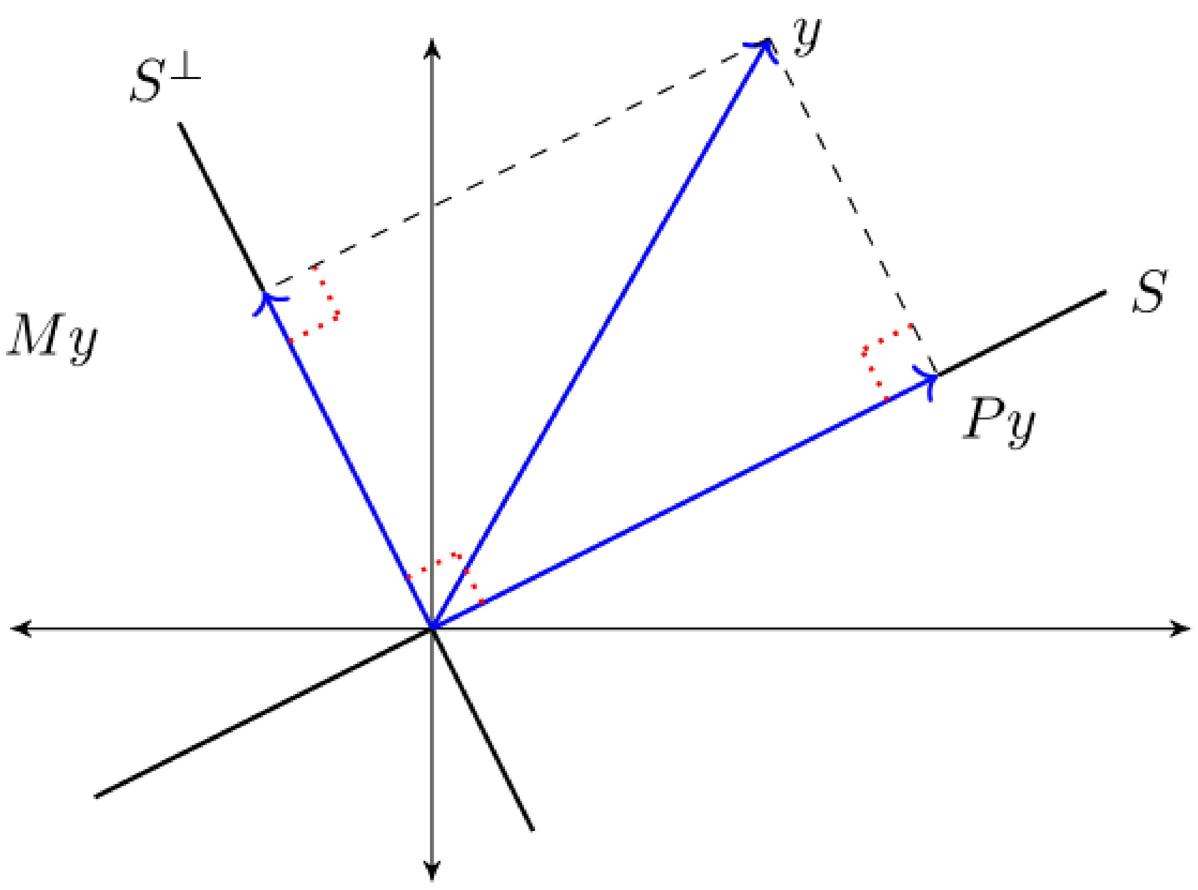
One example of an orthonormal set is the canonical basis $\{e_1, \ldots, e_n\}$ that forms an orthonormal basis of $\mathbb{R}^n$, where e_i is the i th unit vector.

If $\{u_1, \ldots, u_k\}$ is an orthonormal basis of linear subspace S, then

$$x = \sum_{i=1}^{k} \langle x, u_i \rangle u_i \quad \text{for all} \quad x \in S$$

To see this, observe that since $x \in \text{span}\{u_1, \ldots, u_k\}$, we can find scalars $\alpha_1, \ldots, \alpha_k$ that verify

$$x = \sum_{j=1}^{k} \alpha_j u_j \tag{1}$$


$S^{\perp}$
y
My
S
Py

Taking the inner product with respect to u_i gives

$$\langle x, u_i \rangle = \sum_{j=1}^k \alpha_j \langle u_j, u_i \rangle = \alpha_i$$

Combining this result with (1) verifies the claim.

1.4.1 Projection onto an Orthonormal Basis

When a subspace onto which we project is orthonormal, computing the projection simplifies:

Theorem If $\{u_1, \dots, u_k\}$ is an orthonormal basis for S, then

$$Py = \sum_{i=1}^k \langle y, u_i \rangle u_i, \quad \forall\ y \in \mathbb{R}^n \tag{2}$$

Proof: Fix $y \in \mathbb{R}^n$ and let Py be defined as in (2).

Clearly, $Py \in S$.

We claim that $y - Py \perp S$ also holds.

It sufficies to show that $y - Py \perp$ any basis vector u_i.

This is true because

$$\left\langle y - \sum_{i=1}^k \langle y, u_i \rangle u_i, u_j \right\rangle = \langle y, u_j \rangle - \sum_{i=1}^k \langle y, u_i \rangle \langle u_i, u_j \rangle = 0$$

(Why is this sufficient to establish the claim that $y - Py \perp S$?)

1.5 Projection Via Matrix Algebra

Let S be a linear subspace of $\mathbb{R}^n$ and let $y \in \mathbb{R}^n$.

We want to compute the matrix P that verifies

$$\hat E_S y = Py$$

Evidently Py is a linear function from $y \in \mathbb{R}^n$ to $Py \in \mathbb{R}^n$.

This reference is useful.

Theorem. Let the columns of $n \times k$ matrix X form a basis of S. Then

$$P = X(X'X)^{-1}X'$$

Proof: Given arbitrary $y \in \mathbb{R}^n$ and $P = X(X'X)^{-1}X'$, our claim is that

1. $Py \in S$, and
2. $y - Py \perp S$

Claim 1 is true because

$$Py = X(X'X)^{-1}X'y = Xa \quad \text{when} \quad a := (X'X)^{-1}X'y$$

An expression of the form Xa is precisely a linear combination of the columns of X and hence an element of S.

Claim 2 is equivalent to the statement

$$y - X(X'X)^{-1}X'y \perp Xb \quad \text{for all} \quad b \in \mathbb{R}^K$$

To verify this, notice that if $b \in \mathbb{R}^K$, then

$$(Xb)'[y - X(X'X)^{-1}X'y] = b'[X'y - X'y] = 0$$

The proof is now complete.

1.5.1 Starting with the Basis

It is common in applications to start with $n \times k$ matrix X with linearly independent columns and let

$$S := \operatorname{span} X := \operatorname{span}\{\operatorname{col}_1 X, \dots, \operatorname{col}_k X\}$$

Then the columns of X form a basis of S.

From the preceding theorem, $P = X(X'X)^{-1}X'y$ projects y onto S.

In this context, P is often called the **projection matrix**

- The matrix $M = I - P$ satisfies $My = \hat{E}_{S^\perp} y$ and is sometimes called the **annihilator matrix**.

1.5.2 The Orthonormal Case

Suppose that U is $n \times k$ with orthonormal columns.

Let $u_i := \operatorname{col} U_i$ for each i, let $S := \operatorname{span} U$ and let $y \in \mathbb{R}^n$.

We know that the projection of y onto S is

$$Py = U(U'U)^{-1}U'y$$

Since U has orthonormal columns, we have $U'U = I$.

Hence

$$Py = UU'y = \sum_{i=1}^{k} \langle u_i, y \rangle u_i$$

We have recovered our earlier result about projecting onto the span of an orthonormal basis.

1.5.3 Application: Overdetermined Systems of Equations

Let $y \in \mathbb{R}^n$ and let X be $n \times k$ with linearly independent columns.

Given X and y, we seek $b \in \mathbb{R}^k$ that satisfies the system of linear equations $Xb = y$.

If $n > k$ (more equations than unknowns), then b is said to be **overdetermined**.

Intuitively, we may not be able to find a b that satisfies all n equations.

The best approach here is to

- Accept that an exact solution may not exist.
- Look instead for an approximate solution.

By approximate solution, we mean a $b \in \mathbb{R}^k$ such that Xb is close to y.

The next theorem shows that a best approximation is well defined and unique.

The proof uses the OPT.

Theorem The unique minimizer of $\|y - Xb\|$ over $b \in \mathbb{R}^K$ is

$$\hat\beta := (X'X)^{-1}X'y$$

Proof: Note that

$$X\hat\beta = X(X'X)^{-1}X'y = Py$$

Since Py is the orthogonal projection onto $\text{span}(X)$ we have

$$\|y - Py\| \leq \|y - z\| \text{ for any } z \in \text{span}(X)$$

Because $Xb \in \text{span}(X)$

$$\|y - X\hat\beta\| \leq \|y - Xb\| \text{ for any } b \in \mathbb{R}^K$$

This is what we aimed to show.

1.6 Least Squares Regression

Let's apply the theory of orthogonal projection to least squares regression.

This approach provides insights about many geometric properties of linear regression.

We treat only some examples.

1.6.1 Squared Risk Measures

Given pairs $(x, y) \in \mathbb{R}^K \times \mathbb{R}$, consider choosing $f: \mathbb{R}^K \to \mathbb{R}$ to minimize the **risk**

$$R(f) := \mathbb{E}\,[(y - f(x))^2]$$

If probabilities and hence $\mathbb{E}$ are unknown, we cannot solve this problem directly.

However, if a sample is available, we can estimate the risk with the **empirical risk**:

$$\min_{f \in \mathcal{F}} \frac{1}{N} \sum_{n=1}^{N} (y_n - f(x_n))^2$$

Minimizing this expression is called **empirical risk minimization**.

The set $\mathcal{F}$ is sometimes called the hypothesis space.

The theory of statistical learning tells us that to prevent overfitting we should take the set $\mathcal{F}$ to be relatively simple.

If we let $\mathcal{F}$ be the class of linear functions, the problem is

$$\min_{b \in \mathbb{R}^K} \sum_{n=1}^{N} (y_n - b'x_n)^2$$

This is the sample **linear least squares problem**.

1.6.2 Solution

Define the matrices

$$y := \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}, \quad x_n := \begin{pmatrix} x_{n1} \\ x_{n2} \\ \vdots \\ x_{nK} \end{pmatrix} = n\text{-th obs on all regressors}$$

and

$$X := \begin{pmatrix} x_1' \\ x_2' \\ \vdots \\ x_N' \end{pmatrix} =: \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1K} \\ x_{21} & x_{22} & \cdots & x_{2K} \\ \vdots & \vdots & & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{NK} \end{pmatrix}$$

We assume throughout that $N > K$ and X is full column rank.

If you work through the algebra, you will be able to verify that $\|y - Xb\|^2 = \sum_{n=1}^N (y_n - b' x_n)^2$.

Since monotone transforms don't affect minimizers, we have

$$\operatorname*{arg\,min}_{b \in \mathbb{R}^K} \sum_{n=1}^N (y_n - b' x_n)^2 = \operatorname*{arg\,min}_{b \in \mathbb{R}^K} \|y - Xb\|$$

By our results about overdetermined linear systems of equations, the solution is

$$\hat\beta := (X'X)^{-1} X' y$$

Let P and M be the projection and annihilator associated with X:

$$P := X(X'X)^{-1}X' \quad \text{and} \quad M := I - P$$

The **vector of fitted values** is

$$\hat y := X \hat\beta = Py$$

The **vector of residuals** is

$$\hat u := y - \hat y = y - Py = My$$

Here are some more standard definitions:

- The **total sum of squares** is $:= \|y\|^2$.
- The **sum of squared residuals** is $:= \|\hat u\|^2$.
- The **explained sum of squares** is $:= \|\hat y\|^2$.

  TSS = ESS + SSR

We can prove this easily using the OPT.

From the OPT we have $y = \hat y + \hat u$ and $\hat u \perp \hat y$.

Applying the Pythagorean law completes the proof.

1.7 Orthogonalization and Decomposition

Let's return to the connection between linear independence and orthogonality touched on above.

A result of much interest is a famous algorithm for constructing orthonormal sets from linearly independent sets.

The next section gives details.

1.7.1 Gram-Schmidt Orthogonalization

Theorem For each linearly independent set $\{x_1, \ldots, x_k\} \subset \mathbb{R}^n$, there exists an orthonormal set $\{u_1, \ldots, u_k\}$ with

$$\text{span}\{x_1, \ldots, x_i\} = \text{span}\{u_1, \ldots, u_i\} \quad \text{for} \quad i = 1, \ldots, k$$

The **Gram-Schmidt orthogonalization** procedure constructs an orthogonal set $\{u_1, u_2, \ldots, u_n\}$.

One description of this procedure is as follows:

- For $i = 1, \ldots, k$, form $S_i := \text{span}\{x_1, \ldots, x_i\}$ and $S_i^{\perp}$
- Set $v_1 = x_1$
- For $i \geq 2$ set $v_i := \hat{E}_{S_{i-1}^{\perp}} x_i$ and $u_i := v_i / \|v_i\|$

The sequence $u_1, \ldots, u_k$ has the stated properties.

A Gram-Schmidt orthogonalization construction is a key idea behind the Kalman filter described in A First Look at the Kalman filter.

In some exercises below, you are asked to implement this algorithm and test it using projection.

1.7.2 QR Decomposition

The following result uses the preceding algorithm to produce a useful decomposition.

Theorem If X is $n \times k$ with linearly independent columns, then there exists a factorization $X = QR$ where

- R is $k \times k$, upper triangular, and nonsingular
- Q is $n \times k$ with orthonormal columns

Proof sketch: Let

- $x_j := \text{col}_j(X)$
- $\{u_1, \ldots, u_k\}$ be orthonormal with the same span as $\{x_1, \ldots, x_k\}$ (to be constructed using Gram–Schmidt)
- Q be formed from cols u_i

Since $x_j \in \text{span}\{u_1, \ldots, u_j\}$, we have

$$x_j = \sum_{i=1}^{j} \langle u_i, x_j \rangle u_i \quad \text{for } j = 1, \ldots, k$$

Some rearranging gives $X = QR$.

1.7.3 Linear Regression via QR Decomposition

For matrices X and y that overdetermine β in the linear equation system $y = X\beta$, we found the least squares approximator $\hat{\beta} = (X'X)^{-1}X'y$.

Using the QR decomposition $X = QR$ gives

$$\begin{aligned} \hat{\beta} &= (R'Q'QR)^{-1}R'Q'y \\ &= (R'R)^{-1}R'Q'y \\ &= R^{-1}(R')^{-1}R'Q'y = R^{-1}Q'y \end{aligned}$$

Numerical routines would in this case use the alternative form $R\hat{\beta} = Q'y$ and back substitution.

1.8 Exercises

1.8.1 Exercise 1

Show that, for any linear subspace $S \subset \mathbb{R}^n$, $S \cap S^{\perp} = \{0\}$.

1.8.2 Exercise 2

Let $P = X(X'X)^{-1}X'$ and let $M = I - P$. Show that P and M are both idempotent and symmetric. Can you give any intuition as to why they should be idempotent?

1.8.3 Exercise 3

Using Gram-Schmidt orthogonalization, produce a linear projection of y onto the column space of X and verify this using the projection matrix $P := X(X'X)^{-1}X'$ and also using QR decomposition, where:

$$y := \begin{pmatrix} 1 \\ 3 \\ -3 \end{pmatrix},$$

and

$$X := \begin{pmatrix} 1 & 0 \\ 0 & -6 \\ 2 & 2 \end{pmatrix}$$

1.9 Solutions

1.9.1 Exercise 1

If $x \in S$ and $x \in S^{\perp}$, then we have in particular that $\langle x, x \rangle = 0$, but then $x = 0$.

1.9.2 Exercise 2

Symmetry and idempotence of M and P can be established using standard rules for matrix algebra. The intuition behind idempotence of M and P is that both are orthogonal projections. After a point is projected into a given subspace, applying the projection again makes no difference. (A point inside the subspace is not shifted by orthogonal projection onto that space because it is already the closest point in the subspace to itself.).

1.9.3 Exercise 3

Here's a function that computes the orthonormal vectors using the GS algorithm given in the lecture

```python
def gram_schmidt(X):
    """
    Implements Gram-Schmidt orthogonalization.

    Parameters
    ----------
    X : an n x k array with linearly independent columns

    Returns
    -------
    U : an n x k array with orthonormal columns

    """

    # Set up
    n, k = X.shape
    U = np.empty((n, k))
    I = np.eye(n)

    # The first col of U is just the normalized first col of X
    v1 = X[:,0]
    U[:, 0] = v1 / np.sqrt(np.sum(v1 * v1))

    for i in range(1, k):
        # Set up
        b = X[:, i]       # The vector we're going to project
        Z = X[:, 0:i]     # First i-1 columns of X

        # Project onto the orthogonal complement of the col span of Z
        M = I - Z @ np.linalg.inv(Z.T @ Z) @ Z.T
        u = M @ b

        # Normalize
        U[:, i] = u / np.sqrt(np.sum(u * u))

    return U
```

Here are the arrays we'll work with

```python
y = [1, 3, -3]

X = [[1,  0],
     [0, -6],
     [2,  2]]

X, y = [np.asarray(z) for z in (X, y)]
```

First, let's try projection of y onto the column space of X using the ordinary matrix expression:

```
Py1 = X @ np.linalg.inv(X.T @ X) @ X.T @ y
Py1
```

```
array([-0.56521739,  3.26086957, -2.2173913 ])
```

Now let's do the same using an orthonormal basis created from our `gram_schmidt` function

```
U = gram_schmidt(X)
U
```

```
array([[ 0.4472136 , -0.13187609],
       [ 0.        , -0.98907071],
       [ 0.89442719,  0.06593805]])
```

```
Py2 = U @ U.T @ y
Py2
```

```
array([-0.56521739,  3.26086957, -2.2173913 ])
```

This is the same answer. So far so good. Finally, let's try the same thing but with the basis obtained via QR decomposition:

```
Q, R = qr(X, mode='economic')
Q
```

```
array([[-0.4472136 , -0.13187609],
       [-0.        , -0.98907071],
       [-0.89442719,  0.06593805]])
```

```
Py3 = Q @ Q.T @ y
Py3
```

```
array([-0.56521739,  3.26086957, -2.2173913 ])
```

Again, we obtain the same answer.

CHAPTER

TWO

CONTINUOUS STATE MARKOV CHAINS

Contents

- *Continuous State Markov Chains*
  - *Overview*
  - *The Density Case*
  - *Beyond Densities*
  - *Stability*
  - *Exercises*
  - *Solutions*
  - *Appendix*

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

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

2.1 Overview

In a previous lecture, we learned about finite Markov chains, a relatively elementary class of stochastic dynamic models.

The present lecture extends this analysis to continuous (i.e., uncountable) state Markov chains.

Most stochastic dynamic models studied by economists either fit directly into this class or can be represented as continuous state Markov chains after minor modifications.

In this lecture, our focus will be on continuous Markov models that

- evolve in discrete-time
- are often nonlinear

The fact that we accommodate nonlinear models here is significant, because linear stochastic models have their own highly developed toolset, as we'll see *later on*.

The question that interests us most is: Given a particular stochastic dynamic model, how will the state of the system evolve over time?

In particular,

- What happens to the distribution of the state variables?

- Is there anything we can say about the "average behavior" of these variables?
- Is there a notion of "steady state" or "long-run equilibrium" that's applicable to the model?
  - If so, how can we compute it?

Answering these questions will lead us to revisit many of the topics that occupied us in the finite state case, such as simulation, distribution dynamics, stability, ergodicity, etc.

Note: For some people, the term "Markov chain" always refers to a process with a finite or discrete state space. We follow the mainstream mathematical literature (e.g., [MT09]) in using the term to refer to any discrete **time** Markov process.

Let's begin with some imports:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from scipy.stats import lognorm, beta
from quantecon import LAE
from scipy.stats import norm, gaussian_kde
```

2.2 The Density Case

You are probably aware that some distributions can be represented by densities and some cannot.

(For example, distributions on the real numbers $\mathbb{R}$ that put positive probability on individual points have no density representation)

We are going to start our analysis by looking at Markov chains where the one-step transition probabilities have density representations.

The benefit is that the density case offers a very direct parallel to the finite case in terms of notation and intuition.

Once we've built some intuition we'll cover the general case.

2.2.1 Definitions and Basic Properties

In our lecture on finite Markov chains, we studied discrete-time Markov chains that evolve on a finite state space S.

In this setting, the dynamics of the model are described by a stochastic matrix — a nonnegative square matrix $P = P[i, j]$ such that each row $P[i, \cdot]$ sums to one.

The interpretation of P is that $P[i, j]$ represents the probability of transitioning from state i to state j in one unit of time.

In symbols,

$$\mathbb{P}\{X_{t+1} = j \,|\, X_t = i\} = P[i, j]$$

Equivalently,

- P can be thought of as a family of distributions $P[i, \cdot]$, one for each $i \in S$
- $P[i, \cdot]$ is the distribution of X_{t+1} given $X_t = i$

(As you probably recall, when using NumPy arrays, $P[i, \cdot]$ is expressed as `P[i,:]`)

In this section, we'll allow S to be a subset of $\mathbb{R}$, such as

- $\mathbb{R}$ itself
- the positive reals $(0, \infty)$
- a bounded interval (a, b)

The family of discrete distributions $P[i, \cdot]$ will be replaced by a family of densities $p(x, \cdot)$, one for each $x \in S$.

Analogous to the finite state case, $p(x, \cdot)$ is to be understood as the distribution (density) of X_{t+1} given $X_t = x$.

More formally, a *stochastic kernel on* S is a function $p \colon S \times S \to \mathbb{R}$ with the property that

1. $p(x, y) \geq 0$ for all $x, y \in S$
2. $\int p(x, y) dy = 1$ for all $x \in S$

(Integrals are over the whole space unless otherwise specified)

For example, let $S = \mathbb{R}$ and consider the particular stochastic kernel p_w defined by

$$p_w(x, y) := \frac{1}{\sqrt{2\pi}} \exp\left\{ - \frac{(y - x)^2}{2} \right\} \tag{1}$$

What kind of model does p_w represent?

The answer is, the (normally distributed) random walk

$$X_{t+1} = X_t + \xi_{t+1} \quad \text{where} \quad \{\xi_t\} \overset{\text{IID}}{\sim} N(0, 1) \tag{2}$$

To see this, let's find the stochastic kernel p corresponding to (2).

Recall that $p(x, \cdot)$ represents the distribution of X_{t+1} given $X_t = x$.

Letting $X_t = x$ in (2) and considering the distribution of X_{t+1}, we see that $p(x, \cdot) = N(x, 1)$.

In other words, p is exactly p_w, as defined in (1).

2.2.2 Connection to Stochastic Difference Equations

In the previous section, we made the connection between stochastic difference equation (2) and stochastic kernel (1).

In economics and time-series analysis we meet stochastic difference equations of all different shapes and sizes.

It will be useful for us if we have some systematic methods for converting stochastic difference equations into stochastic kernels.

To this end, consider the generic (scalar) stochastic difference equation given by

$$X_{t+1} = \mu(X_t) + \sigma(X_t)\, \xi_{t+1} \tag{3}$$

Here we assume that

- $\{\xi_t\} \overset{\text{IID}}{\sim} \phi$, where ϕ is a given density on $\mathbb{R}$
- μ and σ are given functions on S, with $\sigma(x) > 0$ for all x

Example 1: The random walk (2) is a special case of (3), with $\mu(x) = x$ and $\sigma(x) = 1$.

Example 2: Consider the ARCH model

$$X_{t+1} = \alpha X_t + \sigma_t\, \xi_{t+1}, \qquad \sigma_t^2 = \beta + \gamma X_t^2, \qquad \beta, \gamma > 0$$

Alternatively, we can write the model as

$$X_{t+1} = \alpha X_t + (\beta + \gamma X_t^2)^{1/2} \xi_{t+1} \tag{4}$$

This is a special case of (3) with $\mu(x) = \alpha x$ and $\sigma(x) = (\beta + \gamma x^2)^{1/2}$.

Example 3: With stochastic production and a constant savings rate, the one-sector neoclassical growth model leads to a law of motion for capital per worker such as

$$k_{t+1} = sA_{t+1}f(k_t) + (1-\delta)k_t \tag{5}$$

Here

- s is the rate of savings
- A_{t+1} is a production shock
  - The $t+1$ subscript indicates that A_{t+1} is not visible at time t
- δ is a depreciation rate
- $f \colon \mathbb{R}_+ \to \mathbb{R}_+$ is a production function satisfying $f(k) > 0$ whenever $k > 0$

(The fixed savings rate can be rationalized as the optimal policy for a particular set of technologies and preferences (see [LS18], section 3.1.2), although we omit the details here).

Equation (5) is a special case of (3) with $\mu(x) = (1-\delta)x$ and $\sigma(x) = sf(x)$.

Now let's obtain the stochastic kernel corresponding to the generic model (3).

To find it, note first that if U is a random variable with density f_U, and $V = a + bU$ for some constants a, b with $b > 0$, then the density of V is given by

$$f_V(v) = \frac{1}{b} f_U\left(\frac{v-a}{b}\right) \tag{6}$$

(The proof is *below*. For a multidimensional version see EDTC, theorem 8.1.3).

Taking (6) as given for the moment, we can obtain the stochastic kernel p for (3) by recalling that $p(x, \cdot)$ is the conditional density of X_{t+1} given $X_t = x$.

In the present case, this is equivalent to stating that $p(x, \cdot)$ is the density of $Y := \mu(x) + \sigma(x)\,\xi_{t+1}$ when $\xi_{t+1} \sim \phi$.

Hence, by (6),

$$p(x, y) = \frac{1}{\sigma(x)} \phi\left(\frac{y - \mu(x)}{\sigma(x)}\right) \tag{7}$$

For example, the growth model in (5) has stochastic kernel

$$p(x, y) = \frac{1}{sf(x)} \phi\left(\frac{y - (1-\delta)x}{sf(x)}\right) \tag{8}$$

where ϕ is the density of A_{t+1}.

(Regarding the state space S for this model, a natural choice is $(0, \infty)$ — in which case $\sigma(x) = sf(x)$ is strictly positive for all s as required)

2.2.3 Distribution Dynamics

In this section of our lecture on **finite** Markov chains, we asked the following question: If

1. $\{X_t\}$ is a Markov chain with stochastic matrix P
2. the distribution of X_t is known to be ψ_t

then what is the distribution of X_{t+1}?

Letting ψ_{t+1} denote the distribution of X_{t+1}, the answer we gave was that

$$\psi_{t+1}[j] = \sum_{i \in S} P[i,j] \psi_t[i]$$

This intuitive equality states that the probability of being at j tomorrow is the probability of visiting i today and then going on to j, summed over all possible i.

In the density case, we just replace the sum with an integral and probability mass functions with densities, yielding

$$\psi_{t+1}(y) = \int p(x,y) \psi_t(x) \, dx, \qquad \forall y \in S \tag{9}$$

It is convenient to think of this updating process in terms of an operator.

(An operator is just a function, but the term is usually reserved for a function that sends functions into functions)

Let $\mathscr{D}$ be the set of all densities on S, and let P be the operator from $\mathscr{D}$ to itself that takes density ψ and sends it into new density ψP, where the latter is defined by

$$(\psi P)(y) = \int p(x,y) \psi(x) dx \tag{10}$$

This operator is usually called the *Markov operator* corresponding to p

Note: Unlike most operators, we write P to the right of its argument, instead of to the left (i.e., ψP instead of $P \psi$). This is a common convention, with the intention being to maintain the parallel with the finite case — see here

With this notation, we can write (9) more succinctly as $\psi_{t+1}(y) = (\psi_t P)(y)$ for all y, or, dropping the y and letting "$=$" indicate equality of functions,

$$\psi_{t+1} = \psi_t P \tag{11}$$

Equation (11) tells us that if we specify a distribution for ψ_0, then the entire sequence of future distributions can be obtained by iterating with P.

It's interesting to note that (11) is a deterministic difference equation.

Thus, by converting a stochastic difference equation such as (3) into a stochastic kernel p and hence an operator P, we convert a stochastic difference equation into a deterministic one (albeit in a much higher dimensional space).

Note: Some people might be aware that discrete Markov chains are in fact a special case of the continuous Markov chains we have just described. The reason is that probability mass functions are densities with respect to the counting measure.

2.2.4 Computation

To learn about the dynamics of a given process, it's useful to compute and study the sequences of densities generated by the model.

One way to do this is to try to implement the iteration described by (10) and (11) using numerical integration.

However, to produce ψP from ψ via (10), you would need to integrate at every y, and there is a continuum of such y.

Another possibility is to discretize the model, but this introduces errors of unknown size.

A nicer alternative in the present setting is to combine simulation with an elegant estimator called the *look-ahead* estimator.

Let's go over the ideas with reference to the growth model *discussed above*, the dynamics of which we repeat here for convenience:

$$k_{t+1} = sA_{t+1}f(k_t) + (1-\delta)k_t \tag{12}$$

Our aim is to compute the sequence $\{\psi_t\}$ associated with this model and fixed initial condition ψ_0.

To approximate ψ_t by simulation, recall that, by definition, ψ_t is the density of k_t given $k_0 \sim \psi_0$.

If we wish to generate observations of this random variable, all we need to do is

1. draw k_0 from the specified initial condition ψ_0
2. draw the shocks $A_1, \ldots, A_t$ from their specified density ϕ
3. compute k_t iteratively via (12)

If we repeat this n times, we get n independent observations $k_t^1, \ldots, k_t^n$.

With these draws in hand, the next step is to generate some kind of representation of their distribution ψ_t.

A naive approach would be to use a histogram, or perhaps a smoothed histogram using SciPy's `gaussian_kde` function.

However, in the present setting, there is a much better way to do this, based on the look-ahead estimator.

With this estimator, to construct an estimate of ψ_t, we actually generate n observations of k_{t-1}, rather than k_t.

Now we take these n observations $k_{t-1}^1, \ldots, k_{t-1}^n$ and form the estimate

$$\psi_t^n(y) = \frac{1}{n}\sum_{i=1}^n p(k_{t-1}^i, y) \tag{13}$$

where p is the growth model stochastic kernel in (8).

What is the justification for this slightly surprising estimator?

The idea is that, by the strong law of large numbers,

$$\frac{1}{n}\sum_{i=1}^n p(k_{t-1}^i, y) \to \mathbb{E}p(k_{t-1}^i, y) = \int p(x,y)\psi_{t-1}(x)\,dx = \psi_t(y)$$

with probability one as $n \to \infty$.

Here the first equality is by the definition of ψ_{t-1}, and the second is by (9).

We have just shown that our estimator $\psi_t^n(y)$ in (13) converges almost surely to $\psi_t(y)$, which is just what we want to compute.

In fact, much stronger convergence results are true (see, for example, this paper).

2.2.5 Implementation

A class called `LAE` for estimating densities by this technique can be found in lae.py.

Given our use of the `__call__` method, an instance of `LAE` acts as a callable object, which is essentially a function that can store its own data (see this discussion).

This function returns the right-hand side of (13) using

- the data and stochastic kernel that it stores as its instance data
- the value y as its argument

The function is vectorized, in the sense that if psi is such an instance and y is an array, then the call psi(y) acts elementwise.

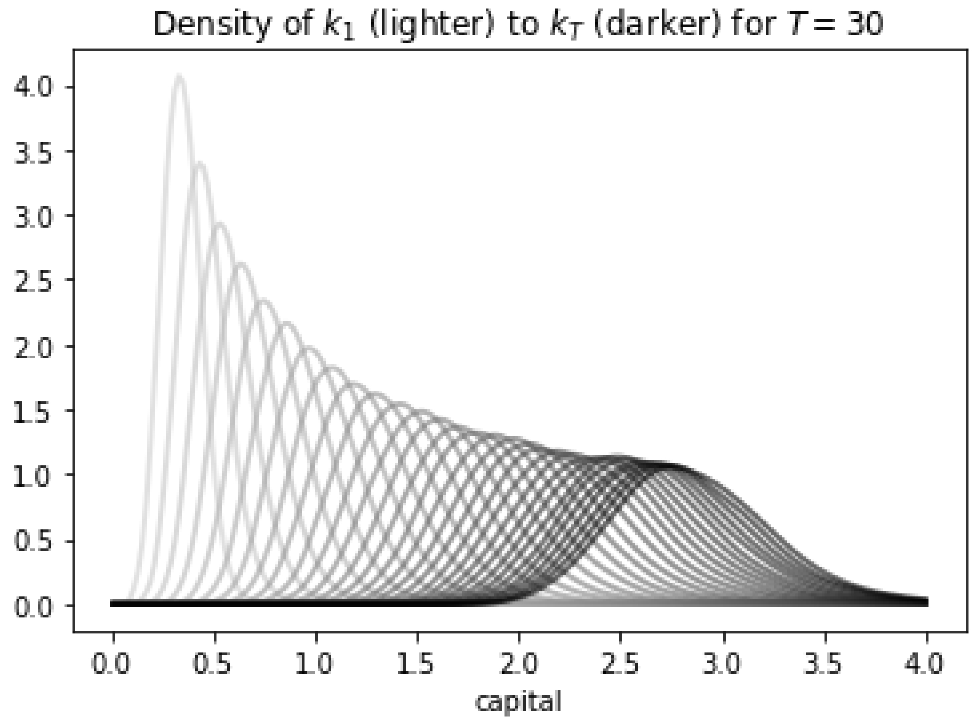
(This is the reason that we reshaped X and y inside the class — to make vectorization work)

Because the implementation is fully vectorized, it is about as efficient as it would be in C or Fortran.

2.2.6 Example

The following code is an example of usage for the stochastic growth model *described above*

```
# == Define parameters == #
s = 0.2
δ = 0.1
a_σ = 0.4                    # A = exp(B) where B ~ N(0, a_σ)
α = 0.4                      # We set f(k) = k**α
ψ_0 = beta(5, 5, scale=0.5)  # Initial distribution
ϕ = lognorm(a_σ)


def p(x, y):
    """
    Stochastic kernel for the growth model with Cobb-Douglas production.
    Both x and y must be strictly positive.
    """
    d = s * x**α
    return ϕ.pdf((y - (1 - δ) * x) / d) / d

n = 10000    # Number of observations at each date t
T = 30       # Compute density of k_t at 1,...,T+1

# == Generate matrix s.t. t-th column is n observations of k_t == #
k = np.empty((n, T))
A = ϕ.rvs((n, T))
k[:, 0] = ψ_0.rvs(n)  # Draw first column from initial distribution
for t in range(T-1):
    k[:, t+1] = s * A[:, t] * k[:, t]**α + (1 - δ) * k[:, t]

# == Generate T instances of LAE using this data, one for each date t == #
laes = [LAE(p, k[:, t]) for t in range(T)]

# == Plot == #
fig, ax = plt.subplots()
ygrid = np.linspace(0.01, 4.0, 200)
greys = [str(g) for g in np.linspace(0.0, 0.8, T)]
greys.reverse()
for ψ, g in zip(laes, greys):
    ax.plot(ygrid, ψ(ygrid), color=g, lw=2, alpha=0.6)
ax.set_xlabel('capital')
ax.set_title(f'Density of $k_1$ (lighter) to $k_T$ (darker) for $T={T}$')
plt.show()
```



The figure shows part of the density sequence $\{\psi_t\}$, with each density computed via the look-ahead estimator.

Notice that the sequence of densities shown in the figure seems to be converging — more on this in just a moment.

Another quick comment is that each of these distributions could be interpreted as a cross-sectional distribution (recall this discussion).

2.3 Beyond Densities

Up until now, we have focused exclusively on continuous state Markov chains where all conditional distributions $p(x, \cdot)$ are densities.

As discussed above, not all distributions can be represented as densities.

If the conditional distribution of X_{t+1} given $X_t = x$ **cannot** be represented as a density for some $x \in S$, then we need a slightly different theory.

The ultimate option is to switch from densities to probability measures, but not all readers will be familiar with measure theory.

We can, however, construct a fairly general theory using distribution functions.

2.3.1 Example and Definitions

To illustrate the issues, recall that Hopenhayn and Rogerson [HR93] study a model of firm dynamics where individual firm productivity follows the exogenous process

$$X_{t+1} = a + \rho X_t + \xi_{t+1}, \quad \text{where} \quad \{\xi_t\} \overset{\text{IID}}{\sim} N(0, \sigma^2)$$

As is, this fits into the density case we treated above.

However, the authors wanted this process to take values in $[0, 1]$, so they added boundaries at the endpoints 0 and 1.

One way to write this is

$$X_{t+1} = h(a + \rho X_t + \xi_{t+1}) \quad \text{where} \quad h(x) := x\,\mathbf{1}\{0 \leq x \leq 1\} + \mathbf{1}\{x > 1\}$$

If you think about it, you will see that for any given $x \in [0, 1]$, the conditional distribution of X_{t+1} given $X_t = x$ puts positive probability mass on 0 and 1.

Hence it cannot be represented as a density.

What we can do instead is use cumulative distribution functions (cdfs).

To this end, set

$$G(x, y) := \mathbb{P}\{h(a + \rho x + \xi_{t+1}) \leq y\} \qquad (0 \leq x, y \leq 1)$$

This family of cdfs $G(x, \cdot)$ plays a role analogous to the stochastic kernel in the density case.

The distribution dynamics in (9) are then replaced by

$$F_{t+1}(y) = \int G(x, y) F_t(dx) \tag{14}$$

Here F_t and F_{t+1} are cdfs representing the distribution of the current state and next period state.

The intuition behind (14) is essentially the same as for (9).

2.3.2 Computation

If you wish to compute these cdfs, you cannot use the look-ahead estimator as before.

Indeed, you should not use any density estimator, since the objects you are estimating/computing are not densities.

One good option is simulation as before, combined with the empirical distribution function.

2.4 Stability

In our lecture on finite Markov chains, we also studied stationarity, stability and ergodicity.

Here we will cover the same topics for the continuous case.

We will, however, treat only the density case (as in *this section*), where the stochastic kernel is a family of densities.

The general case is relatively similar — references are given below.

2.4.1 Theoretical Results

Analogous to the finite case, given a stochastic kernel p and corresponding Markov operator as defined in (10), a density ψ^* on S is called *stationary* for P if it is a fixed point of the operator P.

In other words,

$$\psi^*(y) = \int p(x, y)\psi^*(x)\, dx, \qquad \forall y \in S \tag{15}$$

As with the finite case, if ψ^* is stationary for P, and the distribution of X_0 is ψ^*, then, in view of (11), X_t will have this same distribution for all t.

Hence ψ^* is the stochastic equivalent of a steady state.

In the finite case, we learned that at least one stationary distribution exists, although there may be many.

When the state space is infinite, the situation is more complicated.

Even existence can fail very easily.

For example, the random walk model has no stationary density (see, e.g., EDTC, p. 210).

However, there are well-known conditions under which a stationary density ψ^* exists.

With additional conditions, we can also get a unique stationary density ($\psi \in \mathscr{D}$ and $\psi = \psi P \implies \psi = \psi^*$), and also global convergence in the sense that

$$\forall\, \psi \in \mathscr{D}, \quad \psi P^t \to \psi^* \quad \text{as} \quad t \to \infty \tag{16}$$

This combination of existence, uniqueness and global convergence in the sense of (16) is often referred to as *global stability*.

Under very similar conditions, we get *ergodicity*, which means that

$$\frac{1}{n}\sum_{t=1}^{n} h(X_t) \to \int h(x)\psi^*(x)dx \quad \text{as } n \to \infty \tag{17}$$

for any (measurable) function $h \colon S \to \mathbb{R}$ such that the right-hand side is finite.

Note that the convergence in (17) does not depend on the distribution (or value) of X_0.

This is actually very important for simulation — it means we can learn about ψ^* (i.e., approximate the right-hand side of (17) via the left-hand side) without requiring any special knowledge about what to do with X_0.

So what are these conditions we require to get global stability and ergodicity?

In essence, it must be the case that

1. Probability mass does not drift off to the "edges" of the state space.
2. Sufficient "mixing" obtains.

For one such set of conditions see theorem 8.2.14 of EDTC.

In addition

- [SLP89] contains a classic (but slightly outdated) treatment of these topics.
- From the mathematical literature, [LM94] and [MT09] give outstanding in-depth treatments.
- Section 8.1.2 of EDTC provides detailed intuition, and section 8.3 gives additional references.
- EDTC, section 11.3.4 provides a specific treatment for the growth model we considered in this lecture.

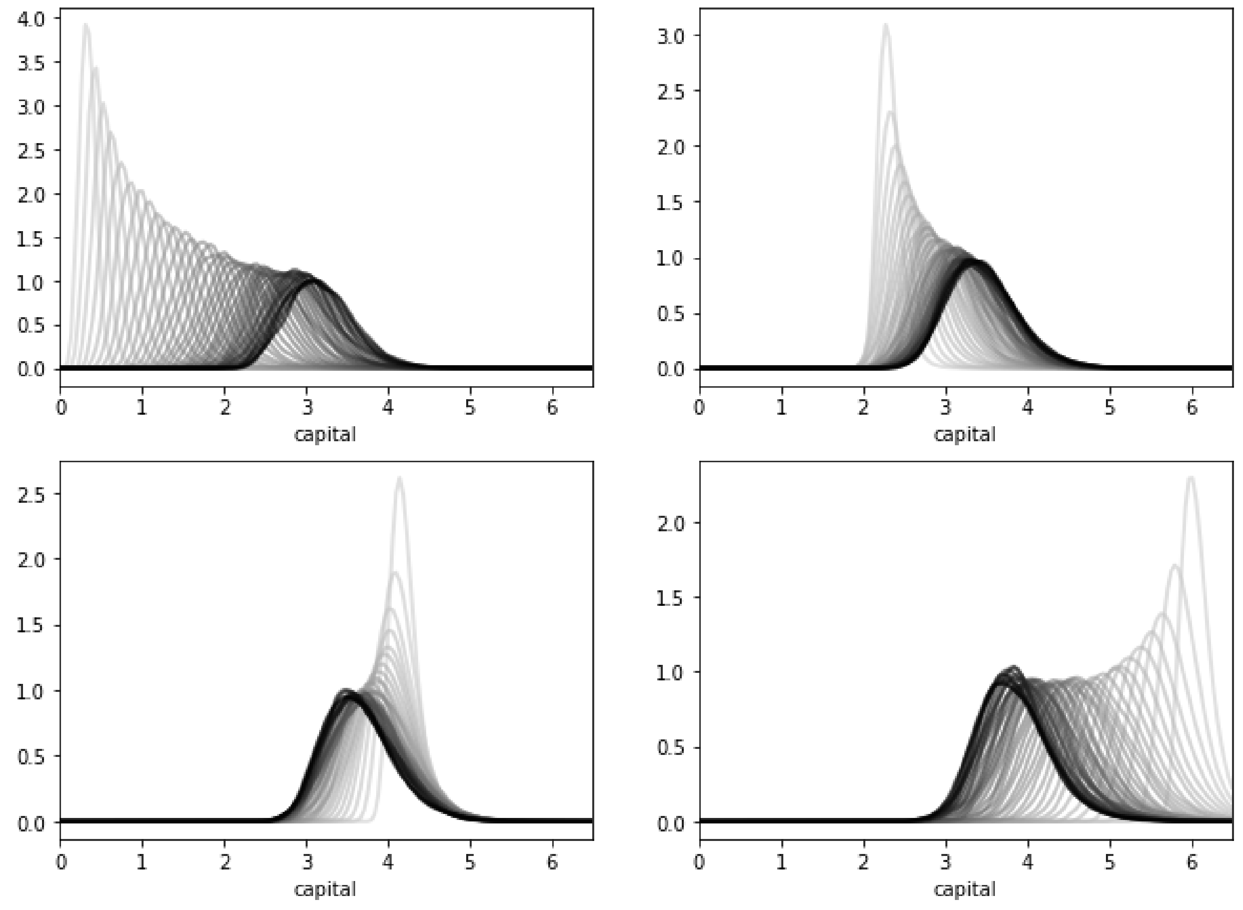
2.4.2 An Example of Stability

As stated above, the *growth model treated here* is stable under mild conditions on the primitives.

- See EDTC, section 11.3.4 for more details.

We can see this stability in action — in particular, the convergence in (16) — by simulating the path of densities from various initial conditions.

Here is such a figure.



All sequences are converging towards the same limit, regardless of their initial condition.

The details regarding initial conditions and so on are given in *this exercise*, where you are asked to replicate the figure.

2.4.3 Computing Stationary Densities

In the preceding figure, each sequence of densities is converging towards the unique stationary density ψ^*.

Even from this figure, we can get a fair idea what ψ^* looks like, and where its mass is located.

However, there is a much more direct way to estimate the stationary density, and it involves only a slight modification of the look-ahead estimator.

Let's say that we have a model of the form (3) that is stable and ergodic.

Let p be the corresponding stochastic kernel, as given in (7).

To approximate the stationary density ψ^*, we can simply generate a long time-series $X_0, X_1, \dots, X_n$ and estimate ψ^* via

$$\psi_n^*(y) = \frac{1}{n} \sum_{t=1}^{n} p(X_t, y) \tag{18}$$

This is essentially the same as the look-ahead estimator (13), except that now the observations we generate are a single time-series, rather than a cross-section.

The justification for (18) is that, with probability one as $n \to \infty$,

$$\frac{1}{n} \sum_{t=1}^{n} p(X_t, y) \to \int p(x, y) \psi^*(x) \, dx = \psi^*(y)$$

where the convergence is by (17) and the equality on the right is by (15).

The right-hand side is exactly what we want to compute.

On top of this asymptotic result, it turns out that the rate of convergence for the look-ahead estimator is very good.

The first exercise helps illustrate this point.

2.5 Exercises

2.5.1 Exercise 1

Consider the simple threshold autoregressive model

$$X_{t+1} = \theta |X_t| + (1 - \theta^2)^{1/2} \xi_{t+1} \qquad \text{where} \quad \{\xi_t\} \overset{\text{IID}}{\sim} N(0, 1) \tag{19}$$

This is one of those rare nonlinear stochastic models where an analytical expression for the stationary density is available.

In particular, provided that $|\theta| < 1$, there is a unique stationary density ψ^* given by

$$\psi^*(y) = 2 \, \phi(y) \, \Phi \left[\frac{\theta y}{(1 - \theta^2)^{1/2}} \right] \tag{20}$$

Here ϕ is the standard normal density and Φ is the standard normal cdf.

As an exercise, compute the look-ahead estimate of ψ^*, as defined in (18), and compare it with ψ^* in (20) to see whether they are indeed close for large n.

In doing so, set $\theta = 0.8$ and $n = 500$.

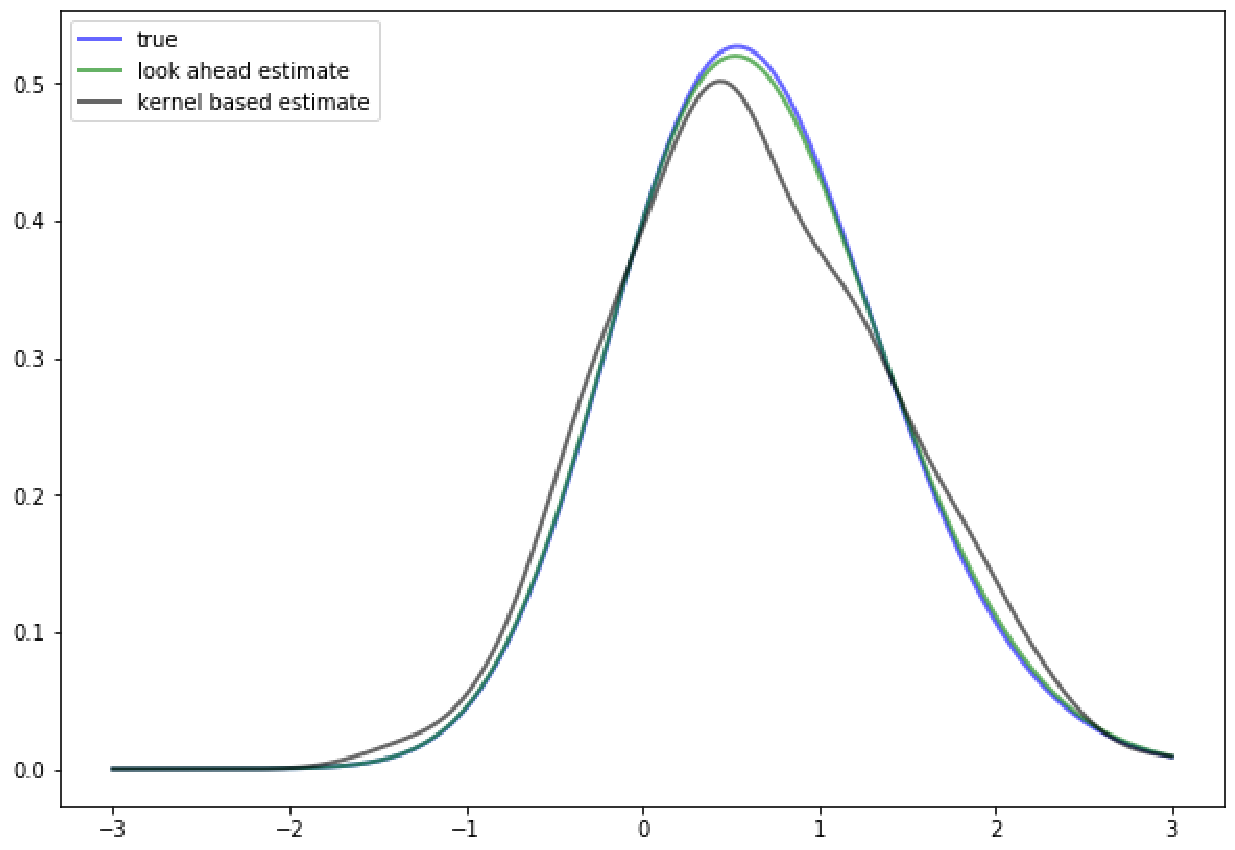
The next figure shows the result of such a computation

The additional density (black line) is a nonparametric kernel density estimate, added to the solution for illustration.

(You can try to replicate it before looking at the solution if you want to)

As you can see, the look-ahead estimator is a much tighter fit than the kernel density estimator.

If you repeat the simulation you will see that this is consistently the case.


true
look ahead estimate
kernel based estimate
0.0
0.1
0.2
0.3
0.4
0.5
−3
−2
−1
0
1
2
3

2.5.2 Exercise 2

Replicate the figure on global convergence *shown above*.

The densities come from the stochastic growth model treated *at the start of the lecture*.

Begin with the code found *above*.

Use the same parameters.

For the four initial distributions, use the shifted beta distributions

```
ψ_0 = beta(5, 5, scale=0.5, loc=i*2)
```

2.5.3 Exercise 3

A common way to compare distributions visually is with boxplots.

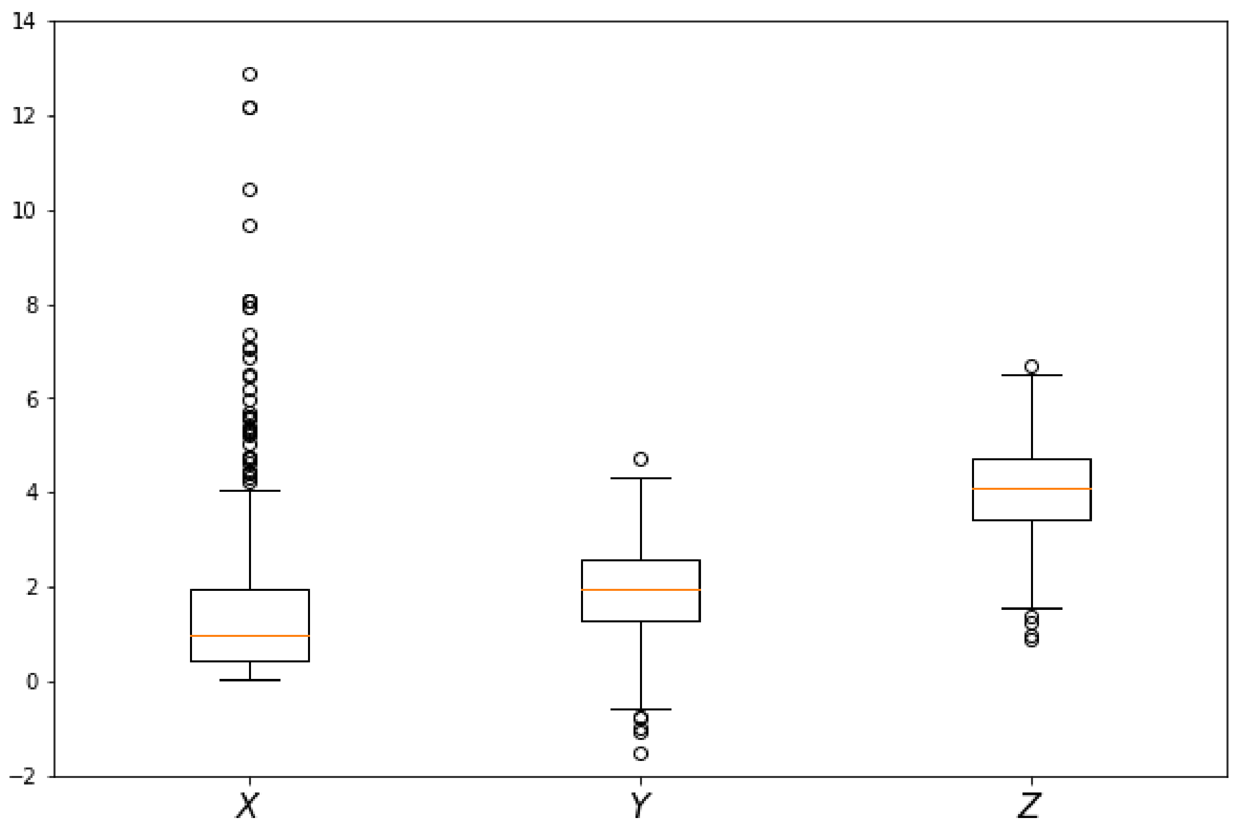
To illustrate, let's generate three artificial data sets and compare them with a boxplot.

The three data sets we will use are:

$$\{X_1, \ldots, X_n\} \sim LN(0,1), \;\; \{Y_1, \ldots, Y_n\} \sim N(2,1), \quad \text{and} \;\; \{Z_1, \ldots, Z_n\} \sim N(4,1),$$

Here is the code and figure:

```
n = 500
x = np.random.randn(n)        # N(0, 1)
x = np.exp(x)                 # Map x to lognormal
y = np.random.randn(n) + 2.0  # N(2, 1)
z = np.random.randn(n) + 4.0  # N(4, 1)

fig, ax = plt.subplots(figsize=(10, 6.6))
ax.boxplot([x, y, z])
ax.set_xticks((1, 2, 3))
ax.set_ylim(-2, 14)
ax.set_xticklabels(('$X$', '$Y$', '$Z$'), fontsize=16)
plt.show()
```



Each data set is represented by a box, where the top and bottom of the box are the third and first quartiles of the data, and the red line in the center is the median.

The boxes give some indication as to

- the location of probability mass for each sample
- whether the distribution is right-skewed (as is the lognormal distribution), etc

Now let's put these ideas to use in a simulation.

Consider the threshold autoregressive model in (19).

We know that the distribution of X_t will converge to (20) whenever $|\theta| < 1$.

Let's observe this convergence from different initial conditions using boxplots.

In particular, the exercise is to generate J boxplot figures, one for each initial condition X_0 in

```
initial_conditions = np.linspace(8, 0, J)
```

For each X_0 in this set,

1. Generate k time-series of length n, each starting at X_0 and obeying (19).
2. Create a boxplot representing n distributions, where the t-th distribution shows the k observations of X_t.

Use $\theta = 0.9, n = 20, k = 5000, J = 8$

2.6 Solutions

2.6.1 Exercise 1

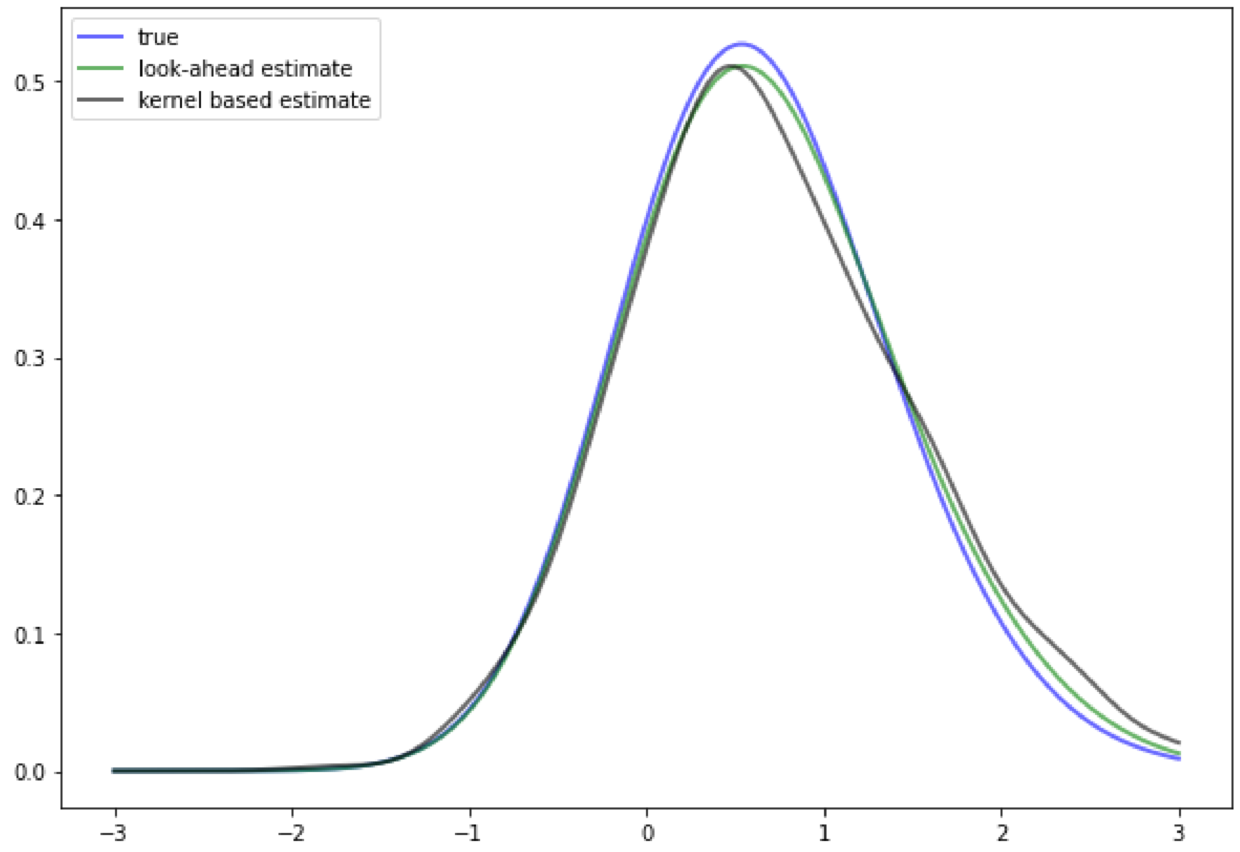
Look-ahead estimation of a TAR stationary density, where the TAR model is

$$X_{t+1} = \theta |X_t| + (1-\theta^2)^{1/2} \xi_{t+1}$$

and $\xi_t \sim N(0,1)$.

Try running at `n = 10, 100, 1000, 10000` to get an idea of the speed of convergence

```python
ϕ = norm()
n = 500
θ = 0.8
# == Frequently used constants == #
d = np.sqrt(1 - θ**2)
δ = θ / d

def ψ_star(y):
    "True stationary density of the TAR Model"
    return 2 * norm.pdf(y) * norm.cdf(δ * y)

def p(x, y):
    "Stochastic kernel for the TAR model."
    return ϕ.pdf((y - θ * np.abs(x)) / d) / d

Z = ϕ.rvs(n)
X = np.empty(n)
for t in range(n-1):
    X[t+1] = θ * np.abs(X[t]) + d * Z[t]
ψ_est = LAE(p, X)
k_est = gaussian_kde(X)

fig, ax = plt.subplots(figsize=(10, 7))
ys = np.linspace(-3, 3, 200)
ax.plot(ys, ψ_star(ys), 'b-', lw=2, alpha=0.6, label='true')
ax.plot(ys, ψ_est(ys), 'g-', lw=2, alpha=0.6, label='look-ahead estimate')
ax.plot(ys, k_est(ys), 'k-', lw=2, alpha=0.6, label='kernel based estimate')
ax.legend(loc='upper left')
plt.show()
```



2.6.2 Exercise 2

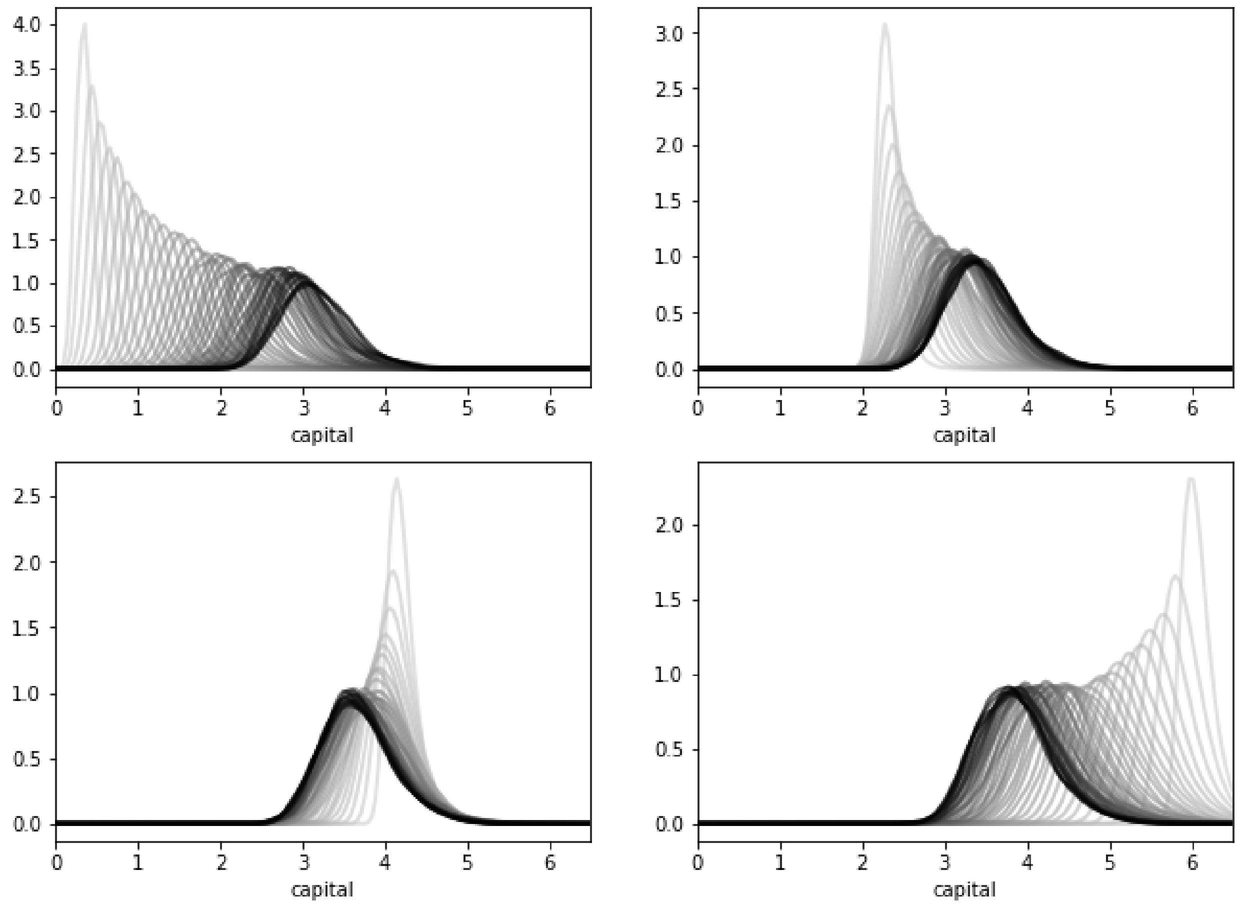
Here's one program that does the job

```
# == Define parameters == #
s = 0.2
δ = 0.1
a_σ = 0.4                                  # A = exp(B) where B ~ N(0, a_σ)
α = 0.4                                    # f(k) = k**α

ϕ = lognorm(a_σ)

def p(x, y):
    "Stochastic kernel, vectorized in x.  Both x and y must be positive."
    d = s * x**α
    return ϕ.pdf((y - (1 - δ) * x) / d) / d

n = 1000                             # Number of observations at each date t
T = 40                               # Compute density of k_t at 1,...,T

fig, axes = plt.subplots(2, 2, figsize=(11, 8))
axes = axes.flatten()
xmax = 6.5

for i in range(4):
    ax = axes[i]
```

(continues on next page)

(continued from previous page)

```
    ax.set_xlim(0, xmax)
    ψ_0 = beta(5, 5, scale=0.5, loc=i*2)  # Initial distribution

    # == Generate matrix s.t. t-th column is n observations of k_t == #
    k = np.empty((n, T))
    A = ϕ.rvs((n, T))
    k[:, 0] = ψ_0.rvs(n)
    for t in range(T-1):
        k[:, t+1] = s * A[:,t] * k[:, t]**α + (1 - δ) * k[:, t]

    # == Generate T instances of lae using this data, one for each t == #
    laes = [LAE(p, k[:, t]) for t in range(T)]

    ygrid = np.linspace(0.01, xmax, 150)
    greys = [str(g) for g in np.linspace(0.0, 0.8, T)]
    greys.reverse()
    for ψ, g in zip(laes, greys):
        ax.plot(ygrid, ψ(ygrid), color=g, lw=2, alpha=0.6)
    ax.set_xlabel('capital')
plt.show()
```

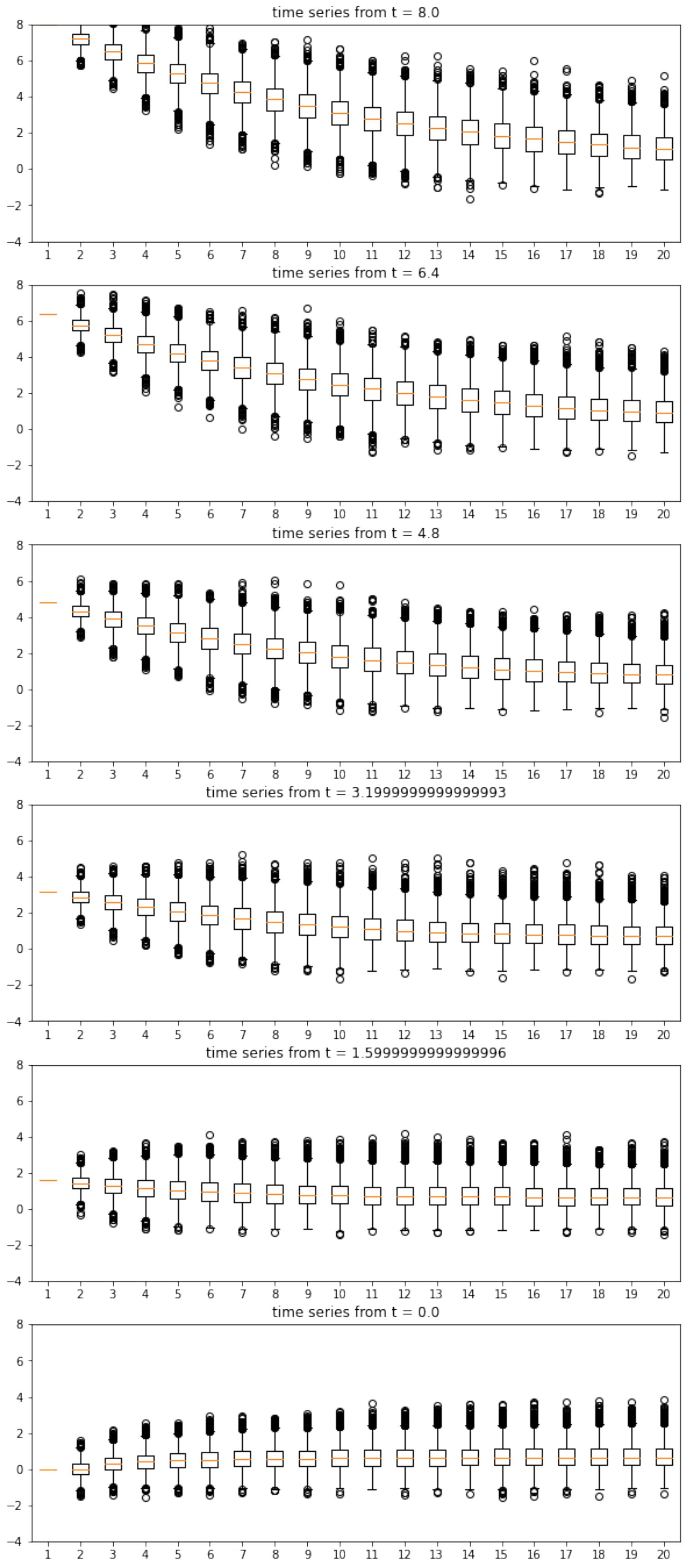


2.6.3 Exercise 3

Here's a possible solution.

Note the way we use vectorized code to simulate the k time series for one boxplot all at once

```python
n = 20
k = 5000
J = 6

θ = 0.9
d = np.sqrt(1 - θ**2)
δ = θ / d

fig, axes = plt.subplots(J, 1, figsize=(10, 4*J))
initial_conditions = np.linspace(8, 0, J)
X = np.empty((k, n))

for j in range(J):

    axes[j].set_ylim(-4, 8)
    axes[j].set_title(f'time series from t = {initial_conditions[j]}')

    Z = np.random.randn(k, n)
    X[:, 0] = initial_conditions[j]
    for t in range(1, n):
        X[:, t] = θ * np.abs(X[:, t-1]) + d * Z[:, t]
    axes[j].boxplot(X)

plt.show()
```


time series from t = 8.0
time series from t = 6.4
time series from t = 4.8
time series from t = 3.1999999999999993
time series from t = 1.5999999999999996
time series from t = 0.0

2.7 Appendix

Here's the proof of (6).

Let F_U and F_V be the cumulative distributions of U and V respectively.

By the definition of V, we have $F_V(v) = \mathbb{P}\{a + bU \leq v\} = \mathbb{P}\{U \leq (v - a)/b\}$.

In other words, $F_V(v) = F_U((v - a)/b)$.

Differentiating with respect to v yields (6).

CHAPTER

THREE

REVERSE ENGINEERING A LA MUTH

Contents

- *Reverse Engineering a la Muth*
  - *Friedman (1956) and Muth (1960)*
  - *A Process for Which Adaptive Expectations are Optimal*
  - *Some Useful State-Space Math*
  - *Estimates of Unobservables*
  - *Relation between Unobservable and Observable*
  - *MA and AR Representations*

In addition to what's in Anaconda, this lecture uses the quantecon library.

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

We'll also need the following imports:

```
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
import scipy.linalg as la

from quantecon import Kalman
from quantecon import LinearStateSpace
from scipy.stats import norm
np.set_printoptions(linewidth=120, precision=4, suppress=True)
```

This lecture uses the Kalman filter to reformulate John F. Muth's first paper [Mut60] about rational expectations.

Muth used *classical* prediction methods to reverse engineer a stochastic process that renders optimal Milton Friedman's [Fri56] "adaptive expectations" scheme.

3.1 Friedman (1956) and Muth (1960)

Milton Friedman [Fri56] (1956) posited that consumer's forecast their future disposable income with the adaptive expectations scheme

$$y^*_{t+i,t} = K \sum_{j=0}^{\infty} (1-K)^j y_{t-j} \tag{1}$$

where $K \in (0,1)$ and $y^*_{t+i,t}$ is a forecast of future y over horizon i.

Milton Friedman justified the **exponential smoothing** forecasting scheme (1) informally, noting that it seemed a plausible way to use past income to forecast future income.

In his first paper about rational expectations, John F. Muth [Mut60] reverse-engineered a univariate stochastic process $\{y_t\}_{t=-\infty}^{\infty}$ for which Milton Friedman's adaptive expectations scheme gives linear least forecasts of y_{t+j} for any horizon i.

Muth sought a setting and a sense in which Friedman's forecasting scheme is optimal.

That is, Muth asked for what optimal forecasting **question** is Milton Friedman's adaptive expectation scheme the **answer**.

Muth (1960) used classical prediction methods based on lag-operators and z-transforms to find the answer to his question.

Please see lectures *Classical Control with Linear Algebra* and *Classical Filtering and Prediction with Linear Algebra* for an introduction to the classical tools that Muth used.

Rather than using those classical tools, in this lecture we apply the Kalman filter to express the heart of Muth's analysis concisely.

The lecture First Look at Kalman Filter describes the Kalman filter.

We'll use limiting versions of the Kalman filter corresponding to what are called **stationary values** in that lecture.

3.2 A Process for Which Adaptive Expectations are Optimal

Suppose that an observable y_t is the sum of an unobserved random walk x_t and an IID shock $\epsilon_{2,t}$:

$$\begin{aligned} x_{t+1} &= x_t + \sigma_x \epsilon_{1,t+1} \\ y_t &= x_t + \sigma_y \epsilon_{2,t} \end{aligned} \tag{2}$$

where

$$\begin{bmatrix} \epsilon_{1,t+1} \\ \epsilon_{2,t} \end{bmatrix} \sim \mathcal{N}(0, I)$$

is an IID process.

Note: A property of the state-space representation (2) is that in general neither $\epsilon_{1,t}$ nor $\epsilon_{2,t}$ is in the space spanned by square-summable linear combinations of $y_t, y_{t-1}, \ldots$.

In general $\begin{bmatrix} \epsilon_{1,t} \\ \epsilon_{2t} \end{bmatrix}$ has more information about future y_{t+j}'s than is contained in $y_t, y_{t-1}, \ldots$.

We can use the asymptotic or stationary values of the Kalman gain and the one-step-ahead conditional state covariance matrix to compute a time-invariant *innovations representation*

$$\begin{aligned} \hat{x}_{t+1} &= \hat{x}_t + K a_t \\ y_t &= \hat{x}_t + a_t \end{aligned} \tag{3}$$

where $\hat{x}_t = E[x_t|y_{t-1}, y_{t-2}, ...]$ and $a_t = y_t - E[y_t|y_{t-1}, y_{t-2}, ...]$.

Note: A key property about an *innovations representation* is that a_t is in the space spanned by square summable linear combinations of $y_t, y_{t-1}, ...$.

For more ramifications of this property, see the lectures *Shock Non-Invertibility* and *Recursive Models of Dynamic Linear Economies*.

Later we'll stack these state-space systems (2) and (3) to display some classic findings of Muth.

But first, let's create an instance of the state-space system (2) then apply the quantecon `Kalman` class, then uses it to construct the associated "innovations representation"

```
# Make some parameter choices
# sigx/sigy are state noise std err and measurement noise std err
μ_0, σ_x, σ_y = 10, 1, 5

# Create a LinearStateSpace object
A, C, G, H = 1, σ_x, 1, σ_y
ss = LinearStateSpace(A, C, G, H, mu_0=μ_0)

# Set prior and initialize the Kalman type
x_hat_0, Σ_0 = 10, 1
kmuth = Kalman(ss, x_hat_0, Σ_0)

# Computes stationary values which we need for the innovation
# representation
S1, K1 = kmuth.stationary_values()

# Form innovation representation state-space
Ak, Ck, Gk, Hk = A, K1, G, 1

ssk = LinearStateSpace(Ak, Ck, Gk, Hk, mu_0=x_hat_0)
```

3.3 Some Useful State-Space Math

Now we want to map the time-invariant innovations representation (3) and the original state-space system (2) into a convenient form for deducing the impulse responses from the original shocks to the x_t and $\hat{x}_t$.

Putting both of these representations into a single state-space system is yet another application of the insight that "finding the state is an art".

We'll define a state vector and appropriate state-space matrices that allow us to represent both systems in one fell swoop.

Note that

$$a_t = x_t + \sigma_y \epsilon_{2,t} - \hat{x}_t$$

so that

$$\begin{aligned} \hat{x}_{t+1} &= \hat{x}_t + K(x_t + \sigma_y \epsilon_{2,t} - \hat{x}_t) \\ &= (1-K)\hat{x}_t + Kx_t + K\sigma_y \epsilon_{2,t} \end{aligned}$$

The stacked system

$$\begin{bmatrix} x_{t+1} \\ \hat{x}_{t+1} \\ \epsilon_{2,t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ K & (1-K) & K\sigma_y \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_t \\ \hat{x}_t \\ \epsilon_{2,t} \end{bmatrix} + \begin{bmatrix} \sigma_x & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \epsilon_{1,t+1} \\ \epsilon_{2,t+1} \end{bmatrix}$$

$$\begin{bmatrix} y_t \\ a_t \end{bmatrix} = \begin{bmatrix} 1 & 0 & \sigma_y \\ 1 & -1 & \sigma_y \end{bmatrix} \begin{bmatrix} x_t \\ \hat{x}_t \\ \epsilon_{2,t} \end{bmatrix}$$

is a state-space system that tells us how the shocks $\begin{bmatrix} \epsilon_{1,t+1} \\ \epsilon_{2,t+1} \end{bmatrix}$ affect states $\hat{x}_{t+1}, x_t$, the observable y_t, and the innovation a_t.

With this tool at our disposal, let's form the composite system and simulate it

```
# Create grand state-space for y_t, a_t as observed vars -- Use
# stacking trick above
Af = np.array([[ 1,      0,        0],
               [K1, 1 - K1, K1 * σ_y],
               [ 0,      0,        0]])
Cf = np.array([[σ_x,        0],
               [  0, K1 * σ_y],
               [  0,        1]])
Gf = np.array([[1,  0, σ_y],
               [1, -1, σ_y]])

μ_true, μ_prior = 10, 10
μ_f = np.array([μ_true, μ_prior, 0]).reshape(3, 1)

# Create the state-space
ssf = LinearStateSpace(Af, Cf, Gf, mu_0=μ_f)

# Draw observations of y from the state-space model
N = 50
xf, yf = ssf.simulate(N)

print(f"Kalman gain = {K1}")
print(f"Conditional variance = {S1}")
```

```
<ipython-input-4-49fa57350fa1>:3: VisibleDeprecationWarning: Creating an ndarray from␣
↪ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays␣
↪with different lengths or shapes) is deprecated. If you meant to do this, you must␣
↪specify 'dtype=object' when creating the ndarray
  Af = np.array([[ 1,      0,        0],
<ipython-input-4-49fa57350fa1>:6: VisibleDeprecationWarning: Creating an ndarray from␣
↪ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays␣
↪with different lengths or shapes) is deprecated. If you meant to do this, you must␣
↪specify 'dtype=object' when creating the ndarray
  Cf = np.array([[σ_x,        0],
```

```
Kalman gain = [[0.181]]
Conditional variance = [[5.5249]]
```

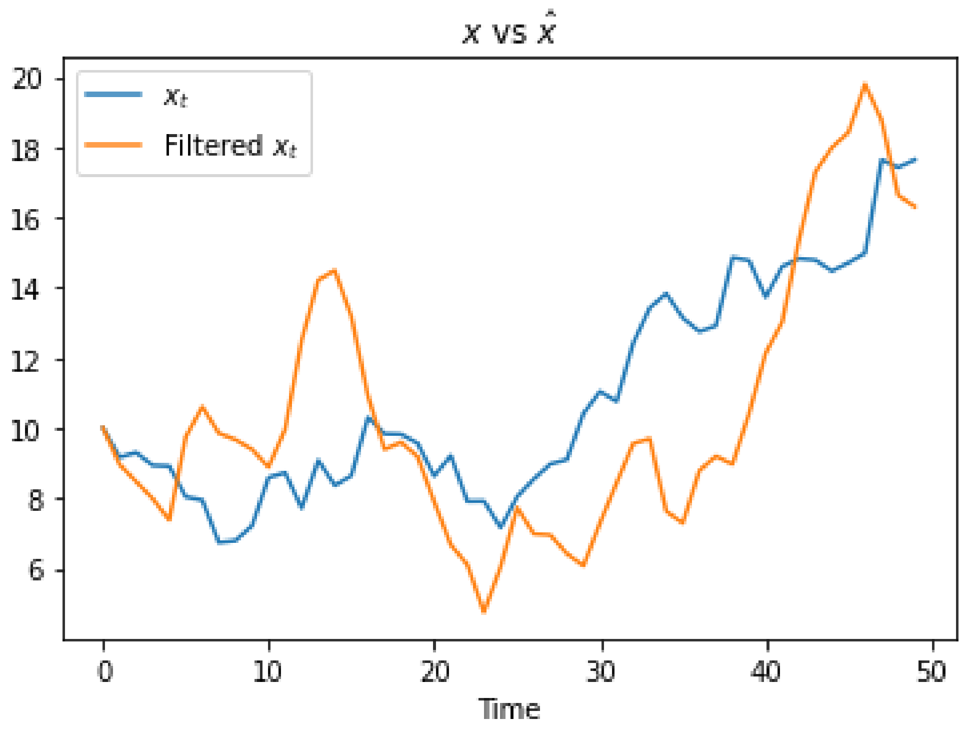
Now that we have simulated our joint system, we have x_t, $\hat{x}_t$, and y_t.

We can now investigate how these variables are related by plotting some key objects.

3.4 Estimates of Unobservables

First, let's plot the hidden state x_t and the filtered version $\hat{x}_t$ that is linear-least squares projection of x_t on the history $y_{t-1}, y_{t-2}, \cdots$

```
fig, ax = plt.subplots()
ax.plot(xf[0, :], label="$x_t$")
ax.plot(xf[1, :], label="Filtered $x_t$")
ax.legend()
ax.set_xlabel("Time")
ax.set_title(r"$x$ vs $\hat{x}$")
plt.show()
```



Note how x_t and $\hat{x}_t$ differ.

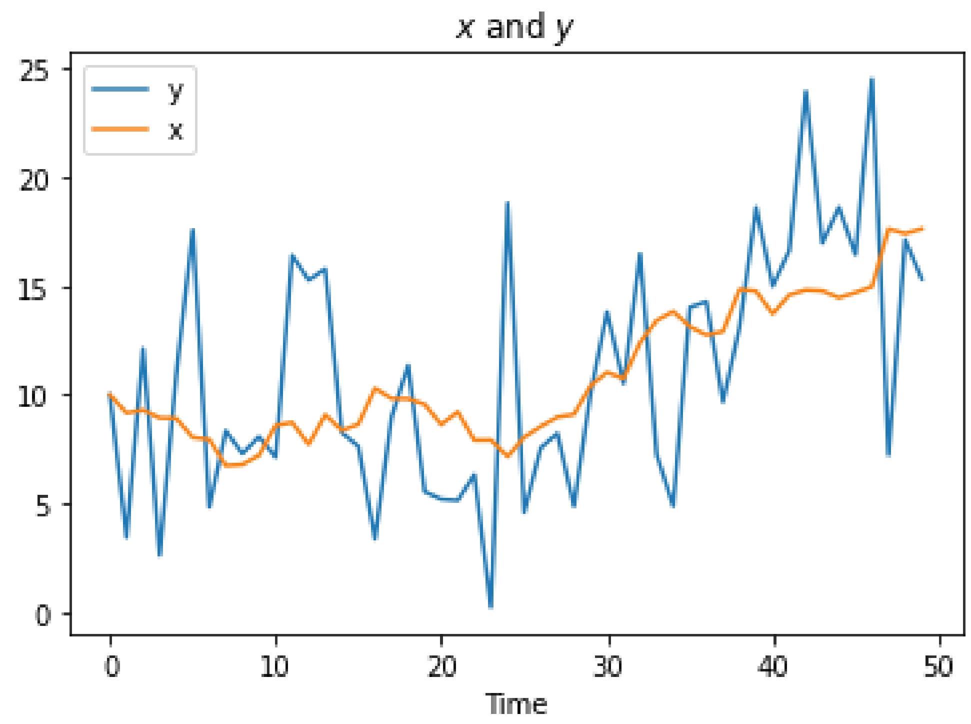
For Friedman, $\hat{x}_t$ and not x_t is the consumer's idea about her/his *permanent income*.

3.5 Relation between Unobservable and Observable

Now let's plot x_t and y_t.

Recall that y_t is just x_t plus white noise

```
fig, ax = plt.subplots()
ax.plot(yf[0, :], label="y")
ax.plot(xf[0, :], label="x")
ax.legend()
ax.set_title(r"$x$ and $y$")
ax.set_xlabel("Time")
plt.show()
```

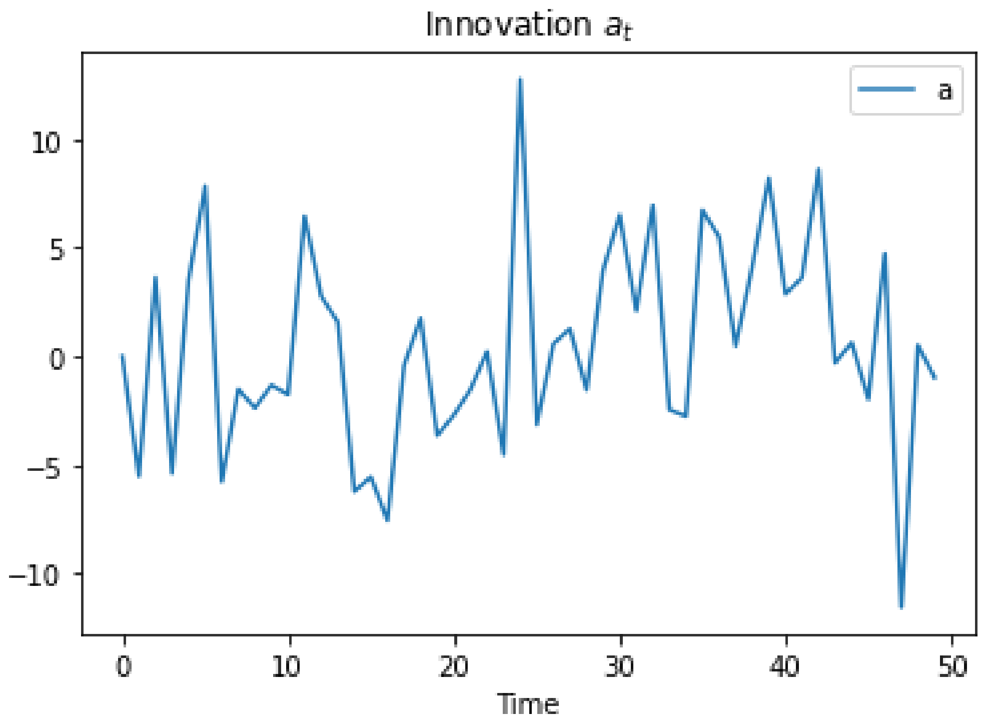


We see above that y seems to look like white noise around the values of x.

3.5.1 Innovations

Recall that we wrote down the innovation representation that depended on a_t. We now plot the innovations $\{a_t\}$:

```
fig, ax = plt.subplots()
ax.plot(yf[1, :], label="a")
ax.legend()
ax.set_title(r"Innovation $a_t$")
ax.set_xlabel("Time")
plt.show()
```



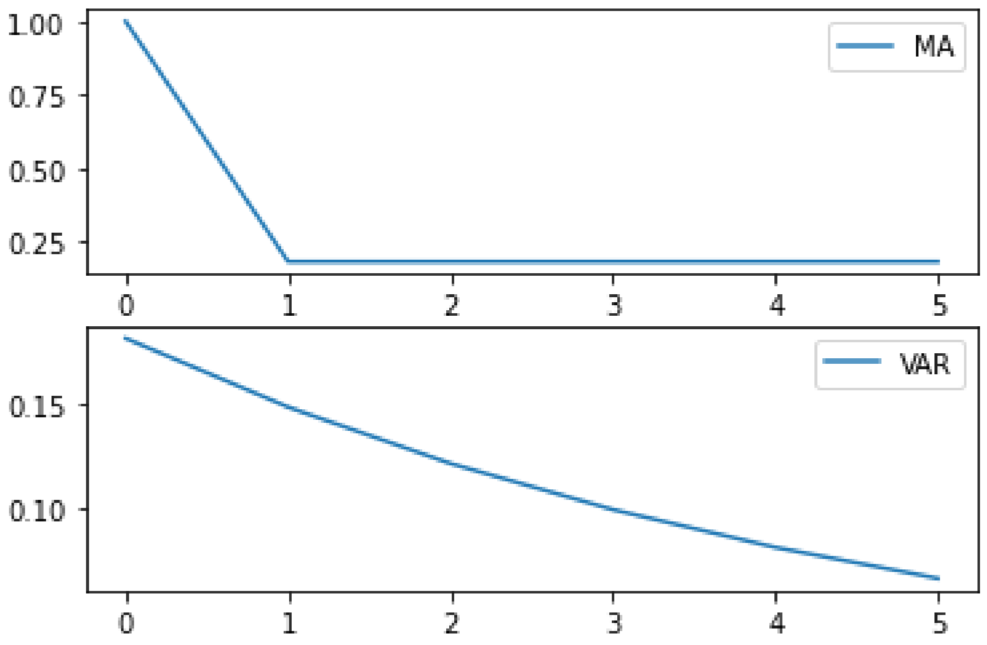
3.6 MA and AR Representations

Now we shall extract from the `Kalman` instance `kmuth` coefficients of

- a fundamental moving average representation that represents y_t as a one-sided moving sum of current and past a_ts that are square summable linear combinations of $y_t, y_{t-1}, \ldots$.
- a univariate autoregression representation that depicts the coefficients in a linear least square projection of y_t on the semi-infinite history $y_{t-1}, y_{t-2}, \ldots$.

Then we'll plot each of them

```
# Kalman Methods for MA and VAR
coefs_ma = kmuth.stationary_coefficients(5, "ma")
coefs_var = kmuth.stationary_coefficients(5, "var")

# Coefficients come in a list of arrays, but we
# want to plot them and so need to stack into an array
coefs_ma_array = np.vstack(coefs_ma)
coefs_var_array = np.vstack(coefs_var)

fig, ax = plt.subplots(2)
ax[0].plot(coefs_ma_array, label="MA")
ax[0].legend()
ax[1].plot(coefs_var_array, label="VAR")
ax[1].legend()

plt.show()
```



The **moving average** coefficients in the top panel show tell-tale signs of y_t being a process whose first difference is a first-order autoregression.

The **autoregressive coefficients** decline geometrically with decay rate $(1 - K)$.

These are exactly the target outcomes that Muth (1960) aimed to reverse engineer

```
print(f'decay parameter 1 - K1 = {1 - K1}')
```

```
decay parameter 1 - K1 = [[0.819]]
```

CHAPTER

FOUR

DISCRETE STATE DYNAMIC PROGRAMMING

Contents

- *Discrete State Dynamic Programming*
  - *Overview*
  - *Discrete DPs*
  - *Solving Discrete DPs*
  - *Example: A Growth Model*
  - *Exercises*
  - *Solutions*
  - *Appendix: Algorithms*

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

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

4.1 Overview

In this lecture we discuss a family of dynamic programming problems with the following features:

1. a discrete state space and discrete choices (actions)
2. an infinite horizon
3. discounted rewards
4. Markov state transitions

We call such problems discrete dynamic programs or discrete DPs.

Discrete DPs are the workhorses in much of modern quantitative economics, including

- monetary economics
- search and labor economics
- household savings and consumption theory
- investment theory
- asset pricing

- industrial organization, etc.

When a given model is not inherently discrete, it is common to replace it with a discretized version in order to use discrete DP techniques.

This lecture covers

- the theory of dynamic programming in a discrete setting, plus examples and applications
- a powerful set of routines for solving discrete DPs from the QuantEcon code library

Let's start with some imports:

```python
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import quantecon as qe
import scipy.sparse as sparse
from quantecon import compute_fixed_point
from quantecon.markov import DiscreteDP
```

4.1.1 How to Read this Lecture

We use dynamic programming many applied lectures, such as

- The shortest path lecture
- The McCall search model lecture

The objective of this lecture is to provide a more systematic and theoretical treatment, including algorithms and implementation while focusing on the discrete case.

4.1.2 Code

Among other things, it offers

- a flexible, well-designed interface
- multiple solution methods, including value function and policy function iteration
- high-speed operations via carefully optimized JIT-compiled functions
- the ability to scale to large problems by minimizing vectorized operators and allowing operations on sparse matrices

JIT compilation relies on Numba, which should work seamlessly if you are using Anaconda as suggested.

4.1.3 References

For background reading on dynamic programming and additional applications, see, for example,

- [LS18]
- [HLL96], section 3.5
- [Put05]
- [SLP89]
- [Rus96]
- [MF02]

- EDTC, chapter 5

4.2 Discrete DPs

Loosely speaking, a discrete DP is a maximization problem with an objective function of the form

$$\mathbb{E} \sum_{t=0}^{\infty} \beta^t r(s_t, a_t) \tag{1}$$

where

- s_t is the state variable
- a_t is the action
- β is a discount factor
- $r(s_t, a_t)$ is interpreted as a current reward when the state is s_t and the action chosen is a_t

Each pair (s_t, a_t) pins down transition probabilities $Q(s_t, a_t, s_{t+1})$ for the next period state s_{t+1}.

Thus, actions influence not only current rewards but also the future time path of the state.

The essence of dynamic programming problems is to trade off current rewards vs favorable positioning of the future state (modulo randomness).

Examples:

- consuming today vs saving and accumulating assets
- accepting a job offer today vs seeking a better one in the future
- exercising an option now vs waiting

4.2.1 Policies

The most fruitful way to think about solutions to discrete DP problems is to compare *policies*.

In general, a policy is a randomized map from past actions and states to current action.

In the setting formalized below, it suffices to consider so-called *stationary Markov policies*, which consider only the current state.

In particular, a stationary Markov policy is a map σ from states to actions

- $a_t = \sigma(s_t)$ indicates that a_t is the action to be taken in state s_t

It is known that, for any arbitrary policy, there exists a stationary Markov policy that dominates it at least weakly.

- See section 5.5 of [Put05] for discussion and proofs.

In what follows, stationary Markov policies are referred to simply as policies.

The aim is to find an optimal policy, in the sense of one that maximizes (1).

Let's now step through these ideas more carefully.

4.2.2 Formal Definition

Formally, a discrete dynamic program consists of the following components:

1. A finite set of *states* $S = \{0, \ldots, n-1\}$.
2. A finite set of *feasible actions* $A(s)$ for each state $s \in S$, and a corresponding set of *feasible state-action pairs*.

$$SA := \{(s, a) \mid s \in S, \ a \in A(s)\}$$

3. A *reward function* $r \colon SA \to \mathbb{R}$.
4. A *transition probability function* $Q \colon SA \to \Delta(S)$, where $\Delta(S)$ is the set of probability distributions over S.
5. A *discount factor* $\beta \in [0, 1)$.

We also use the notation $A := \bigcup_{s \in S} A(s) = \{0, \ldots, m-1\}$ and call this set the *action space*.

A *policy* is a function $\sigma \colon S \to A$.

A policy is called *feasible* if it satisfies $\sigma(s) \in A(s)$ for all $s \in S$.

Denote the set of all feasible policies by Σ.

If a decision-maker uses a policy $\sigma \in \Sigma$, then

- the current reward at time t is $r(s_t, \sigma(s_t))$
- the probability that $s_{t+1} = s'$ is $Q(s_t, \sigma(s_t), s')$

For each $\sigma \in \Sigma$, define

- r_σ by $r_\sigma(s) := r(s, \sigma(s)))$
- Q_σ by $Q_\sigma(s, s') := Q(s, \sigma(s), s')$

Notice that Q_σ is a stochastic matrix on S.

It gives transition probabilities of the *controlled chain* when we follow policy σ.

If we think of r_σ as a column vector, then so is $Q_\sigma^t r_\sigma$, and the s-th row of the latter has the interpretation

$$(Q_\sigma^t r_\sigma)(s) = \mathbb{E}[r(s_t, \sigma(s_t)) \mid s_0 = s] \quad \text{when } \{s_t\} \sim Q_\sigma \tag{2}$$

Comments

- $\{s_t\} \sim Q_\sigma$ means that the state is generated by stochastic matrix Q_σ.
- See this discussion on computing expectations of Markov chains for an explanation of the expression in (2).

Notice that we're not really distinguishing between functions from S to $\mathbb{R}$ and vectors in $\mathbb{R}^n$.

This is natural because they are in one to one correspondence.

4.2.3 Value and Optimality

Let $v_\sigma(s)$ denote the discounted sum of expected reward flows from policy σ when the initial state is s.

To calculate this quantity we pass the expectation through the sum in (1) and use (2) to get

$$v_\sigma(s) = \sum_{t=0}^{\infty} \beta^t (Q_\sigma^t r_\sigma)(s) \qquad (s \in S)$$

This function is called the *policy value function* for the policy σ.

The *optimal value function*, or simply *value function*, is the function $v^* \colon S \to \mathbb{R}$ defined by

$$v^*(s) = \max_{\sigma \in \Sigma} v_\sigma(s) \qquad (s \in S)$$

(We can use max rather than sup here because the domain is a finite set)

A policy $\sigma \in \Sigma$ is called *optimal* if $v_\sigma(s) = v^*(s)$ for all $s \in S$.

Given any $w \colon S \to \mathbb{R}$, a policy $\sigma \in \Sigma$ is called w-greedy if

$$\sigma(s) \in \arg\max_{a \in A(s)} \left\{ r(s,a) + \beta \sum_{s' \in S} w(s') Q(s,a,s') \right\} \qquad (s \in S)$$

As discussed in detail below, optimal policies are precisely those that are v^*-greedy.

4.2.4 Two Operators

It is useful to define the following operators:

- The *Bellman operator* $T \colon \mathbb{R}^S \to \mathbb{R}^S$ is defined by

$$(Tv)(s) = \max_{a \in A(s)} \left\{ r(s,a) + \beta \sum_{s' \in S} v(s') Q(s,a,s') \right\} \qquad (s \in S)$$

- For any policy function $\sigma \in \Sigma$, the operator $T_\sigma \colon \mathbb{R}^S \to \mathbb{R}^S$ is defined by

$$(T_\sigma v)(s) = r(s, \sigma(s)) + \beta \sum_{s' \in S} v(s') Q(s, \sigma(s), s') \qquad (s \in S)$$

This can be written more succinctly in operator notation as

$$T_\sigma v = r_\sigma + \beta Q_\sigma v$$

The two operators are both monotone

- $v \leq w$ implies $Tv \leq Tw$ pointwise on S, and similarly for T_σ

They are also contraction mappings with modulus β

- $\|Tv - Tw\| \leq \beta \|v - w\|$ and similarly for T_σ, where $\|\cdot\|$ is the max norm

For any policy σ, its value v_σ is the unique fixed point of T_σ.

For proofs of these results and those in the next section, see, for example, EDTC, chapter 10.

4.2.5 The Bellman Equation and the Principle of Optimality

The main principle of the theory of dynamic programming is that

- the optimal value function v^* is a unique solution to the *Bellman equation*

$$v(s) = \max_{a \in A(s)} \left\{ r(s,a) + \beta \sum_{s' \in S} v(s') Q(s,a,s') \right\} \qquad (s \in S)$$

or in other words, v^* is the unique fixed point of T, and

- σ^* is an optimal policy function if and only if it is v^*-greedy

By the definition of greedy policies given above, this means that

$$\sigma^*(s) \in \arg\max_{a \in A(s)} \left\{ r(s,a) + \beta \sum_{s' \in S} v^*(s') Q(s, \sigma(s), s') \right\} \qquad (s \in S)$$

4.3 Solving Discrete DPs

Now that the theory has been set out, let's turn to solution methods.

The code for solving discrete DPs is available in ddp.py from the QuantEcon.py code library.

It implements the three most important solution methods for discrete dynamic programs, namely

- value function iteration
- policy function iteration
- modified policy function iteration

Let's briefly review these algorithms and their implementation.

4.3.1 Value Function Iteration

Perhaps the most familiar method for solving all manner of dynamic programs is value function iteration.

This algorithm uses the fact that the Bellman operator T is a contraction mapping with fixed point v^*.

Hence, iterative application of T to any initial function $v^0 : S \to \mathbb{R}$ converges to v^*.

The details of the algorithm can be found in *the appendix*.

4.3.2 Policy Function Iteration

This routine, also known as Howard's policy improvement algorithm, exploits more closely the particular structure of a discrete DP problem.

Each iteration consists of

1. A policy evaluation step that computes the value v_σ of a policy σ by solving the linear equation $v = T_\sigma v$.
2. A policy improvement step that computes a v_σ-greedy policy.

In the current setting, policy iteration computes an exact optimal policy in finitely many iterations.

- See theorem 10.2.6 of EDTC for a proof.

The details of the algorithm can be found in *the appendix*.

4.3.3 Modified Policy Function Iteration

Modified policy iteration replaces the policy evaluation step in policy iteration with "partial policy evaluation".

The latter computes an approximation to the value of a policy σ by iterating T_σ for a specified number of times.

This approach can be useful when the state space is very large and the linear system in the policy evaluation step of policy iteration is correspondingly difficult to solve.

The details of the algorithm can be found in *the appendix*.

4.4 Example: A Growth Model

Let's consider a simple consumption-saving model.

A single household either consumes or stores its own output of a single consumption good.

The household starts each period with current stock s.

Next, the household chooses a quantity a to store and consumes $c = s - a$

- Storage is limited by a global upper bound M.
- Flow utility is $u(c) = c^{\alpha}$.

Output is drawn from a discrete uniform distribution on $\{0, \dots, B\}$.

The next period stock is therefore

$$s' = a + U \quad \text{where} \quad U \sim U[0, \dots, B]$$

The discount factor is $\beta \in [0, 1)$.

4.4.1 Discrete DP Representation

We want to represent this model in the format of a discrete dynamic program.

To this end, we take

- the state variable to be the stock s
- the state space to be $S = \{0, \dots, M + B\}$
  - hence $n = M + B + 1$
- the action to be the storage quantity a
- the set of feasible actions at s to be $A(s) = \{0, \dots, \min\{s, M\}\}$
  - hence $A = \{0, \dots, M\}$ and $m = M + 1$
- the reward function to be $r(s, a) = u(s - a)$
- the transition probabilities to be

$$Q(s, a, s') := \begin{cases} \frac{1}{B+1} & \text{if } a \leq s' \leq a + B \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

4.4.2 Defining a DiscreteDP Instance

This information will be used to create an instance of DiscreteDP by passing the following information

1. An $n \times m$ reward array R.
2. An $n \times m \times n$ transition probability array Q.
3. A discount factor β.

For R we set $R[s, a] = u(s - a)$ if $a \leq s$ and $-\infty$ otherwise.

For Q we follow the rule in (3).

Note:

- The feasibility constraint is embedded into R by setting $R[s,a] = -\infty$ for $a \notin A(s)$.
- Probability distributions for (s,a) with $a \notin A(s)$ can be arbitrary.

The following code sets up these objects for us

```python
class SimpleOG:

    def __init__(self, B=10, M=5, α=0.5, β=0.9):
        """
        Set up R, Q and β, the three elements that define an instance of
        the DiscreteDP class.
        """

        self.B, self.M, self.α, self.β  = B, M, α, β
        self.n = B + M + 1
        self.m = M + 1

        self.R = np.empty((self.n, self.m))
        self.Q = np.zeros((self.n, self.m, self.n))

        self.populate_Q()
        self.populate_R()

    def u(self, c):
        return c**self.α

    def populate_R(self):
        """
        Populate the R matrix, with R[s, a] = -np.inf for infeasible
        state-action pairs.
        """
        for s in range(self.n):
            for a in range(self.m):
                self.R[s, a] = self.u(s - a) if a <= s else -np.inf

    def populate_Q(self):
        """
        Populate the Q matrix by setting

            Q[s, a, s'] = 1 / (1 + B) if a <= s' <= a + B

        and zero otherwise.
        """

        for a in range(self.m):
            self.Q[:, a, a:(a + self.B + 1)] = 1.0 / (self.B + 1)
```

Let’s run this code and create an instance of `SimpleOG`.

```python
g = SimpleOG()  # Use default parameters
```

Instances of `DiscreteDP` are created using the signature `DiscreteDP(R, Q, β)`.

Let’s create an instance using the objects stored in `g`

```python
ddp = qe.markov.DiscreteDP(g.R, g.Q, g.β)
```

Now that we have an instance `ddp` of `DiscreteDP` we can solve it as follows

```
results = ddp.solve(method='policy_iteration')
```

Let's see what we've got here

```
dir(results)
```

```
['max_iter', 'mc', 'method', 'num_iter', 'sigma', 'v']
```

(In IPython version 4.0 and above you can also type `results.` and hit the tab key)

The most important attributes are `v`, the value function, and `σ`, the optimal policy

```
results.v
```

```
array([19.01740222, 20.01740222, 20.43161578, 20.74945302, 21.04078099,
       21.30873018, 21.54479816, 21.76928181, 21.98270358, 22.18824323,
       22.3845048 , 22.57807736, 22.76109127, 22.94376708, 23.11533996,
       23.27761762])
```

```
results.sigma
```

```
array([0, 0, 0, 0, 1, 1, 1, 2, 2, 3, 3, 4, 5, 5, 5, 5])
```

Since we've used policy iteration, these results will be exact unless we hit the iteration bound `max_iter`.

Let's make sure this didn't happen

```
results.max_iter
```

```
250
```

```
results.num_iter
```

```
3
```

Another interesting object is `results.mc`, which is the controlled chain defined by Q_{σ^*}, where σ^* is the optimal policy.

In other words, it gives the dynamics of the state when the agent follows the optimal policy.

Since this object is an instance of MarkovChain from QuantEcon.py (see this lecture for more discussion), we can easily simulate it, compute its stationary distribution and so on.

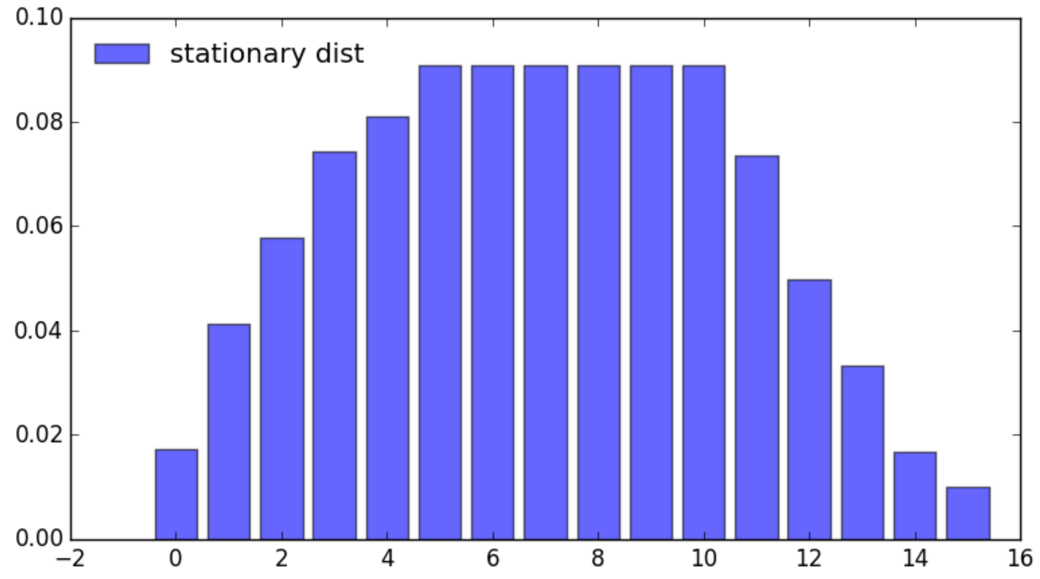
```
results.mc.stationary_distributions
```

```
array([[0.01732187, 0.04121063, 0.05773956, 0.07426848, 0.08095823,
        0.09090909, 0.09090909, 0.09090909, 0.09090909, 0.09090909,
        0.09090909, 0.07358722, 0.04969846, 0.03316953, 0.01664061,
        0.00995086]])
```

Here's the same information in a bar graph

What happens if the agent is more patient?

```
ddp = qe.markov.DiscreteDP(g.R, g.Q, 0.99)  # Increase β to 0.99
results = ddp.solve(method='policy_iteration')
results.mc.stationary_distributions
```



```
array([[0.00546913, 0.02321342, 0.03147788, 0.04800681, 0.05627127,
        0.09090909, 0.09090909, 0.09090909, 0.09090909, 0.09090909,
        0.09090909, 0.08543996, 0.06769567, 0.05943121, 0.04290228,
        0.03463782]])
```

If we look at the bar graph we can see the rightward shift in probability mass

4.4.3 State-Action Pair Formulation

The `DiscreteDP` class in fact, provides a second interface to set up an instance.

One of the advantages of this alternative set up is that it permits the use of a sparse matrix for `Q`.

(An example of using sparse matrices is given in the exercises below)

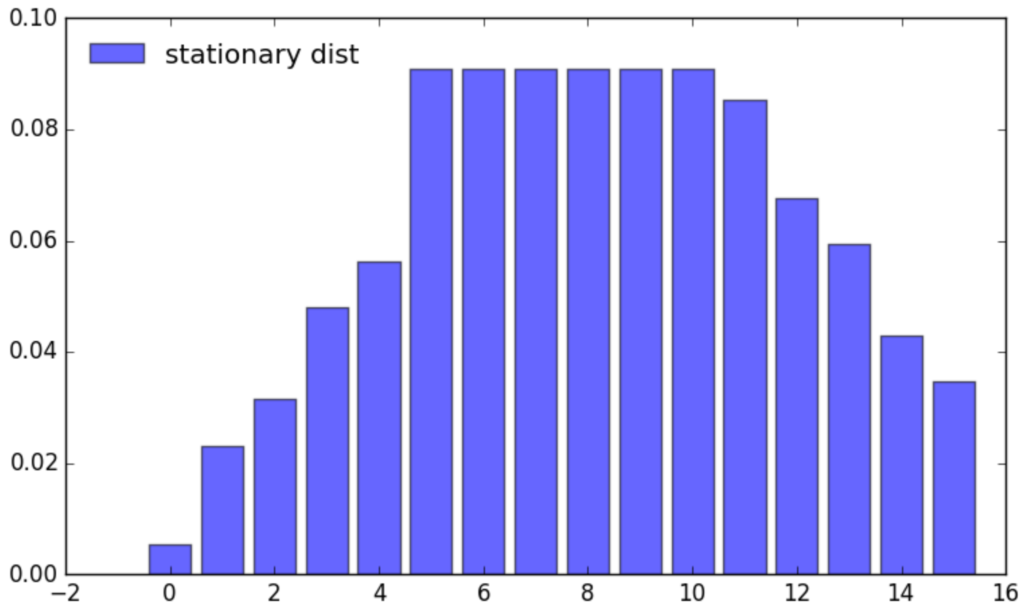
The call signature of the second formulation is `DiscreteDP(R, Q, β, s_indices, a_indices)` where

- `s_indices` and `a_indices` are arrays of equal length `L` enumerating all feasible state-action pairs
- `R` is an array of length `L` giving corresponding rewards
- `Q` is an `L x n` transition probability array

Here's how we could set up these objects for the preceding example

```
B, M, α, β = 10, 5, 0.5, 0.9
n = B + M + 1
m = M + 1

def u(c):
    return c**α

s_indices = []
a_indices = []
```

(continues on next page)



(continued from previous page)

```
Q = []
R = []
b = 1.0 / (B + 1)

for s in range(n):
    for a in range(min(M, s) + 1):  # All feasible a at this s
        s_indices.append(s)
        a_indices.append(a)
        q = np.zeros(n)
        q[a:(a + B + 1)] = b        # b on these values, otherwise 0
        Q.append(q)
        R.append(u(s - a))

ddp = qe.markov.DiscreteDP(R, Q, β, s_indices, a_indices)
```

For larger problems, you might need to write this code more efficiently by vectorizing or using Numba.

4.5 Exercises

In the stochastic optimal growth lecture from our introductory lecture series, we solve a benchmark model that has an analytical solution.

The exercise is to replicate this solution using `DiscreteDP`.

4.6 Solutions

4.6.1 Setup

Details of the model can be found in the lecture on optimal growth.

We let $f(k) = k^\alpha$ with $\alpha = 0.65$, $u(c) = \log c$, and $\beta = 0.95$

```
α = 0.65
f = lambda k: k**α
u = np.log
β = 0.95
```

Here we want to solve a finite state version of the continuous state model above.

We discretize the state space into a grid of size `grid_size=500`, from 10^{-6} to `grid_max=2`

```
grid_max = 2
grid_size = 500
grid = np.linspace(1e-6, grid_max, grid_size)
```

We choose the action to be the amount of capital to save for the next period (the state is the capital stock at the beginning of the period).

Thus the state indices and the action indices are both `0`, …, `grid_size-1`.

Action (indexed by) `a` is feasible at state (indexed by) `s` if and only if `grid[a] < f([grid[s])` (zero consumption is not allowed because of the log utility).

Thus the Bellman equation is:

$$v(k) = \max_{0<k'<f(k)} u(f(k) - k') + \beta v(k'),$$

where k' is the capital stock in the next period.

The transition probability array `Q` will be highly sparse (in fact it is degenerate as the model is deterministic), so we formulate the problem with state-action pairs, to represent `Q` in scipy sparse matrix format.

We first construct indices for state-action pairs:

```
# Consumption matrix, with nonpositive consumption included
C = f(grid).reshape(grid_size, 1) - grid.reshape(1, grid_size)

# State-action indices
s_indices, a_indices = np.where(C > 0)

# Number of state-action pairs
L = len(s_indices)

print(L)
print(s_indices)
print(a_indices)
```

```
118841
[  0   1   1 ... 499 499 499]
[  0   0   1 ... 389 390 391]
```

Reward vector `R` (of length `L`):

```
R = u(C[s_indices, a_indices])
```

(Degenerate) transition probability matrix `Q` (of shape `(L, grid_size)`), where we choose the scipy.sparse.lil_matrix format, while any format will do (internally it will be converted to the csr format):

```
Q = sparse.lil_matrix((L, grid_size))
Q[np.arange(L), a_indices] = 1
```

(If you are familiar with the data structure of scipy.sparse.csr_matrix, the following is the most efficient way to create the `Q` matrix in the current case)

```
# data = np.ones(L)
# indptr = np.arange(L+1)
# Q = sparse.csr_matrix((data, a_indices, indptr), shape=(L, grid_size))
```

Discrete growth model:

```
ddp = DiscreteDP(R, Q, β, s_indices, a_indices)
```

Notes

Here we intensively vectorized the operations on arrays to simplify the code.

As noted, however, vectorization is memory consumptive, and it can be prohibitively so for grids with large size.

4.6.2 Solving the Model

Solve the dynamic optimization problem:

```
res = ddp.solve(method='policy_iteration')
v, σ, num_iter = res.v, res.sigma, res.num_iter
num_iter
```

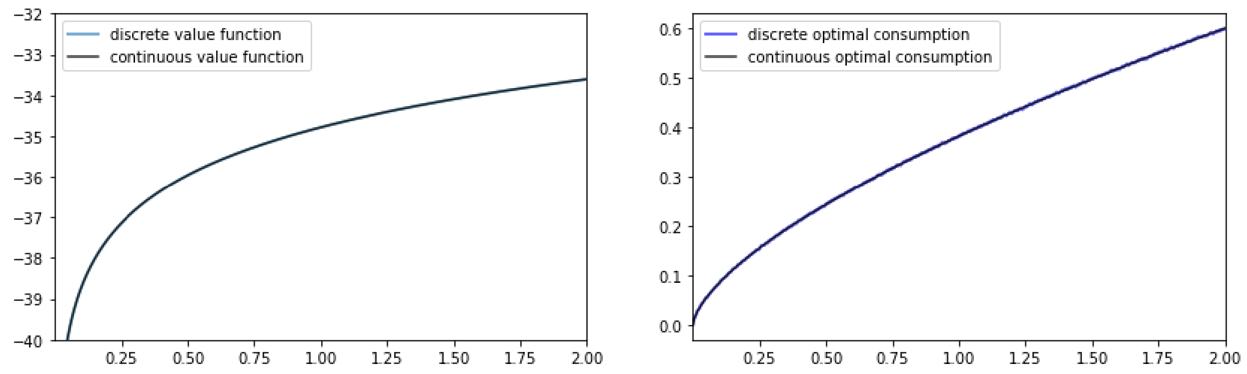
```
10
```

Note that `sigma` contains the *indices* of the optimal *capital stocks* to save for the next period. The following translates `sigma` to the corresponding consumption vector.

```
# Optimal consumption in the discrete version
c = f(grid) - grid[σ]

# Exact solution of the continuous version
ab = α * β
c1 = (np.log(1 - ab) + np.log(ab) * ab / (1 - ab)) / (1 - β)
c2 = α / (1 - ab)

def v_star(k):
    return c1 + c2 * np.log(k)

def c_star(k):
    return (1 - ab) * k**α
```

Let us compare the solution of the discrete model with that of the original continuous model

```
fig, ax = plt.subplots(1, 2, figsize=(14, 4))
ax[0].set_ylim(-40, -32)
ax[0].set_xlim(grid[0], grid[-1])
ax[1].set_xlim(grid[0], grid[-1])

lb0 = 'discrete value function'
ax[0].plot(grid, v, lw=2, alpha=0.6, label=lb0)

lb0 = 'continuous value function'
ax[0].plot(grid, v_star(grid), 'k-', lw=1.5, alpha=0.8, label=lb0)
ax[0].legend(loc='upper left')

lb1 = 'discrete optimal consumption'
ax[1].plot(grid, c, 'b-', lw=2, alpha=0.6, label=lb1)

lb1 = 'continuous optimal consumption'
ax[1].plot(grid, c_star(grid), 'k-', lw=1.5, alpha=0.8, label=lb1)
ax[1].legend(loc='upper left')
plt.show()
```



The outcomes appear very close to those of the continuous version.

Except for the “boundary” point, the value functions are very close:

```
np.abs(v - v_star(grid)).max()
```

```
121.49819147053378
```

```
np.abs(v - v_star(grid))[1:].max()
```

```
0.012681735127500815
```

The optimal consumption functions are close as well:

```
np.abs(c - c_star(grid)).max()
```

```
0.003826523100010082
```

In fact, the optimal consumption obtained in the discrete version is not really monotone, but the decrements are quite small:

```
diff = np.diff(c)
(diff >= 0).all()
```

```
False
```

```
dec_ind = np.where(diff < 0)[0]
len(dec_ind)
```

```
174
```

```
np.abs(diff[dec_ind]).max()
```

```
0.0019618533397668398
```

The value function is monotone:

```
(np.diff(v) > 0).all()
```

```
True
```

4.6.3 Comparison of the Solution Methods

Let us solve the problem with the other two methods.

Value Iteration

```
ddp.epsilon = 1e-4
ddp.max_iter = 500
res1 = ddp.solve(method='value_iteration')
res1.num_iter
```

```
294
```

```
np.array_equal(σ, res1.sigma)
```

```
True
```

Modified Policy Iteration

```
res2 = ddp.solve(method='modified_policy_iteration')
res2.num_iter
```

```
16
```

```
np.array_equal(σ, res2.sigma)
```

```
True
```

Speed Comparison

```
%timeit ddp.solve(method='value_iteration')
%timeit ddp.solve(method='policy_iteration')
%timeit ddp.solve(method='modified_policy_iteration')
```

```
221 ms ± 2.77 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
19.6 ms ± 327 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
21.4 ms ± 255 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

As is often the case, policy iteration and modified policy iteration are much faster than value iteration.

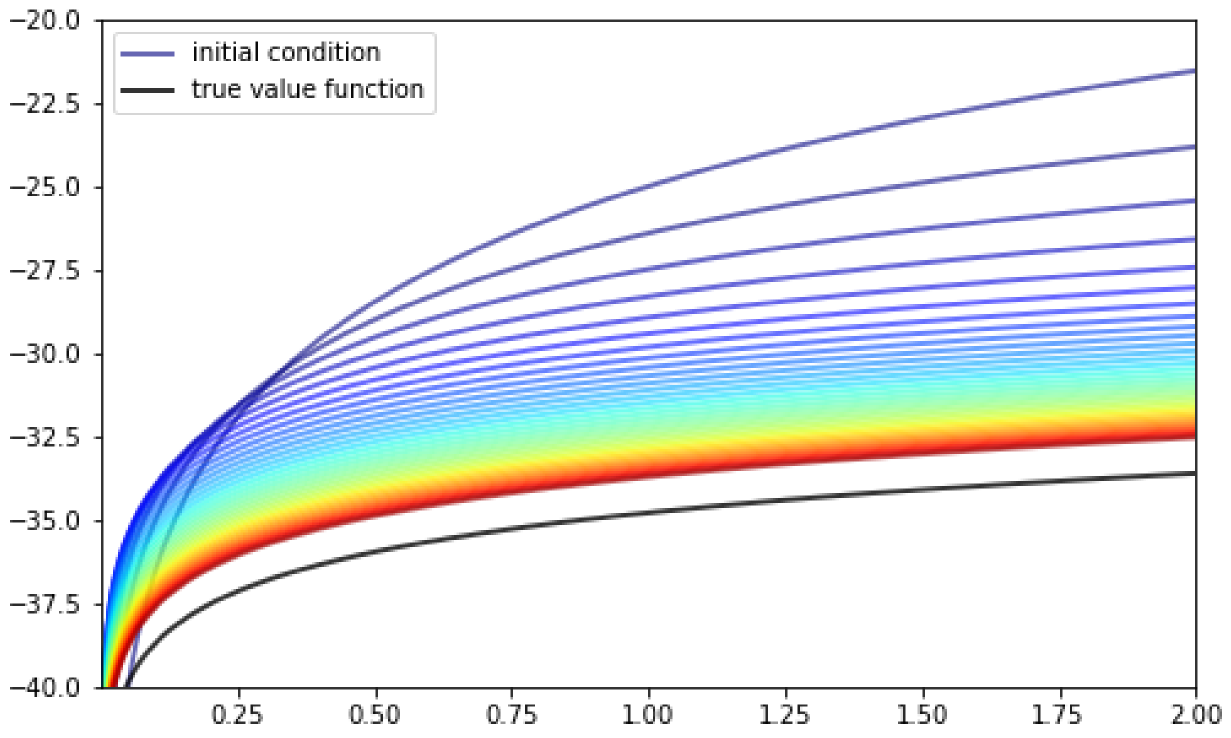
4.6.4 Replication of the Figures

Using `DiscreteDP` we replicate the figures shown in the lecture.

Convergence of Value Iteration

Let us first visualize the convergence of the value iteration algorithm as in the lecture, where we use `ddp.bellman_operator` implemented as a method of `DiscreteDP`

```
w = 5 * np.log(grid) - 25  # Initial condition
n = 35
fig, ax = plt.subplots(figsize=(8,5))
ax.set_ylim(-40, -20)
ax.set_xlim(np.min(grid), np.max(grid))
lb = 'initial condition'
ax.plot(grid, w, color=plt.cm.jet(0), lw=2, alpha=0.6, label=lb)
for i in range(n):
    w = ddp.bellman_operator(w)
    ax.plot(grid, w, color=plt.cm.jet(i / n), lw=2, alpha=0.6)
lb = 'true value function'
ax.plot(grid, v_star(grid), 'k-', lw=2, alpha=0.8, label=lb)
ax.legend(loc='upper left')

plt.show()
```



We next plot the consumption policies along with the value iteration

```
w = 5 * u(grid) - 25           # Initial condition

fig, ax = plt.subplots(3, 1, figsize=(8, 10))
true_c = c_star(grid)

for i, n in enumerate((2, 4, 6)):
    ax[i].set_ylim(0, 1)
    ax[i].set_xlim(0, 2)
    ax[i].set_yticks((0, 1))
    ax[i].set_xticks((0, 2))

    w = 5 * u(grid) - 25       # Initial condition
    compute_fixed_point(ddp.bellman_operator, w, max_iter=n, print_skip=1)
    σ = ddp.compute_greedy(w)  # Policy indices
    c_policy = f(grid) - grid[σ]

    ax[i].plot(grid, c_policy, 'b-', lw=2, alpha=0.8,
               label='approximate optimal consumption policy')
    ax[i].plot(grid, true_c, 'k-', lw=2, alpha=0.8,
               label='true optimal consumption policy')
    ax[i].legend(loc='upper left')
    ax[i].set_title(f'{n} value function iterations')
plt.show()
```

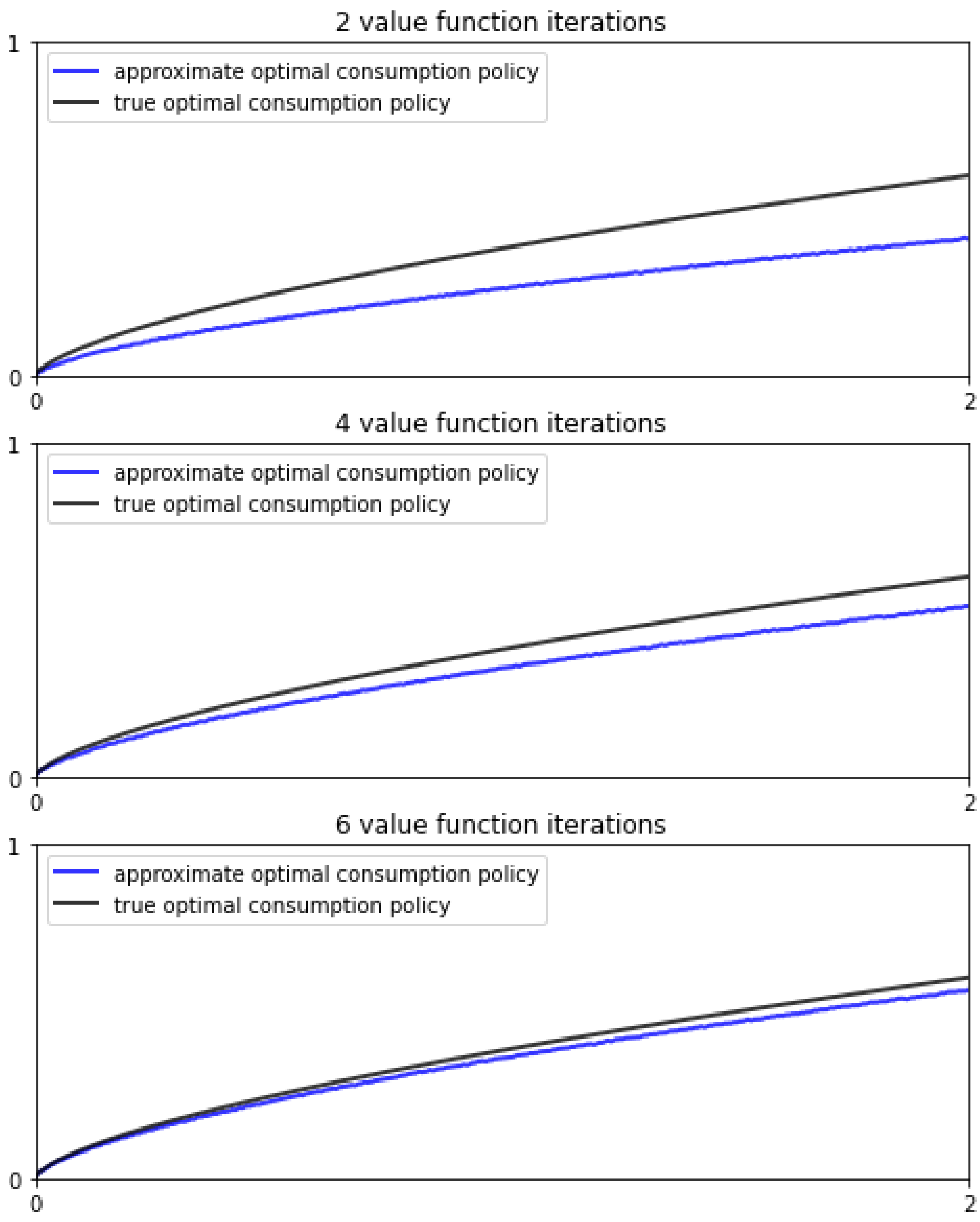
```
Iteration    Distance       Elapsed (seconds)
---------------------------------------------
1            5.518e+00      1.409e-03
2            4.070e+00      2.533e-03
Iteration    Distance       Elapsed (seconds)
```

(continues on next page)

(continued from previous page)

```
---------------------------------------------
1            5.518e+00      9.253e-04
2            4.070e+00      2.020e-03
3            3.866e+00      3.011e-03
4            3.673e+00      3.854e-03
Iteration    Distance       Elapsed (seconds)
---------------------------------------------
1            5.518e+00      8.845e-04
2            4.070e+00      1.707e-03
3            3.866e+00      3.022e-03
4            3.673e+00      4.036e-03
5            3.489e+00      4.991e-03
6            3.315e+00      5.861e-03
```

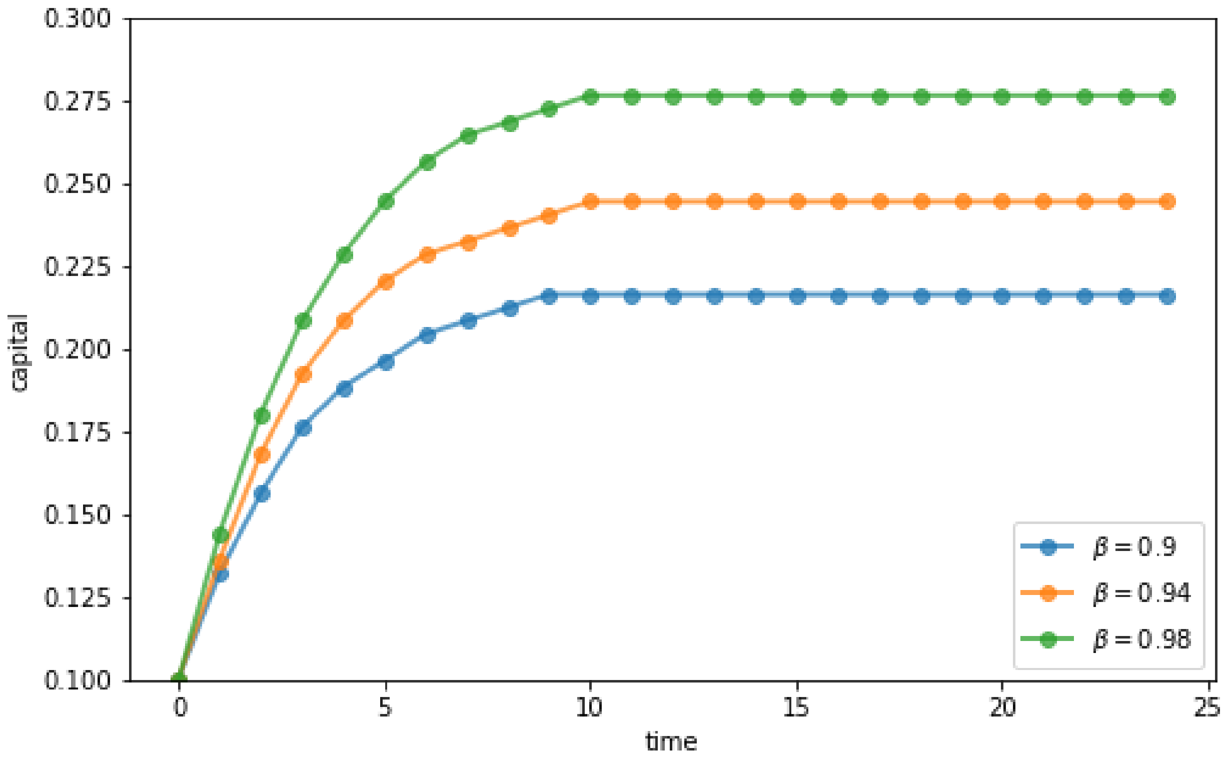
```
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/quantecon/compute_fp.
 ↪py:151: RuntimeWarning: max_iter attained before convergence in compute_fixed_point
  warnings.warn(_non_convergence_msg, RuntimeWarning)
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/quantecon/compute_fp.
 ↪py:151: RuntimeWarning: max_iter attained before convergence in compute_fixed_point
  warnings.warn(_non_convergence_msg, RuntimeWarning)
/usr/share/miniconda3/envs/quantecon/lib/python3.8/site-packages/quantecon/compute_fp.
 ↪py:151: RuntimeWarning: max_iter attained before convergence in compute_fixed_point
  warnings.warn(_non_convergence_msg, RuntimeWarning)
```


2 value function iterations
approximate optimal consumption policy
true optimal consumption policy
1
0
0
2
4 value function iterations
approximate optimal consumption policy
true optimal consumption policy
1
0
0
2
6 value function iterations
approximate optimal consumption policy
true optimal consumption policy
1
0
0
2

Dynamics of the Capital Stock

Finally, let us work on Exercise 2, where we plot the trajectories of the capital stock for three different discount factors, 0.9, 0.94, and 0.98, with initial condition $k_0 = 0.1$.

```python
discount_factors = (0.9, 0.94, 0.98)
k_init = 0.1

# Search for the index corresponding to k_init
k_init_ind = np.searchsorted(grid, k_init)

sample_size = 25

fig, ax = plt.subplots(figsize=(8,5))
ax.set_xlabel("time")
ax.set_ylabel("capital")
ax.set_ylim(0.10, 0.30)

# Create a new instance, not to modify the one used above
ddp0 = DiscreteDP(R, Q, β, s_indices, a_indices)

for beta in discount_factors:
    ddp0.beta = beta
    res0 = ddp0.solve()
    k_path_ind = res0.mc.simulate(init=k_init_ind, ts_length=sample_size)
    k_path = grid[k_path_ind]
    ax.plot(k_path, 'o-', lw=2, alpha=0.75, label=f'$\\beta = {beta}$')

ax.legend(loc='lower right')
plt.show()
```



4.7 Appendix: Algorithms

This appendix covers the details of the solution algorithms implemented for `DiscreteDP`.

We will make use of the following notions of approximate optimality:

- For $\varepsilon > 0$, v is called an ε-approximation of v^* if $\|v - v^*\| < \varepsilon$.
- A policy $\sigma \in \Sigma$ is called ε-optimal if v_σ is an ε-approximation of v^*.

4.7.1 Value Iteration

The `DiscreteDP` value iteration method implements value function iteration as follows

1. Choose any $v^0 \in \mathbb{R}^n$, and specify $\varepsilon > 0$; set $i = 0$.
2. Compute $v^{i+1} = Tv^i$.
3. If $\|v^{i+1} - v^i\| < [(1-\beta)/(2\beta)]\varepsilon$, then go to step 4; otherwise, set $i = i + 1$ and go to step 2.
4. Compute a v^{i+1}-greedy policy σ, and return v^{i+1} and σ.

Given $\varepsilon > 0$, the value iteration algorithm

- terminates in a finite number of iterations
- returns an $\varepsilon/2$-approximation of the optimal value function and an ε-optimal policy function (unless `iter_max` is reached)

(While not explicit, in the actual implementation each algorithm is terminated if the number of iterations reaches `iter_max`)

4.7.2 Policy Iteration

The `DiscreteDP` policy iteration method runs as follows

1. Choose any $v^0 \in \mathbb{R}^n$ and compute a v^0-greedy policy σ^0; set $i = 0$.
2. Compute the value v_{σ^i} by solving the equation $v = T_{\sigma^i} v$.
3. Compute a v_{σ^i}-greedy policy σ^{i+1}; let $\sigma^{i+1} = \sigma^i$ if possible.
4. If $\sigma^{i+1} = \sigma^i$, then return v_{σ^i} and σ^{i+1}; otherwise, set $i = i + 1$ and go to step 2.

The policy iteration algorithm terminates in a finite number of iterations.

It returns an optimal value function and an optimal policy function (unless `iter_max` is reached).

4.7.3 Modified Policy Iteration

The `DiscreteDP` modified policy iteration method runs as follows:

1. Choose any $v^0 \in \mathbb{R}^n$, and specify $\varepsilon > 0$ and $k \geq 0$; set $i = 0$.
2. Compute a v^i-greedy policy σ^{i+1}; let $\sigma^{i+1} = \sigma^i$ if possible (for $i \geq 1$).
3. Compute $u = Tv^i$ $(= T_{\sigma^{i+1}} v^i)$. If $\mathrm{span}(u - v^i) < [(1-\beta)/\beta]\varepsilon$, then go to step 5; otherwise go to step 4.
   - Span is defined by $\mathrm{span}(z) = \max(z) - \min(z)$.
4. Compute $v^{i+1} = (T_{\sigma^{i+1}})^k u$ $(= (T_{\sigma^{i+1}})^{k+1} v^i)$; set $i = i + 1$ and go to step 2.

5. Return $v = u + [\beta/(1-\beta)][(\min(u - v^i) + \max(u - v^i))/2]\mathbf{1}$ and σ_{i+1}.

Given $\varepsilon > 0$, provided that v^0 is such that $Tv^0 \geq v^0$, the modified policy iteration algorithm terminates in a finite number of iterations.

It returns an $\varepsilon/2$-approximation of the optimal value function and an ε-optimal policy function (unless `iter_max` is reached).

See also the documentation for `DiscreteDP`.