A First Course in Quantitative Economics with Python

Thomas J. Sargent & John Stachurski

Sep 17, 2024

CONTENTS

This lecture series provides an introduction to quantitative economics using Python.

- Estimation

  - *Simple Linear Regression Model*

  - *Maximum Likelihood Estimation*

- Other

  - *Troubleshooting*

  - *References*

  - *Execution Statistics*

Part I

Introduction

ABOUT THESE LECTURES

1.1 About

This lecture series introduces quantitative economics using elementary mathematics and statistics plus computer code written in Python.

The lectures emphasize simulation and visualization through code as a way to convey ideas, rather than focusing on mathematical details.

Although the presentation is quite novel, the ideas are rather foundational.

We emphasize the deep and fundamental importance of economic theory, as well as the value of analyzing data and understanding stylized facts.

The lectures can be used for university courses, self-study, reading groups or workshops.

Researchers and policy professionals might also find some parts of the series valuable for their work.

We hope the lectures will be of interest to students of economics who want to learn both economics and computing, as well as students from fields such as computer science and engineering who are curious about economics.

1.2 Level

The lecture series is aimed at undergraduate students.

The level of the lectures varies from truly introductory (suitable for first year undergraduates or even high school students) to more intermediate.

The more intermediate lectures require comfort with linear algebra and some mathematical maturity (e.g., calmly reading theorems and trying to understand their meaning).

In general, easier lectures occur earlier in the lecture series and harder lectures occur later.

We assume that readers have covered the easier parts of the QuantEcon lecture series on Python programming.

In particular, readers should be familiar with basic Python syntax including Python functions. Knowledge of classes and Matplotlib will be beneficial but not essential.

1.3 Credits

In building this lecture series, we had invaluable assistance from research assistants at QuantEcon, as well as our QuantEcon colleagues. Without their help this series would not have been possible.

In particular, we sincerely thank and give credit to

- Aakash Gupta
- Shu Hu
- Jiacheng Li
- Smit Lunagariya
- Matthew McKay
- Maanasee Sharma
- Humphrey Yang

We also thank Noritaka Kudoh for encouraging us to start this project and providing thoughtful suggestions.

Part II

Economic Data

TWO

长期增长

2.1 概述

在本讲座中，我们将使用Python、pandas 和 Matplotlib 来下载、处理和可视化经济增长的历史数据。

除了学习如何运用这些工具之外，我们还将使用它们来描述跨越多个世纪的多个国家的经济增长经历。

这些"增长事实"之所以有趣，原因是多方面的。

首先解释增长事实是"发展经济学"和"经济史"的主要目的。

其次增长事实也是历史学家研究地缘政治力量和动态的重要依据。

因此，亚当-托兹在阐述第一次世界大战的地缘政治先例和前因后果时，首先描述了欧洲大国的国内生产总值（GDP）在 1914 年之前的 70 年间是如何演变的（见 [Tooze, 2014] 第 1 章）。

使用与 Tooze 构建图表时相同的数据（时间线稍长），下面是我们对他的第 1 章图表的解读。

（这只是我们的图 Fig. 2.6 的副本。我们将在本讲座的稍后部分介绍如何绘制该图）。

[Tooze, 2014] 的第1章用他的图表说明了美国的GDP在19世纪初如何远远落后于大英帝国的GDP。

到 19 世纪末，美国的 GDP 已赶上大英帝国的 GDP，而在 20 世纪上半叶，美国的国内生产总值超过了大英帝国。
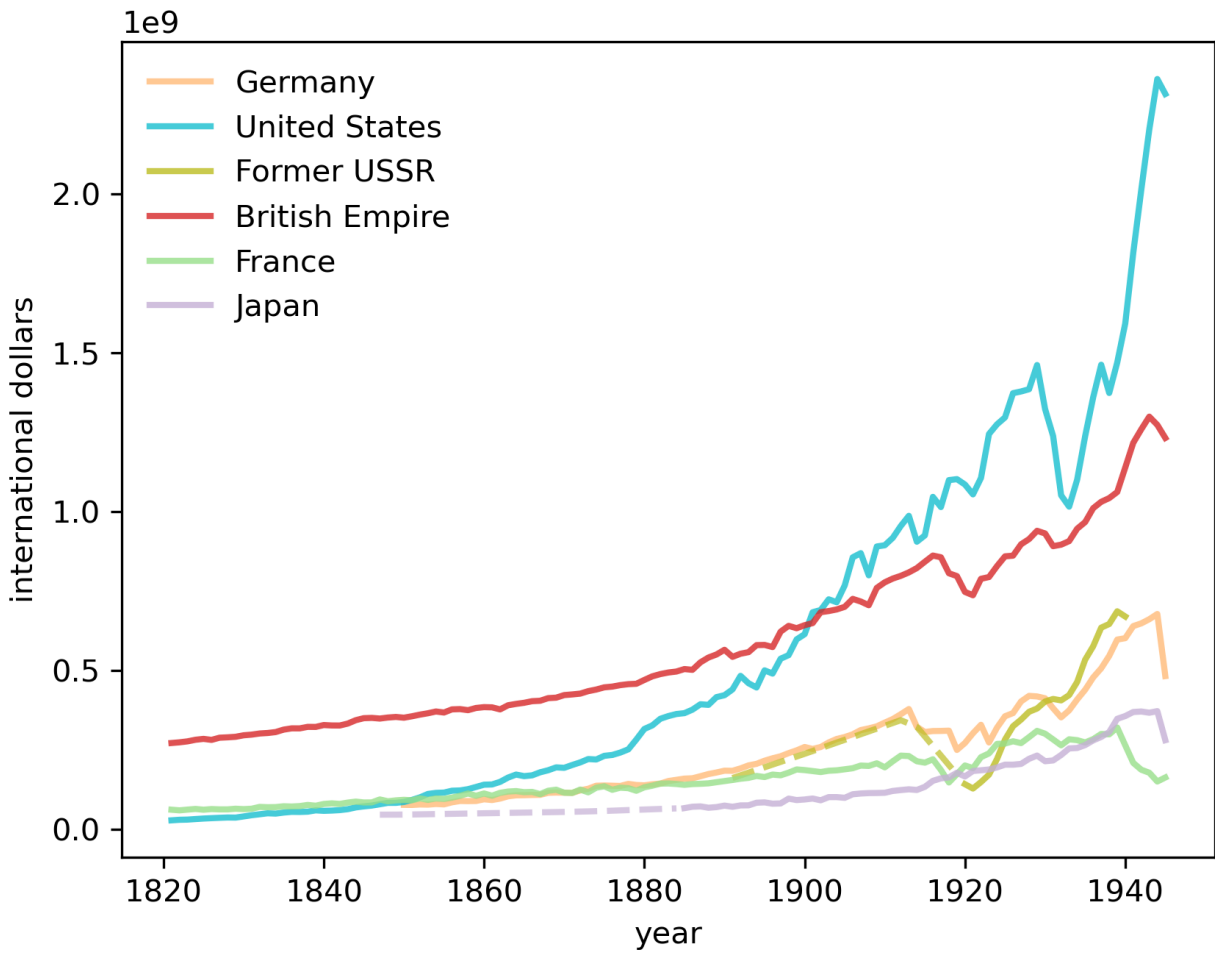
在亚当-图兹看来，这一事实是"美国世纪"的关键地缘政治基础。

看了这张图，再看看它是如何为"美国的20世纪"搭建地缘政治舞台的，自然而然地诱使人们想要一个与他的图表相对应的 2014 年或以后的图表。

(感兴趣的读者若想找到答案的提示，现在不妨跳到前面看看图Fig. 2.7)。

正如我们将看到的，通过类比推理，这张图表或许奠定了"XXX国的21世纪"的基础,你可以自由猜测国家XXX。

在我们收集数据以构建这两个图表的过程中，我们还将研究多国在尽可能长的时间范围内的增长经历。

这些图表将描绘"工业革命"如何在18世纪末始于英国，然后传播到一个国家又一个国家。

简而言之，本讲记录了各个国家在很长一段时间内的增长轨迹。

一些国家经历了持续百年的长期快速增长，而另一些国家则没有。

由于各国的人口数量不同，而且一个国家内部的人口数量也会随着时间的推移而变化。因此，描述一个国家的 GDP 总量和人均 GDP 的变化是非常有趣的。

首先，让我们导入所需的库。

```python
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np
from collections import namedtuple
```

2.2 准备数据

由安格斯-麦迪逊（Angus Maddison）发起的一个项目收集了许多与经济增长相关的历史时间序列、有些可以追溯到第一个世纪。

可以从 Maddison Historical Statistics 下载数据，点击"Latest Maddison Project Release"即可获取。

我们将从 QuantEcon GitHub 存储库中读取数据。

本节的目的是生成一个方便的 `DataFrame` 实例，其中包含不同国家的人均 GDP。

在这里，我们将 Maddison 的数据读入一个 pandas `DataFrame`:

```python
data_url = "https://github.com/QuantEcon/lecture-python-intro/raw/main/lectures/
↪datasets/mpd2020.xlsx"
data = pd.read_excel(data_url,
                     sheet_name='Full data')
data.head()
```

```
  countrycode      country  year   gdppc     pop
0         AFG  Afghanistan  1820     NaN  3280.0
1         AFG  Afghanistan  1870     NaN  4207.0
2         AFG  Afghanistan  1913     NaN  5730.0
3         AFG  Afghanistan  1950  1156.0  8150.0
4         AFG  Afghanistan  1951  1170.0  8284.0
```

我们可以看到，该数据集包含许多国家和年份的人均国内生产总值（`gdppc`）和人口（`pop`）。

让我们看看这个数据集中有多少国家和哪些国家

```
countries = data.country.unique()
len(countries)
```

```
169
```

通过运行上面的代码，我们可以找到有多少个国家的数据。

下面，我们将列出这些国家。

```
countries
```

```
array(['Afghanistan', 'Angola', 'Albania', 'United Arab Emirates',
       'Argentina', 'Armenia', 'Australia', 'Austria', 'Azerbaijan',
       'Burundi', 'Belgium', 'Benin', 'Burkina Faso', 'Bangladesh',
       'Bulgaria', 'Bahrain', 'Bosnia and Herzegovina', 'Belarus',
       'Bolivia (Plurinational State of)', 'Brazil', 'Barbados',
       'Botswana', 'Central African Republic', 'Canada', 'Switzerland',
       'Chile', 'China', "Côte d'Ivoire", 'Cameroon', 'D.R. of the Congo',
       'Congo', 'Colombia', 'Comoros', 'Cabo Verde', 'Costa Rica',
       'Czechoslovakia', 'Cuba', 'Cyprus', 'Czech Republic', 'Germany',
       'Djibouti', 'Dominica', 'Denmark', 'Dominican Republic', 'Algeria',
       'Ecuador', 'Egypt', 'Spain', 'Estonia', 'Ethiopia', 'Finland',
       'France', 'Gabon', 'United Kingdom', 'Georgia', 'Ghana', 'Guinea',
       'Gambia', 'Guinea-Bissau', 'Equatorial Guinea', 'Greece',
       'Guatemala', 'China, Hong Kong SAR', 'Honduras', 'Croatia',
       'Haiti', 'Hungary', 'Indonesia', 'India', 'Ireland',
       'Iran (Islamic Republic of)', 'Iraq', 'Iceland', 'Israel', 'Italy',
       'Jamaica', 'Jordan', 'Japan', 'Kazakhstan', 'Kenya', 'Kyrgyzstan',
       'Cambodia', 'Republic of Korea', 'Kuwait', "Lao People's DR",
       'Lebanon', 'Liberia', 'Libya', 'Saint Lucia', 'Sri Lanka',
       'Lesotho', 'Lithuania', 'Luxembourg', 'Latvia', 'Morocco',
       'Republic of Moldova', 'Madagascar', 'Mexico', 'TFYR of Macedonia',
       'Mali', 'Malta', 'Myanmar', 'Montenegro', 'Mongolia', 'Mozambique',
       'Mauritania', 'Mauritius', 'Malawi', 'Malaysia', 'Namibia',
       'Niger', 'Nigeria', 'Nicaragua', 'Netherlands', 'Norway', 'Nepal',
       'New Zealand', 'Oman', 'Pakistan', 'Panama', 'Peru', 'Philippines',
       'Poland', 'Puerto Rico', 'D.P.R. of Korea', 'Portugal', 'Paraguay',
       'State of Palestine', 'Qatar', 'Romania', 'Russian Federation',
       'Rwanda', 'Saudi Arabia', 'Sudan (Former)', 'Senegal', 'Singapore',
       'Sierra Leone', 'El Salvador', 'Serbia', 'Sao Tome and Principe',
       'Former USSR', 'Slovakia', 'Slovenia', 'Sweden', 'Swaziland',
       'Seychelles', 'Syrian Arab Republic', 'Chad', 'Togo', 'Thailand',
       'Tajikistan', 'Turkmenistan', 'Trinidad and Tobago', 'Tunisia',
```

```
       'Turkey', 'Taiwan, Province of China',
       'U.R. of Tanzania: Mainland', 'Uganda', 'Ukraine', 'Uruguay',
       'United States', 'Uzbekistan',
       'Venezuela (Bolivarian Republic of)', 'Viet Nam', 'Yemen',
       'Former Yugoslavia', 'South Africa', 'Zambia', 'Zimbabwe'],
      dtype=object)
```

我们现在可以探索一些可用的169个国家。

让我们遍历每个国家，以了解每个国家可用的年份。

```
country_years = []
for country in countries:
    cy_data = data[data.country == country]['year']
    ymin, ymax = cy_data.min(), cy_data.max()
    country_years.append((country, ymin, ymax))
country_years = pd.DataFrame(country_years,
                 columns=['country', 'min_year', 'max_year']).set_index('country')
country_years.head()
```

```
                      min_year  max_year
country
Afghanistan               1820      2018
Angola                    1950      2018
Albania                      1      2018
United Arab Emirates      1950      2018
Argentina                 1800      2018
```

现在，让我们将原始数据重塑为一些方便的变量，以便更快地访问各国的时间序列数据。

我们可以在该数据集中的国家代码（countrycode）和国家名称（country）之间建立一个有用的映射关系

```
code_to_name = data[
    ['countrycode', 'country']].drop_duplicates().reset_index(drop=True).set_index([
↪'countrycode'])
```

现在，我们调用人均 GDP（gdppc），并生成一个宽格式的数据

```
gdp_pc = data.set_index(['countrycode', 'year'])['gdppc']
gdp_pc = gdp_pc.unstack('countrycode')
```

```
gdp_pc.tail()
```

```
countrycode       AFG        AGO         ALB        ARE         ARG  \
year
2014        2022.0000  8673.0000   9808.0000  72601.0000  19183.0000
2015        1928.0000  8689.0000  10032.0000  74746.0000  19502.0000
2016        1929.0000  8453.0000  10342.0000  75876.0000  18875.0000
2017        2014.7453  8146.4354  10702.1201  76643.4984  19200.9061
2018        1934.5550  7771.4418  11104.1665  76397.8181  18556.3831

countrycode       ARM         AUS         AUT         AZE        BDI  ...  \
year                                                                  ...
2014         9735.0000  47867.0000  41338.0000  17439.0000  748.0000  ...
2015        10042.0000  48357.0000  41294.0000  17460.0000  694.0000  ...
2016        10080.0000  48845.0000  41445.0000  16645.0000  665.0000  ...
2017        10859.3783  49265.6135  42177.3706  16522.3072  671.3169  ...
2018        11454.4251  49830.7993  42988.0709  16628.0553  651.3589  ...

countrycode        URY         USA         UZB         VEN        VNM  \
year
2014        19160.0000  51664.0000   9085.0000  20317.0000  5455.0000
2015        19244.0000  52591.0000   9720.0000  18802.0000  5763.0000
2016        19468.0000  53015.0000  10381.0000  15219.0000  6062.0000
2017        19918.1361  54007.7698  10743.8666  12879.1350  6422.0865
2018        20185.8360  55334.7394  11220.3702  10709.9506  6814.1423

countrycode       YEM         YUG         ZAF        ZMB        ZWE
year
2014        4054.0000  14627.0000  12242.0000  3478.0000  1594.0000
2015        2844.0000  14971.0000  12246.0000  3478.0000  1560.0000
2016        2506.0000  15416.0000  12139.0000  3479.0000  1534.0000
2017        2321.9239  15960.8432  12189.3579  3497.5818  1582.3662
2018        2284.8899  16558.3123  12165.7948  3534.0337  1611.4052

[5 rows x 169 columns]
```

我们创建了一个变量 color_mapping 来存储国家代码和颜色之间的映射，以保证接下来线条的颜色一致

2.3 人均国内生产总值

在本节中，我们将研究几个不同国家的长期人均国内生产总值。

2.3.1 英国

首先我们研究英国的 GDP 增长情况

现在我们可以提取并绘制这些国家的 GDP 数据。

```python
fig, ax = plt.subplots(dpi=300)
country = 'GBR'
gdp_pc[country].plot(
        ax=ax,
        ylabel='international dollars',
        xlabel='year',
        color=color_mapping[country]
    );
```
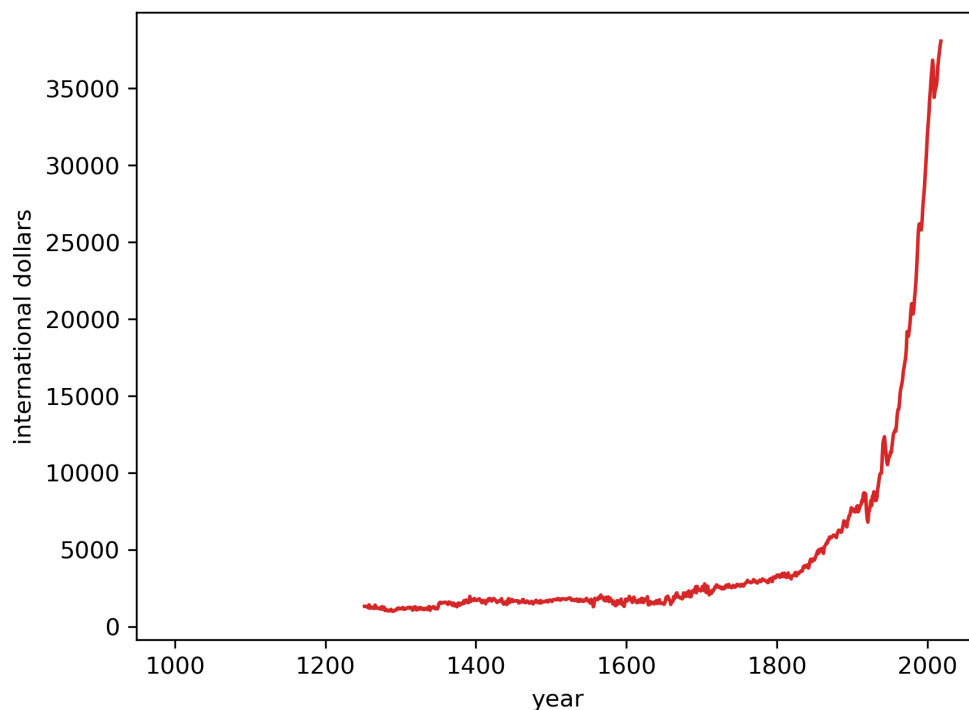

Fig. 2.1: GDP per Capita (GBR)

Note: 国际美元 是一种假设的货币单位，在特定时间点与美元在美国的购买力平价相同。它们也被称为 Geary-Khamis 美元（GK 美元）。

我们可以看到，在本千年早期的 250 年中，较长时期的数据是不连续的，因此我们可以选择内插法得到连续的线图。

在这里，我们用虚线表示插值趋势

```python
fig, ax = plt.subplots(dpi=300)
country = 'GBR'
ax.plot(gdp_pc[country].interpolate(),
        linestyle='--',
        lw=2,
        color=color_mapping[country])

ax.plot(gdp_pc[country],
        lw=2,
        color=color_mapping[country])
ax.set_ylabel('international dollars')
ax.set_xlabel('year')
plt.show()
```
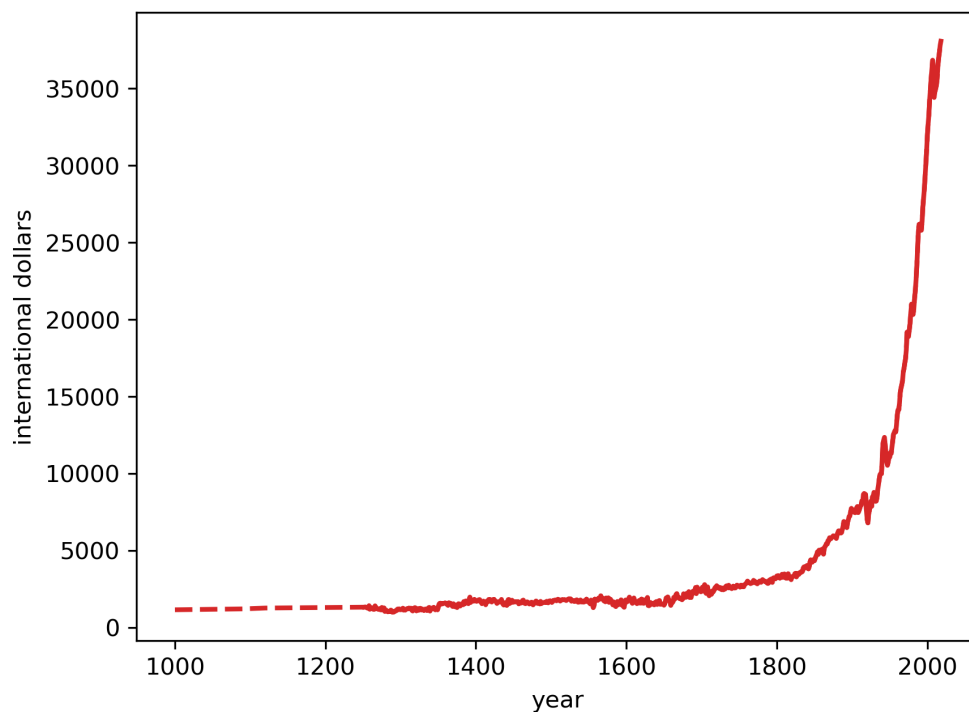

Fig. 2.2: GDP per Capita (GBR)

2.3.2 美国、英国和中国的比较

在本节中，我们将比较美国、英国和中国的GDP增长。

第一步，我们创建一个函数，为一组国家生成绘图

```python
def draw_interp_plots(series,          # pandas 数据
                      country,         # 国家代码列表
                      ylabel,          # y 轴标签
                      xlabel,          # x 轴标签
                      color_mapping,   # 代码- 颜色映射
                      code_to_name,    # 代码- 国家名映射
                      lw,              # 线宽
                      logscale,        # Y 轴是否要对数刻度
                      ax               # matplolib 轴
                     ):

    for c in country:
        # 获取插值数据
        df_interpolated = series[c].interpolate(limit_area='inside')
        interpolated_data = df_interpolated[series[c].isnull()]

        # 用虚线绘制插值数据
        ax.plot(interpolated_data,
                linestyle='--',
                lw=lw,
                alpha=0.7,
                color=color_mapping[c])

        # 用实线绘制非插值数据
        ax.plot(series[c],
                lw=lw,
                color=color_mapping[c],
                alpha=0.8,
                label=code_to_name.loc[c]['country'])

        if logscale:
            ax.set_yscale('log')

    # 将图例绘制在图外
    ax.legend(loc='upper left', frameon=False)
    ax.set_ylabel(ylabel)
```

(continues on next page)

```
    ax.set_xlabel(xlabel)
```

从图表中可以看出，经济增长从 18 世纪真正开始，并持续了两百年。

这与其他国家的增长轨迹相比如何？

让我们看看美国（USA）、英国（GBR）和中国（CHN）

```
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 33322 (\N{CJK UNIFIED IDEOGRAPH-822A})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 28023 (\N{CJK UNIFIED IDEOGRAPH-6D77})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 27861 (\N{CJK UNIFIED IDEOGRAPH-6CD5})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 26696 (\N{CJK UNIFIED IDEOGRAPH-6848})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 38381 (\N{CJK UNIFIED IDEOGRAPH-95ED})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 20851 (\N{CJK UNIFIED IDEOGRAPH-5173})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 38145 (\N{CJK UNIFIED IDEOGRAPH-9501})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 22269 (\N{CJK UNIFIED IDEOGRAPH-56FD})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 24223 (\N{CJK UNIFIED IDEOGRAPH-5E9F})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
```

```
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 38500 (\N{CJK UNIFIED IDEOGRAPH-9664})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 32654 (\N{CJK UNIFIED IDEOGRAPH-7F8E})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 29420 (\N{CJK UNIFIED IDEOGRAPH-72EC})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 31435 (\N{CJK UNIFIED IDEOGRAPH-7ACB})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 25112 (\N{CJK UNIFIED IDEOGRAPH-6218})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 20105 (\N{CJK UNIFIED IDEOGRAPH-4E89})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 24037 (\N{CJK UNIFIED IDEOGRAPH-5DE5})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 19994 (\N{CJK UNIFIED IDEOGRAPH-4E1A})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 38761 (\N{CJK UNIFIED IDEOGRAPH-9769})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 21629 (\N{CJK UNIFIED IDEOGRAPH-547D})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 22823 (\N{CJK UNIFIED IDEOGRAPH-5927})␣
↪missing from current font.
```

```
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 33831 (\N{CJK UNIFIED IDEOGRAPH-8427})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 26465 (\N{CJK UNIFIED IDEOGRAPH-6761})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 25913 (\N{CJK UNIFIED IDEOGRAPH-6539})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 24320 (\N{CJK UNIFIED IDEOGRAPH-5F00})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 25918 (\N{CJK UNIFIED IDEOGRAPH-653E})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
```
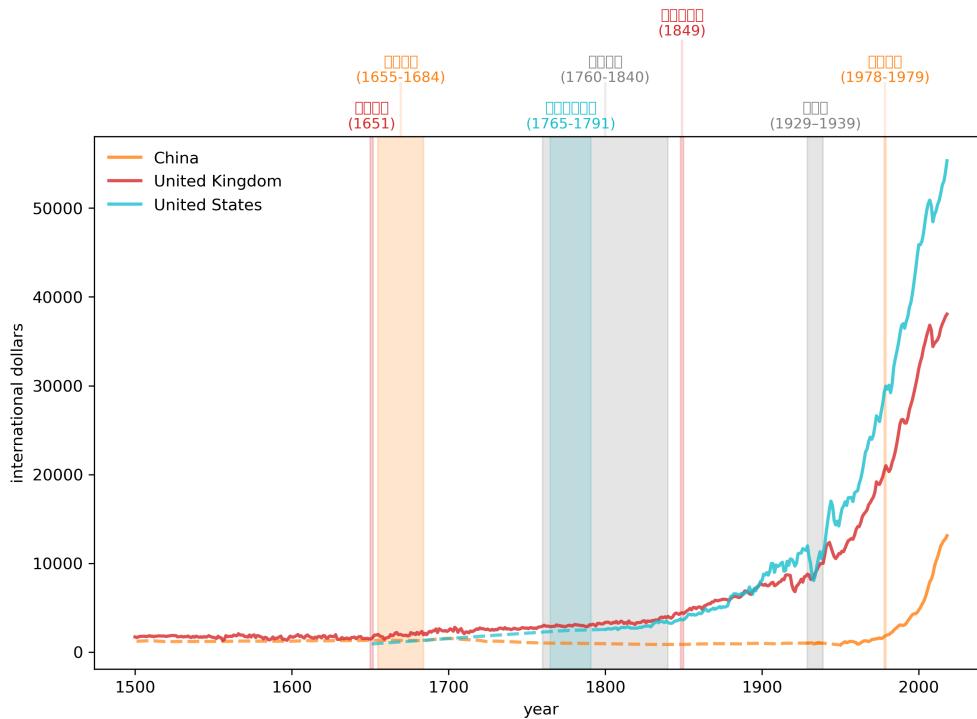

Fig. 2.3: 人均 GDP，1500-（中国、英国、美国）

上图展示了工业革命的传播如何随着时间的推移逐渐提高了大量人群的生活水平

- 大部分增长发生在工业革命之后的过去150年间。

- 从1820年到1940年，美国和英国的人均GDP上升并与中国拉开差距。

- 1950年后，尤其是1970年代末后，差距迅速缩小。

- 这些结果反映了技术和经济政策因素的复杂组合，经济增长的研究者们试图理解并量化这些因素。

2.3.3 聚焦于中国

下图所展示的中国从1500年到1970年的人均GDP水平非常有趣。

我们可以看到，从1700年到20世纪初，中国的人均GDP经历了长期的下降。

因此，图表显示了

- 清政府闭关锁国政策后的长期经济下行和停滞。

- 英国工业革命开始后，中国的经历与英国截然不同。

- 自强运动似乎主要帮助了中国的增长。

- 现代中国经济政策的惊人增长成果，最终在1970年代末进行的改革和开放政策中达到高潮。

```
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 24180 (\N{CJK UNIFIED IDEOGRAPH-5E74})↪
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 20221 (\N{CJK UNIFIED IDEOGRAPH-4EFD})↪
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 22269 (\N{CJK UNIFIED IDEOGRAPH-56FD})↪
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 38469 (\N{CJK UNIFIED IDEOGRAPH-9645})↪
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 32654 (\N{CJK UNIFIED IDEOGRAPH-7F8E})↪
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 20803 (\N{CJK UNIFIED IDEOGRAPH-5143})↪
↪missing from current font.
```

```
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 38381 (\N{CJK UNIFIED IDEOGRAPH-95ED})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 20851 (\N{CJK UNIFIED IDEOGRAPH-5173})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 38145 (\N{CJK UNIFIED IDEOGRAPH-9501})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 25919 (\N{CJK UNIFIED IDEOGRAPH-653F})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 31574 (\N{CJK UNIFIED IDEOGRAPH-7B56})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 24037 (\N{CJK UNIFIED IDEOGRAPH-5DE5})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 19994 (\N{CJK UNIFIED IDEOGRAPH-4E1A})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 38761 (\N{CJK UNIFIED IDEOGRAPH-9769})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 21629 (\N{CJK UNIFIED IDEOGRAPH-547D})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 31532 (\N{CJK UNIFIED IDEOGRAPH-7B2C})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 19968 (\N{CJK UNIFIED IDEOGRAPH-4E00})␣
↪missing from current font.
```

```
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 27425 (\N{CJK UNIFIED IDEOGRAPH-6B21})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 40486 (\N{CJK UNIFIED IDEOGRAPH-9E26})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 29255 (\N{CJK UNIFIED IDEOGRAPH-7247})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 25112 (\N{CJK UNIFIED IDEOGRAPH-6218})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 20105 (\N{CJK UNIFIED IDEOGRAPH-4E89})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 33258 (\N{CJK UNIFIED IDEOGRAPH-81EA})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 24378 (\N{CJK UNIFIED IDEOGRAPH-5F3A})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 36816 (\N{CJK UNIFIED IDEOGRAPH-8FD0})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 21160 (\N{CJK UNIFIED IDEOGRAPH-52A8})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 20108 (\N{CJK UNIFIED IDEOGRAPH-4E8C})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 19990 (\N{CJK UNIFIED IDEOGRAPH-4E16})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
```

```
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 30028 (\N{CJK UNIFIED IDEOGRAPH-754C})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 22823 (\N{CJK UNIFIED IDEOGRAPH-5927})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 20013 (\N{CJK UNIFIED IDEOGRAPH-4E2D})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 21326 (\N{CJK UNIFIED IDEOGRAPH-534E})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 20154 (\N{CJK UNIFIED IDEOGRAPH-4EBA})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 27665 (\N{CJK UNIFIED IDEOGRAPH-6C11})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 20849 (\N{CJK UNIFIED IDEOGRAPH-5171})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 21644 (\N{CJK UNIFIED IDEOGRAPH-548C})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 25104 (\N{CJK UNIFIED IDEOGRAPH-6210})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 31435 (\N{CJK UNIFIED IDEOGRAPH-7ACB})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 36291 (\N{CJK UNIFIED IDEOGRAPH-8DC3})␣
↪missing from current font.
```

```
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
 ↪pylabtools.py:152: UserWarning: Glyph 36827 (\N{CJK UNIFIED IDEOGRAPH-8FDB})↵
 ↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
 ↪pylabtools.py:152: UserWarning: Glyph 25913 (\N{CJK UNIFIED IDEOGRAPH-6539})↵
 ↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
 ↪pylabtools.py:152: UserWarning: Glyph 24320 (\N{CJK UNIFIED IDEOGRAPH-5F00})↵
 ↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
 ↪pylabtools.py:152: UserWarning: Glyph 25918 (\N{CJK UNIFIED IDEOGRAPH-653E})↵
 ↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
```


Fig. 2.4: 人均GDP，1500-2000（中国）

2.3.4 聚焦于美国和英国

现在我们更详细地研究美国（USA）和英国（GBR）。

在以下图表中，请注意

- 贸易政策（航海法案）的影响。

- 工业革命带来的生产力变化。

- 美国逐渐接近并超过英国，为"美国世纪"奠定基础。

- 战争的意外后果。

- 经济周期衰退和萧条带来的中断和伤痕。

```
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
 ↪pylabtools.py:152: UserWarning: Glyph 24180 (\N{CJK UNIFIED IDEOGRAPH-5E74})␣
 ↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
 ↪pylabtools.py:152: UserWarning: Glyph 20221 (\N{CJK UNIFIED IDEOGRAPH-4EFD})␣
 ↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
 ↪pylabtools.py:152: UserWarning: Glyph 22269 (\N{CJK UNIFIED IDEOGRAPH-56FD})␣
 ↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
 ↪pylabtools.py:152: UserWarning: Glyph 38469 (\N{CJK UNIFIED IDEOGRAPH-9645})␣
 ↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
 ↪pylabtools.py:152: UserWarning: Glyph 32654 (\N{CJK UNIFIED IDEOGRAPH-7F8E})␣
 ↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
 ↪pylabtools.py:152: UserWarning: Glyph 20803 (\N{CJK UNIFIED IDEOGRAPH-5143})␣
 ↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
 ↪pylabtools.py:152: UserWarning: Glyph 33322 (\N{CJK UNIFIED IDEOGRAPH-822A})␣
 ↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
 ↪pylabtools.py:152: UserWarning: Glyph 28023 (\N{CJK UNIFIED IDEOGRAPH-6D77})␣
 ↪missing from current font.
```
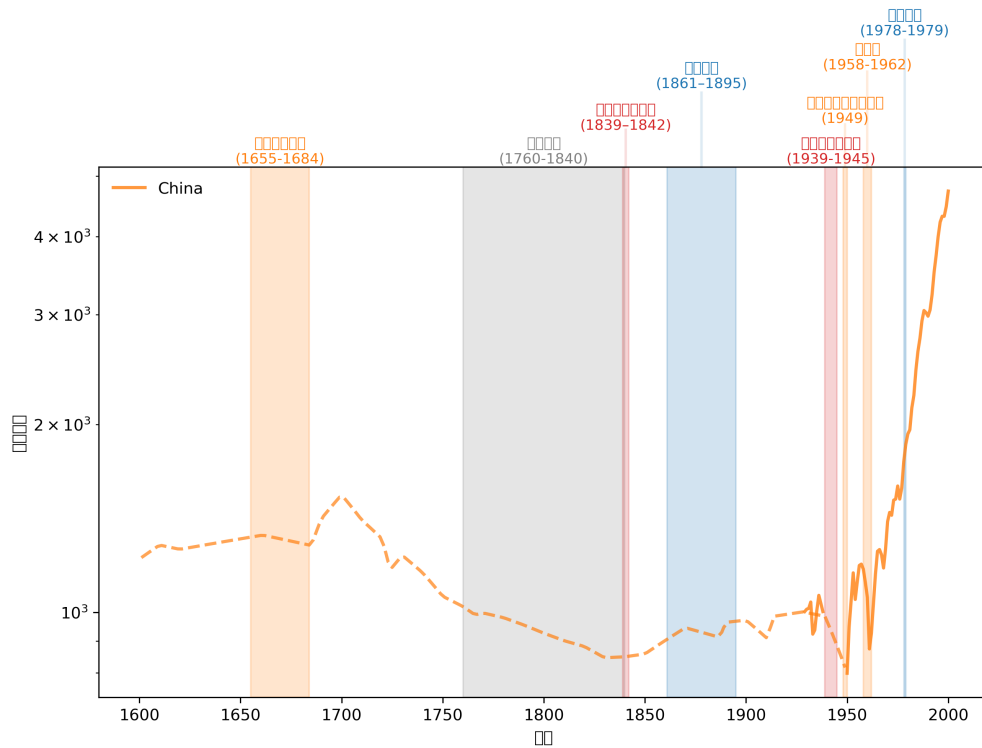
```
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 27861 (\N{CJK UNIFIED IDEOGRAPH-6CD5})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 65288 (\N{FULLWIDTH LEFT PARENTHESIS})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 33521 (\N{CJK UNIFIED IDEOGRAPH-82F1})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 65289 (\N{FULLWIDTH RIGHT PARENTHESIS})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 29420 (\N{CJK UNIFIED IDEOGRAPH-72EC})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 31435 (\N{CJK UNIFIED IDEOGRAPH-7ACB})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 25112 (\N{CJK UNIFIED IDEOGRAPH-6218})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 20105 (\N{CJK UNIFIED IDEOGRAPH-4E89})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 24037 (\N{CJK UNIFIED IDEOGRAPH-5DE5})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 19994 (\N{CJK UNIFIED IDEOGRAPH-4E1A})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 38761 (\N{CJK UNIFIED IDEOGRAPH-9769})␣
↪missing from current font.
```

```
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 21629 (\N{CJK UNIFIED IDEOGRAPH-547D})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 24223 (\N{CJK UNIFIED IDEOGRAPH-5E9F})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 38500 (\N{CJK UNIFIED IDEOGRAPH-9664})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 20869 (\N{CJK UNIFIED IDEOGRAPH-5185})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 31532 (\N{CJK UNIFIED IDEOGRAPH-7B2C})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 19968 (\N{CJK UNIFIED IDEOGRAPH-4E00})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 27425 (\N{CJK UNIFIED IDEOGRAPH-6B21})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 19990 (\N{CJK UNIFIED IDEOGRAPH-4E16})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 30028 (\N{CJK UNIFIED IDEOGRAPH-754C})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 22823 (\N{CJK UNIFIED IDEOGRAPH-5927})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 33831 (\N{CJK UNIFIED IDEOGRAPH-8427})␣
↪missing from current font.
```

```
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 26465 (\N{CJK UNIFIED IDEOGRAPH-6761})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 20108 (\N{CJK UNIFIED IDEOGRAPH-4E8C})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
```
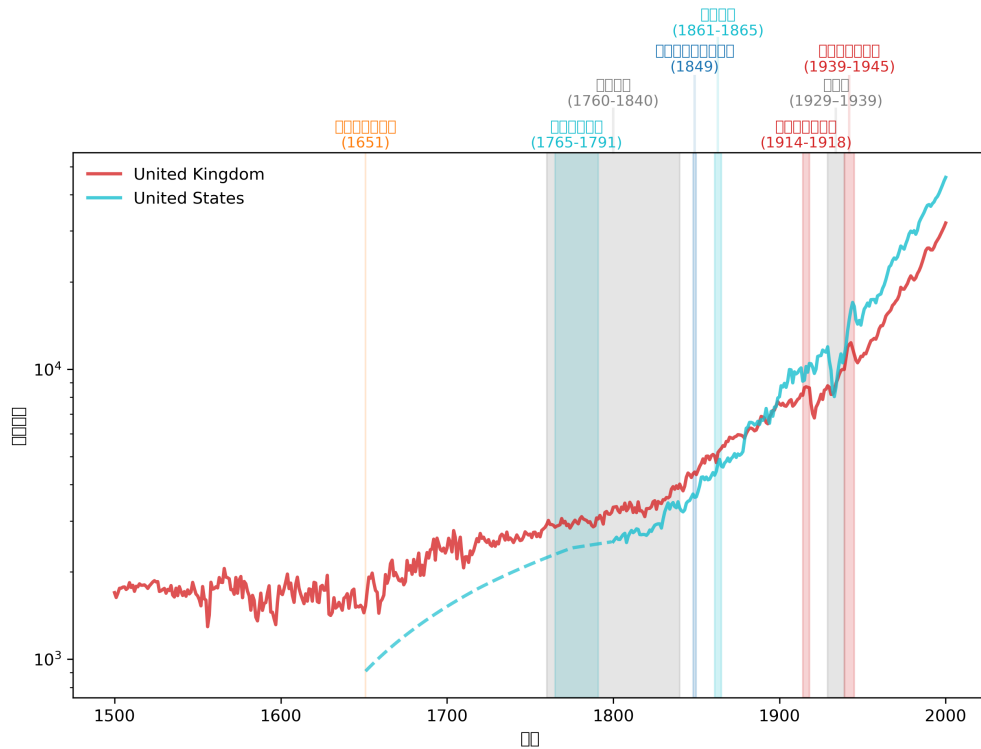

Fig. 2.5: 人均GDP，1500-2000（英国和美国）

2.4 GDP增长

现在我们将构建一些对地缘政治历史学家如亚当·图兹感兴趣的图表。

我们将关注总国内生产总值（GDP）（作为"国家地缘政治-军事力量"的代理变量），而不是人均GDP（作为生活水平的代理变量）。

```
data = pd.read_excel(data_url, sheet_name='Full data')
data.set_index(['countrycode', 'year'], inplace=True)
data['gdp'] = data['gdppc'] * data['pop']
gdp = data['gdp'].unstack('countrycode')
```

2.4.1 早期工业化（1820至1940年）

我们首先可视化中国、俄国/前苏联、日本、英国和美国的趋势。

最显著的趋势是美国的崛起，在1860年代超过英国，并在1880年代超过中国。

这种增长持续到1930年代的大萧条到来时出现大幅下降。

与此同时，俄国在第一次世界大战期间经历了重大挫折，并在二月革命后显著恢复。

```
fig, ax = plt.subplots(dpi=300)
country = ['CHN', 'SUN', 'JPN', 'GBR', 'USA']
start_year, end_year = (1820, 1945)
draw_interp_plots(gdp[country].loc[start_year:end_year],
                  country,
                  '国际美元', '年份',
                  color_mapping, code_to_name, 2, False, ax)
```

```
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
 ↪events.py:82: UserWarning: Glyph 24180 (\N{CJK UNIFIED IDEOGRAPH-5E74}) missing↪
 ↪from current font.
  func(*args, **kwargs)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
 ↪events.py:82: UserWarning: Glyph 20221 (\N{CJK UNIFIED IDEOGRAPH-4EFD}) missing↪
 ↪from current font.
  func(*args, **kwargs)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
 ↪events.py:82: UserWarning: Glyph 22269 (\N{CJK UNIFIED IDEOGRAPH-56FD}) missing↪
 ↪from current font.
  func(*args, **kwargs)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
 ↪events.py:82: UserWarning: Glyph 38469 (\N{CJK UNIFIED IDEOGRAPH-9645}) missing↪
 ↪from current font.
  func(*args, **kwargs)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
 ↪events.py:82: UserWarning: Glyph 32654 (\N{CJK UNIFIED IDEOGRAPH-7F8E}) missing↪
 ↪from current font.
  func(*args, **kwargs)
```

```
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪events.py:82: UserWarning: Glyph 20803 (\N{CJK UNIFIED IDEOGRAPH-5143}) missing␣
↪from current font.
  func(*args, **kwargs)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 24180 (\N{CJK UNIFIED IDEOGRAPH-5E74})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 20221 (\N{CJK UNIFIED IDEOGRAPH-4EFD})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 22269 (\N{CJK UNIFIED IDEOGRAPH-56FD})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 38469 (\N{CJK UNIFIED IDEOGRAPH-9645})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 32654 (\N{CJK UNIFIED IDEOGRAPH-7F8E})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 20803 (\N{CJK UNIFIED IDEOGRAPH-5143})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
```

构建类似于图兹的图

在本节中，我们将介绍如何构建本讲座开始时讨论过的 [Tooze, 2014] 第1章中引人注目的图表。

首先，让我们定义一个由大英帝国（BEM）组成的国家集合，这样我们就可以用Tooze的数据中复制绘制这幅图

```python
BEM = ['GBR', 'IND', 'AUS', 'NZL', 'CAN', 'ZAF']
# 插值不完全时间序列
gdp['BEM'] = gdp[BEM].loc[start_year-1:end_year].interpolate(method='index').
↪sum(axis=1)
```
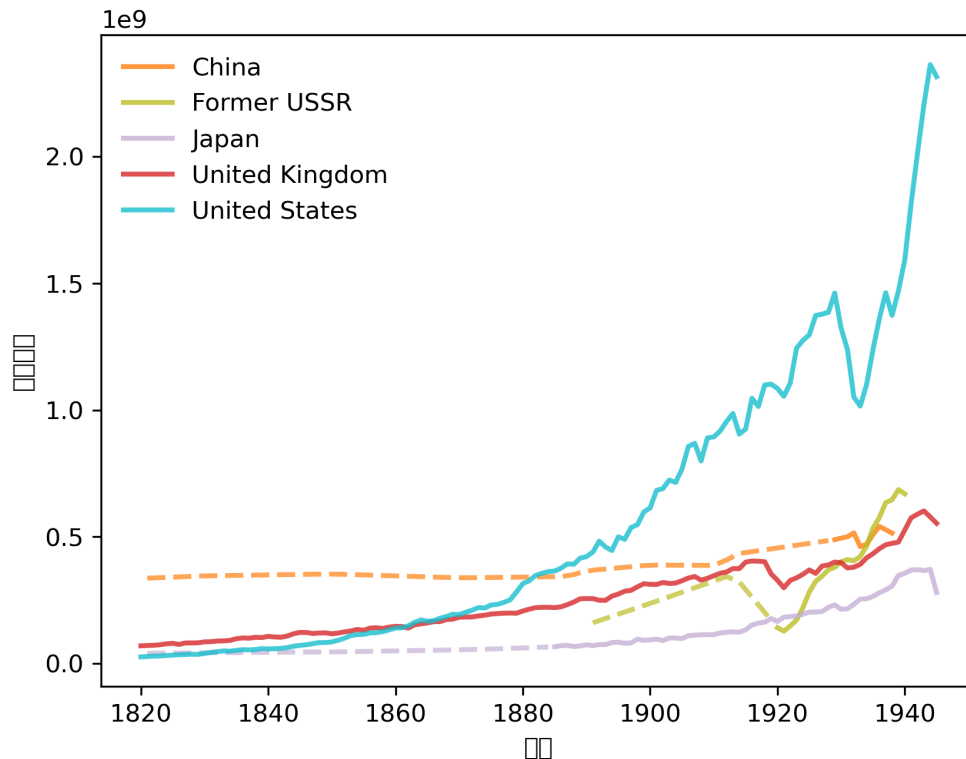
现在，让我们组装这些数据序列

Fig. 2.6: 早期工业化时代的GDP

```
color_mapping['BEM'] = color_mapping['GBR']
bem = pd.DataFrame(["British Empire"], index=["BEM"], columns=['country'])
bem.index.name = 'countrycode'
code_to_name = pd.concat([code_to_name, bem])
```

并绘制他们

```
fig, ax = plt.subplots(dpi=300)
country = ['DEU', 'USA', 'SUN', 'BEM', 'FRA', 'JPN']
start_year, end_year = (1821, 1945)
draw_interp_plots(gdp[country].loc[start_year:end_year],
                  country,
                  'international dollars', 'year',
                  color_mapping, code_to_name, 2, False, ax)

plt.savefig("./_static/lecture_specific/long_run_growth/tooze_ch1_graph.png", dpi=300,
            bbox_inches='tight')
plt.show()
```
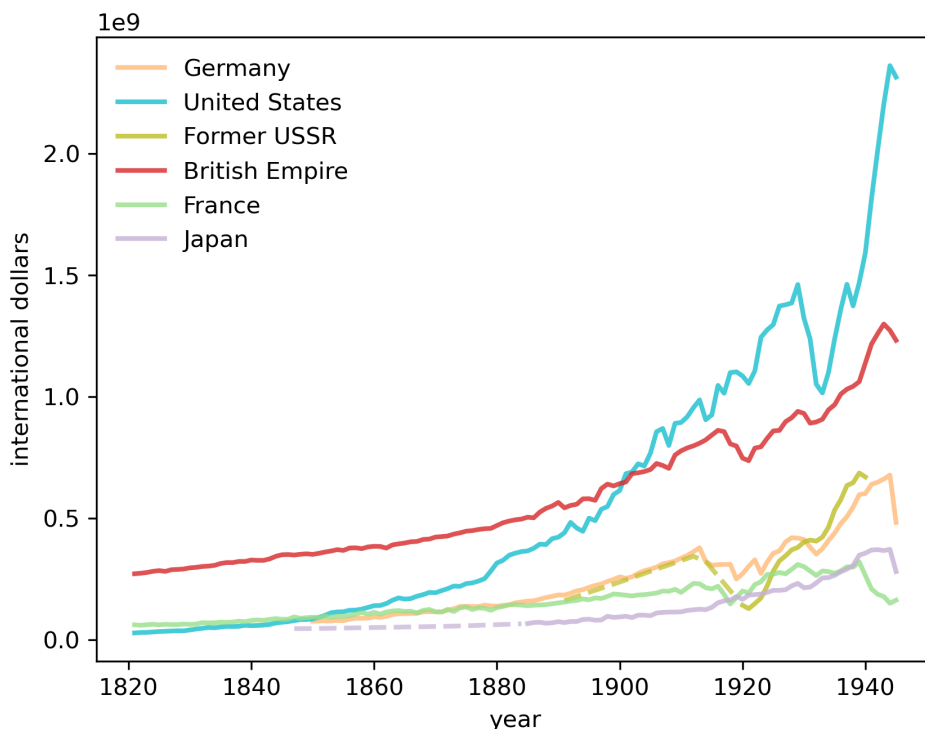
在本讲座的开头，我们提到了美国的 GDP 是如何在 19 世纪初"从无到有"，到 19 世纪末与大英帝国的 GDP 相媲美，进而超越大英帝国的 GDP，为"美国的二十世纪"奠定了地缘政治的基础。

让我们将时间向前推移，大致从第二次世界大战后图示停止的地方开始。

按照图兹第一章的分析，这样做可以提供一些有关当今地缘政治现实的信息。

2.4.2 现代时代（1950至2020年）

下图显示了中国自二十世纪七十年代后期以来增长的迅速。

```python
fig, ax = plt.subplots(dpi=300)
country = ['CHN', 'SUN', 'JPN', 'GBR', 'USA']
start_year, end_year = (1950, 2020)
draw_interp_plots(gdp[country].loc[start_year:end_year],
                  country,
                  '国际美元', '年份',
                  color_mapping, code_to_name, 2, False, ax)
```

```
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪events.py:82: UserWarning: Glyph 24180 (\N{CJK UNIFIED IDEOGRAPH-5E74}) missing↪
↪from current font.
  func(*args, **kwargs)
```

(continues on next page)

```
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪events.py:82: UserWarning: Glyph 20221 (\N{CJK UNIFIED IDEOGRAPH-4EFD}) missing␣
↪from current font.
  func(*args, **kwargs)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪events.py:82: UserWarning: Glyph 22269 (\N{CJK UNIFIED IDEOGRAPH-56FD}) missing␣
↪from current font.
  func(*args, **kwargs)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪events.py:82: UserWarning: Glyph 38469 (\N{CJK UNIFIED IDEOGRAPH-9645}) missing␣
↪from current font.
  func(*args, **kwargs)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪events.py:82: UserWarning: Glyph 32654 (\N{CJK UNIFIED IDEOGRAPH-7F8E}) missing␣
↪from current font.
  func(*args, **kwargs)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪events.py:82: UserWarning: Glyph 20803 (\N{CJK UNIFIED IDEOGRAPH-5143}) missing␣
↪from current font.
  func(*args, **kwargs)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 24180 (\N{CJK UNIFIED IDEOGRAPH-5E74})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 20221 (\N{CJK UNIFIED IDEOGRAPH-4EFD})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 22269 (\N{CJK UNIFIED IDEOGRAPH-56FD})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 38469 (\N{CJK UNIFIED IDEOGRAPH-9645})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 32654 (\N{CJK UNIFIED IDEOGRAPH-7F8E})␣
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 20803 (\N{CJK UNIFIED IDEOGRAPH-5143})␣
↪missing from current font.
```

```
fig.canvas.print_figure(bytes_io, **kw)
```
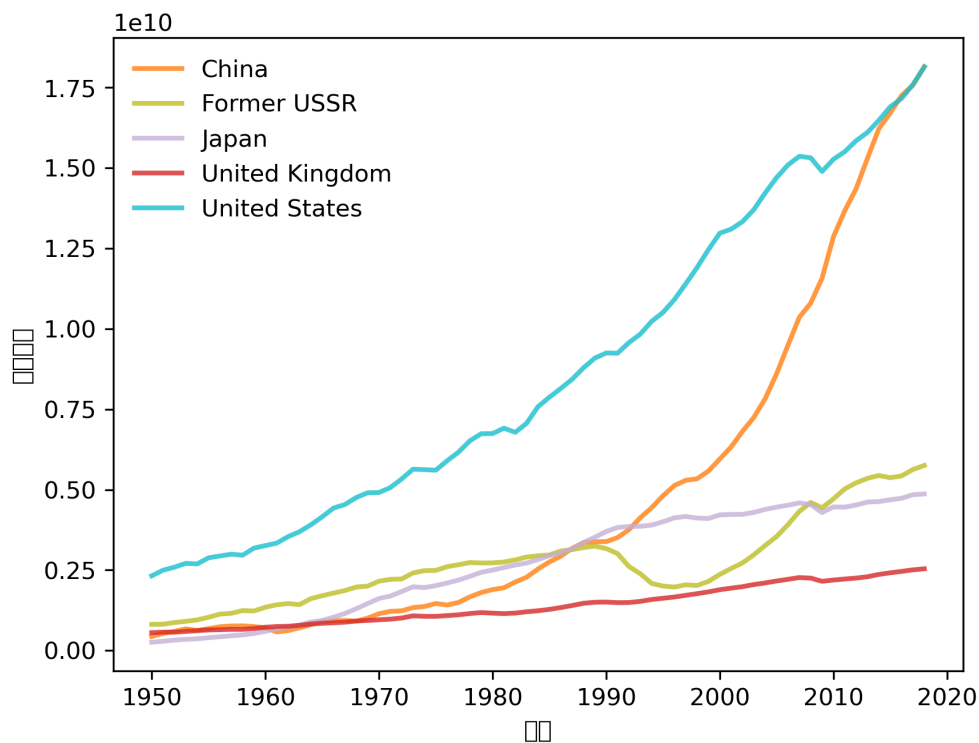

Fig. 2.7: 现代时代的GDP

很自然的我们会将这个图表与图Fig. 2.6相比较，后者显示了美国在"美国世纪"开始时超过英国的情况。

2.5 地区分析

我们经常要研究"世界大国"俱乐部之外的国家的历史。

Maddison Historical Statistics 数据集还整合了地区汇总信息

```
data = pd.read_excel(data_url,
                     sheet_name='Regional data',
                     header=(0,1,2),
                     index_col=0)
data.columns = data.columns.droplevel(level=2)
```

我们可以将原始数据存储在更方便的格式中来构建地区人均GDP的单一表格

```
regionalgdp_pc = data['gdppc_2011'].copy()
regionalgdp_pc.index = pd.to_datetime(regionalgdp_pc.index, format='%Y')
```

我们基于时间进行插值来填补数据集中的任何缺口，以便于绘图

```
regionalgdp_pc.interpolate(method='time', inplace=True)
```

进行更深入的研究，我们将西方分支（Western Offshoots）和（撒哈拉以南非洲）Sub-Saharan Africa 的时间序列与世界各地多个不同地区进行比较。

下图再次展示了工业革命之后西方与世界其他地区的差距，以及1950年之后的世界趋同

```
fig, ax = plt.subplots(dpi=300)
regionalgdp_pc.plot(ax=ax, xlabel='年份',
                    lw=2,
                    ylabel='国际美元')
ax.set_yscale('log')
plt.legend(loc='lower center',
           ncol=3, bbox_to_anchor=[0.5, -0.5])
plt.show()
```

```
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 24180 (\N{CJK UNIFIED IDEOGRAPH-5E74})↵
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 20221 (\N{CJK UNIFIED IDEOGRAPH-4EFD})↵
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 22269 (\N{CJK UNIFIED IDEOGRAPH-56FD})↵
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 38469 (\N{CJK UNIFIED IDEOGRAPH-9645})↵
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 32654 (\N{CJK UNIFIED IDEOGRAPH-7F8E})↵
↪missing from current font.
  fig.canvas.print_figure(bytes_io, **kw)
/Users/mmcky/anaconda3/envs/quantecon/lib/python3.11/site-packages/IPython/core/
↪pylabtools.py:152: UserWarning: Glyph 20803 (\N{CJK UNIFIED IDEOGRAPH-5143})↵
↪missing from current font.
```

(continues on next page)

```
fig.canvas.print_figure(bytes_io, **kw)
```
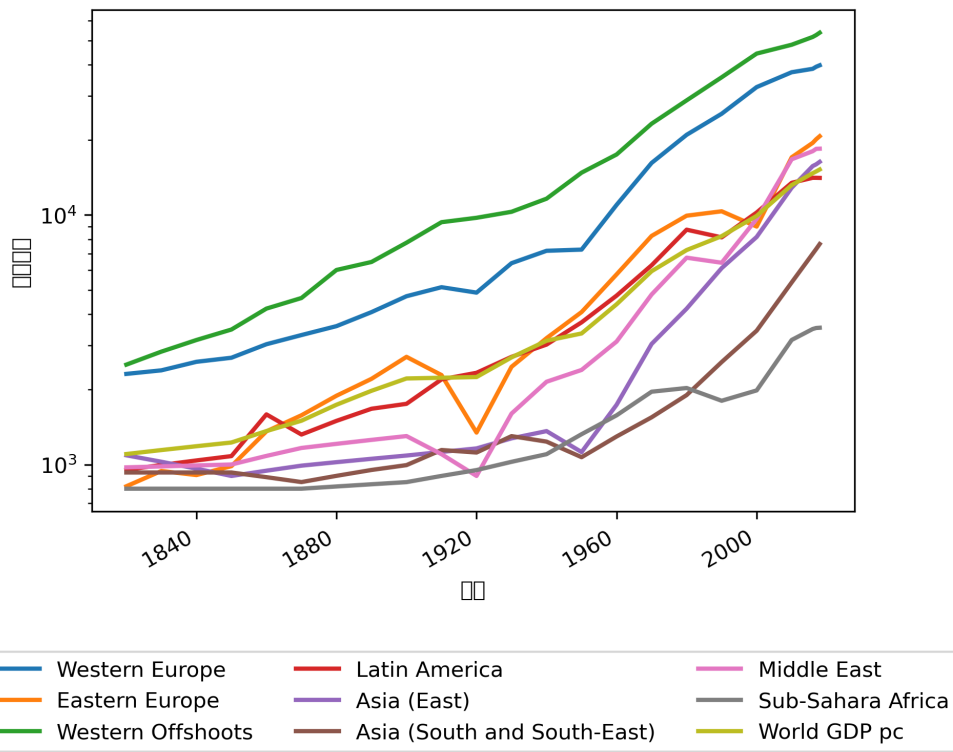

Fig. 2.8: 地区人均GDP

BUSINESS CYCLES

3.1 Overview

In this lecture we review some empirical aspects of business cycles.

Business cycles are fluctuations in economic activity over time.

These include expansions (also called booms) and contractions (also called recessions).

For our study, we will use economic indicators from the World Bank and FRED.

In addition to the packages already installed by Anaconda, this lecture requires

```
!pip install wbgapi
!pip install pandas-datareader
```

We use the following imports

```python
import matplotlib.pyplot as plt
import pandas as pd
import datetime
import wbgapi as wb
import pandas_datareader.data as web
```

Here's some minor code to help with colors in our plots.

3.2 Data acquisition

We will use the World Bank's data API `wbgapi` and `pandas_datareader` to retrieve data.

We can use `wb.series.info` with the argument `q` to query available data from the World Bank.

For example, let's retrieve the GDP growth data ID to query GDP growth data.

```
wb.series.info(q='GDP growth')
```

```
id                value
----------------  --------------------
NY.GDP.MKTP.KD.ZG  GDP growth (annual %)
                  1 elements
```

Now we use this series ID to obtain the data.

```
gdp_growth = wb.data.DataFrame('NY.GDP.MKTP.KD.ZG',
           ['USA', 'ARG', 'GBR', 'GRC', 'JPN'],
           labels=True)
gdp_growth
```

```
                Country   YR1960     YR1961     YR1962     YR1963     YR1964  \
economy
JPN               Japan      NaN  12.043536   8.908973   8.473642  11.676708
GRC              Greece      NaN  13.203841   0.364811  11.844867   9.409678
GBR      United Kingdom      NaN   2.701314   1.098696   4.859545   5.594811
ARG           Argentina      NaN   5.427843  -0.852022  -5.308197  10.130298
USA       United States      NaN   2.300000   6.100000   4.400000   5.800000


            YR1965     YR1966     YR1967     YR1968  ...     YR2014     YR2015  \
economy                                             ...
JPN       5.819708  10.638562  11.082142  12.882468  ...   0.296206   1.560627
GRC      10.768010   6.494501   5.669486   7.203719  ...   0.475696  -0.196088
GBR       2.130333   1.567450   2.775738   5.472693  ...   3.195782   2.220258
ARG      10.569433  -0.659726   3.191997   4.822501  ...  -2.512615   2.731160
USA       6.400000   6.500000   2.500000   4.800000  ...   2.523820   2.945550


            YR2016    YR2017    YR2018    YR2019     YR2020    YR2021  \
economy
JPN       0.753827  1.675332  0.643391 -0.402169  -4.147119  2.559320
GRC      -0.487173  1.092149  1.668429  1.879490  -9.316436  8.379944
GBR       1.921063  2.655070  1.403792  1.641611 -10.359901  8.674904
```

(continues on next page)

```
ARG    -2.080328  2.818503 -2.617396 -2.000861  -9.900485  10.718010
USA     1.819451  2.457622  2.966505  2.467038  -2.213469   5.800206


          YR2022    YR2023
economy
JPN     0.954737  1.923056
GRC     5.557287  2.011521
GBR     4.345229  0.104018
ARG     4.956370 -1.550502
USA     1.935496  2.542700


[5 rows x 65 columns]
```

We can look at the series' metadata to learn more about the series (click to expand).

```
wb.series.metadata.get('NY.GDP.MKTP.KD.ZG')
```

3.3 GDP growth rate

First we look at GDP growth.

Let's source our data from the World Bank and clean it.

```python
# Use the series ID retrieved before
gdp_growth = wb.data.DataFrame('NY.GDP.MKTP.KD.ZG',
          ['USA', 'ARG', 'GBR', 'GRC', 'JPN'],
          labels=True)
gdp_growth = gdp_growth.set_index('Country')
gdp_growth.columns = gdp_growth.columns.str.replace('YR', '').astype(int)
```

Here's a first look at the data

```
gdp_growth
```

```
                    1960      1961      1962      1963      1964       1965  \
Country
Japan               NaN  12.043536  8.908973   8.473642  11.676708   5.819708
Greece              NaN  13.203841  0.364811  11.844867   9.409678  10.768010
United Kingdom      NaN   2.701314  1.098696   4.859545   5.594811   2.130333
Argentina           NaN   5.427843 -0.852022  -5.308197  10.130298  10.569433
```

```
United States    NaN    2.300000  6.100000    4.400000    5.800000    6.400000


                    1966        1967        1968        1969  ...        2014  \
Country                                                       ...
Japan           10.638562  11.082142  12.882468  12.477895  ...   0.296206
Greece           6.494501   5.669486   7.203719  11.563668  ...   0.475696
United Kingdom   1.567450   2.775738   5.472693   1.939138  ...   3.195782
Argentina       -0.659726   3.191997   4.822501   9.679526  ...  -2.512615
United States    6.500000   2.500000   4.800000   3.100000  ...   2.523820


                    2015       2016       2017      2018      2019        2020  \
Country
Japan            1.560627   0.753827   1.675332  0.643391 -0.402169   -4.147119
Greece          -0.196088  -0.487173   1.092149  1.668429  1.879490   -9.316436
United Kingdom   2.220258   1.921063   2.655070  1.403792  1.641611  -10.359901
Argentina        2.731160  -2.080328   2.818503 -2.617396 -2.000861   -9.900485
United States    2.945550   1.819451   2.457622  2.966505  2.467038   -2.213469


                    2021       2022       2023
Country
Japan            2.559320   0.954737   1.923056
Greece           8.379944   5.557287   2.011521
United Kingdom   8.674904   4.345229   0.104018
Argentina       10.718010   4.956370  -1.550502
United States    5.800206   1.935496   2.542700

[5 rows x 64 columns]
```

We write a function to generate plots for individual countries taking into account the recessions.

Let's start with the United States.

```python
fig, ax = plt.subplots()


country = 'United States'
ylabel = 'GDP growth rate (%)'
plot_series(gdp_growth, country,
            ylabel, 0.1, ax,
            g_params, b_params, t_params)
plt.show()
```

GDP growth is positive on average and trending slightly downward over time.
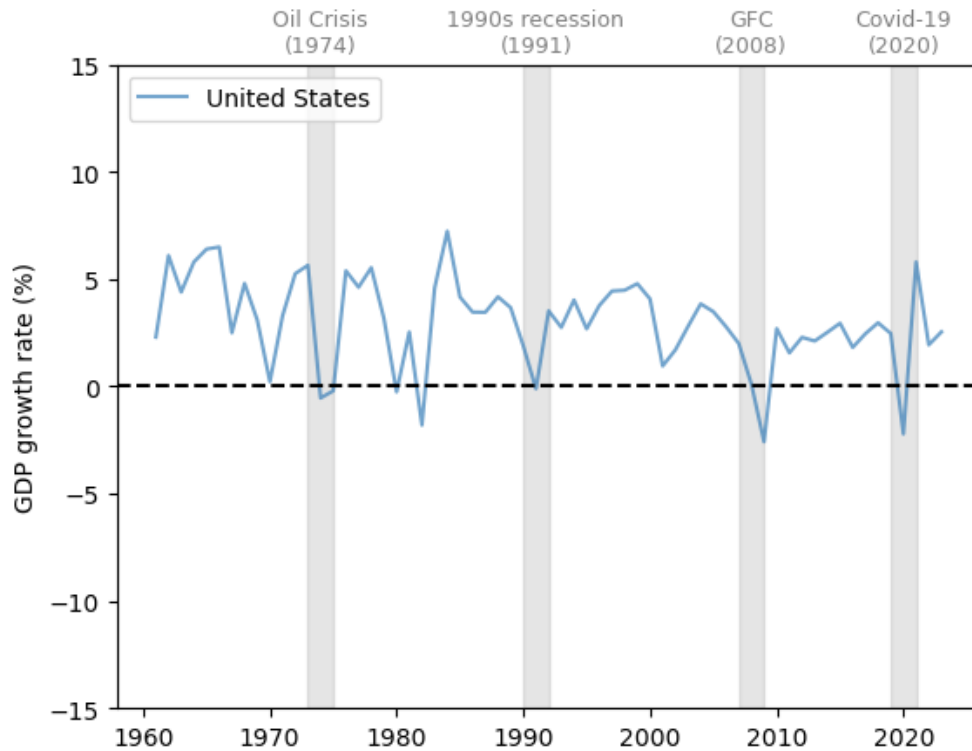
Fig. 3.1: United States (GDP growth rate %)

We also see fluctuations over GDP growth over time, some of which are quite large.

Let's look at a few more countries to get a basis for comparison.

The United Kingdom (UK) has a similar pattern to the US, with a slow decline in the growth rate and significant fluctuations.

Notice the very large dip during the Covid-19 pandemic.

```python
fig, ax = plt.subplots()

country = 'United Kingdom'
plot_series(gdp_growth, country,
            ylabel, 0.1, ax,
            g_params, b_params, t_params)
plt.show()
```

Now let's consider Japan, which experienced rapid growth in the 1960s and 1970s, followed by slowed expansion in the past two decades.

Major dips in the growth rate coincided with the Oil Crisis of the 1970s, the Global Financial Crisis (GFC) and the
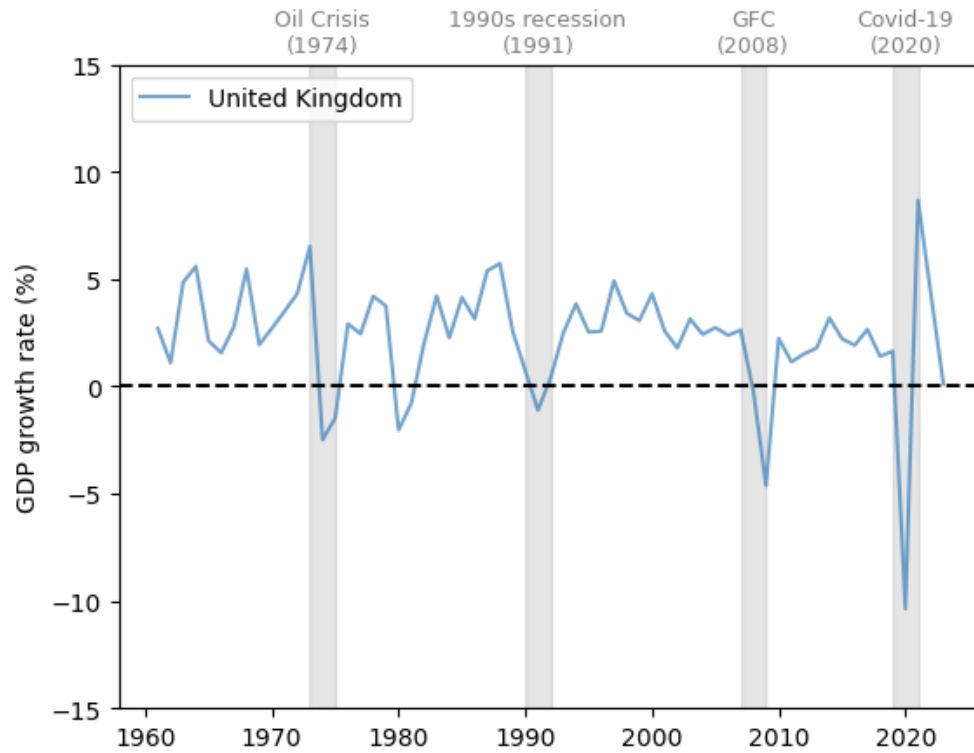
Fig. 3.2: United Kingdom (GDP growth rate %)

Covid-19 pandemic.

```
fig, ax = plt.subplots()

country = 'Japan'
plot_series(gdp_growth, country,
            ylabel, 0.1, ax,
            g_params, b_params, t_params)
plt.show()
```

Now let's study Greece.

```
fig, ax = plt.subplots()

country = 'Greece'
plot_series(gdp_growth, country,
            ylabel, 0.1, ax,
            g_params, b_params, t_params)
plt.show()
```
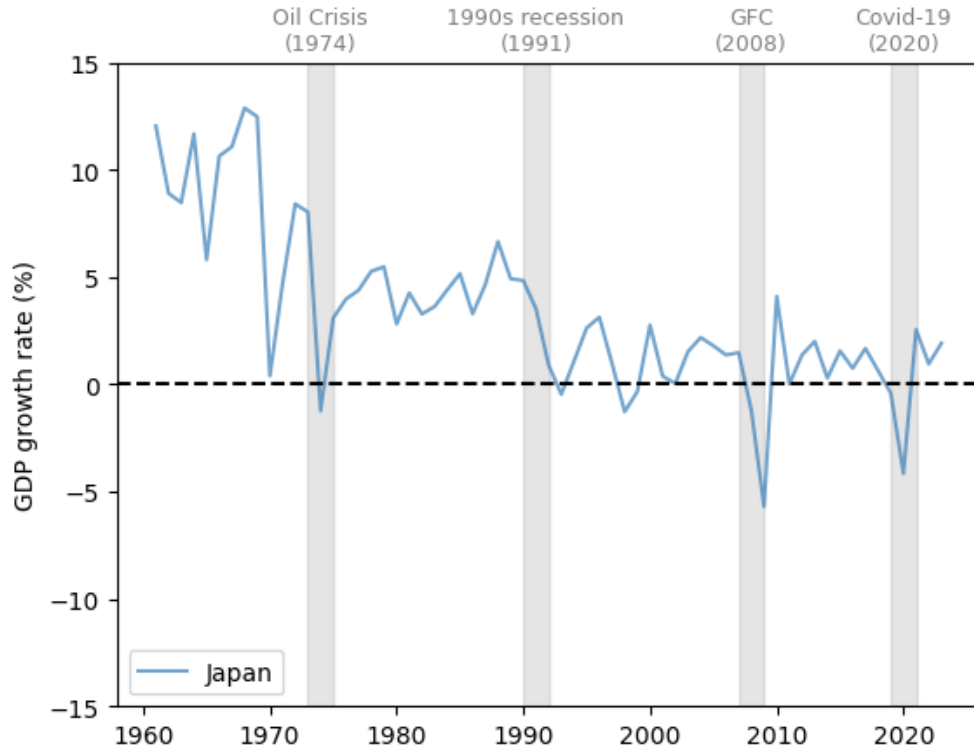
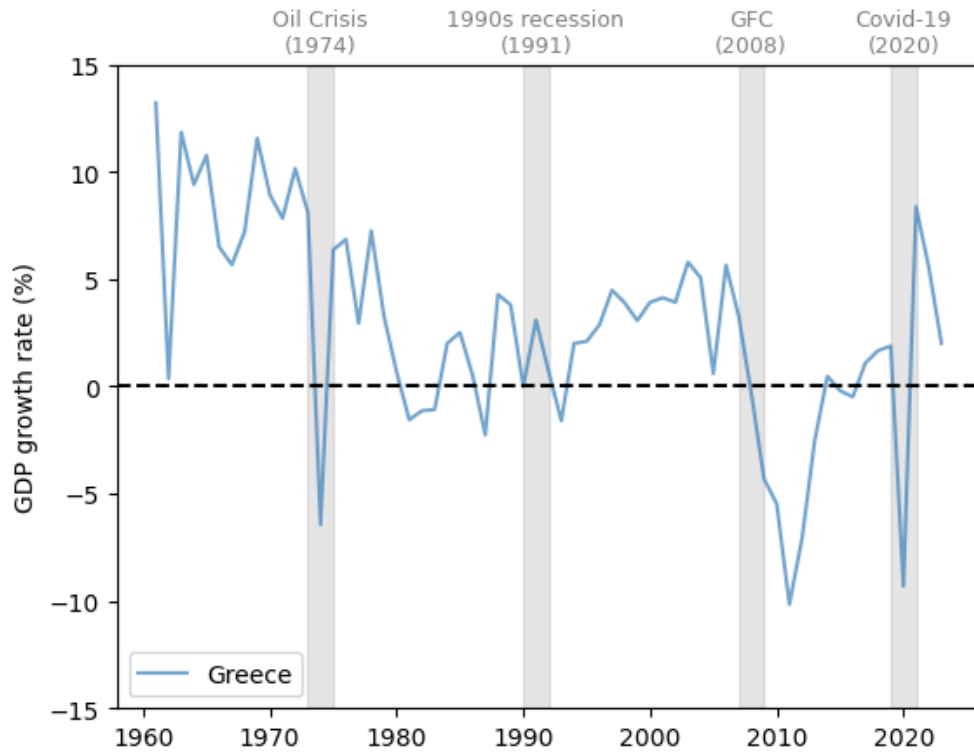Fig. 3.3: Japan (GDP growth rate %)

Fig. 3.4: Greece (GDP growth rate %)

Greece experienced a very large drop in GDP growth around 2010-2011, during the peak of the Greek debt crisis.

Next let's consider Argentina.

```
fig, ax = plt.subplots()

country = 'Argentina'
plot_series(gdp_growth, country,
            ylabel, 0.1, ax,
            g_params, b_params, t_params)
plt.show()
```
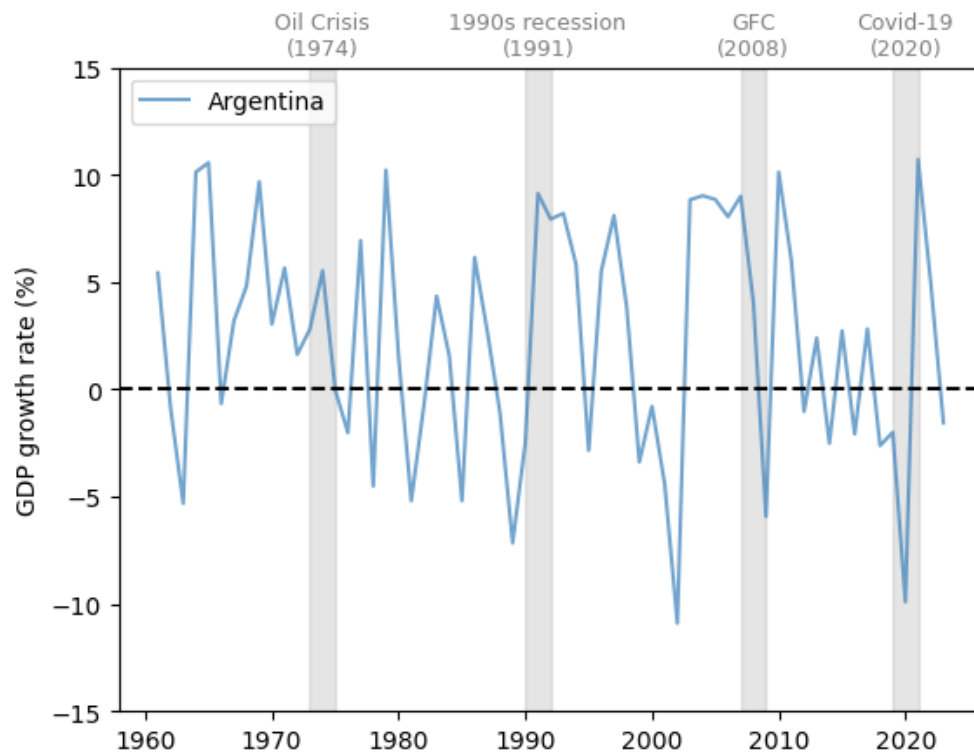

Fig. 3.5: Argentina (GDP growth rate %)

Notice that Argentina has experienced far more volatile cycles than the economies examined above.

At the same time, Argentina's growth rate did not fall during the two developed economy recessions in the 1970s and 1990s.

3.4 Unemployment

Another important measure of business cycles is the unemployment rate.

We study unemployment using rate data from FRED spanning from 1929-1942 to 1948-2022, combined unemployment rate data over 1942-1948 estimated by the Census Bureau.

Let's plot the unemployment rate in the US from 1929 to 2022 with recessions defined by the NBER.
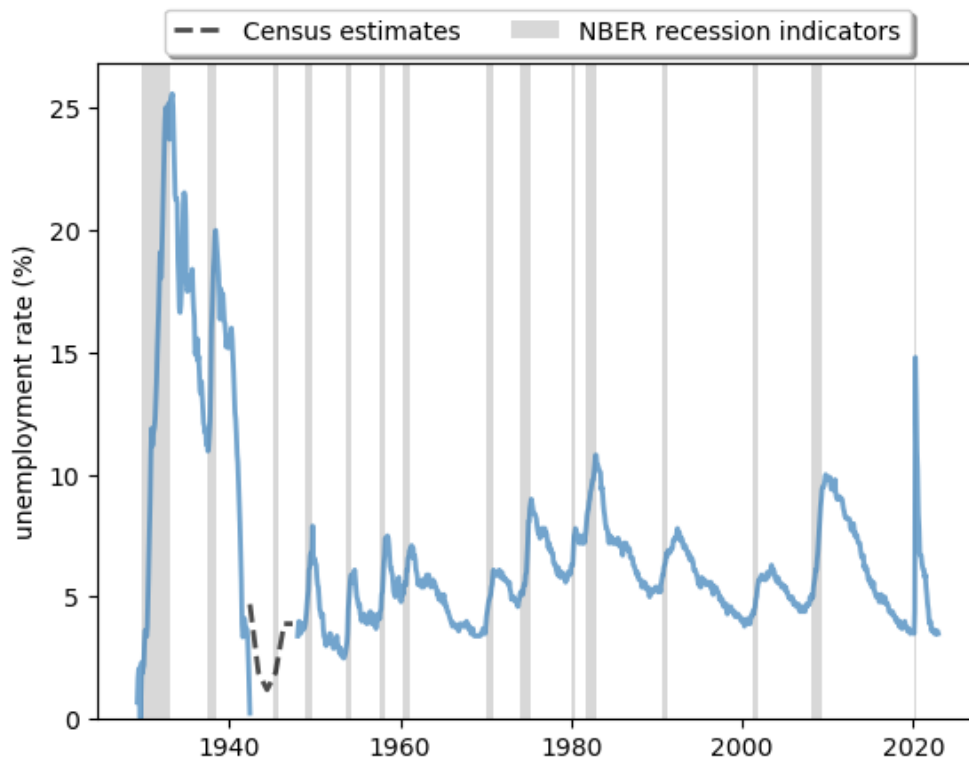

Fig. 3.6: Long-run unemployment rate, US (%)

The plot shows that

- expansions and contractions of the labor market have been highly correlated with recessions.

- cycles are, in general, asymmetric: sharp rises in unemployment are followed by slow recoveries.

It also shows us how unique labor market conditions were in the US during the post-pandemic recovery.

The labor market recovered at an unprecedented rate after the shock in 2020-2021.

3.5 Synchronization

In our *previous discussion*, we found that developed economies have had relatively synchronized periods of recession.

At the same time, this synchronization did not appear in Argentina until the 2000s.

Let's examine this trend further.

With slight modifications, we can use our previous function to draw a plot that includes multiple countries.

Here we compare the GDP growth rate of developed economies and developing economies.

We use the United Kingdom, United States, Germany, and Japan as examples of developed economies.
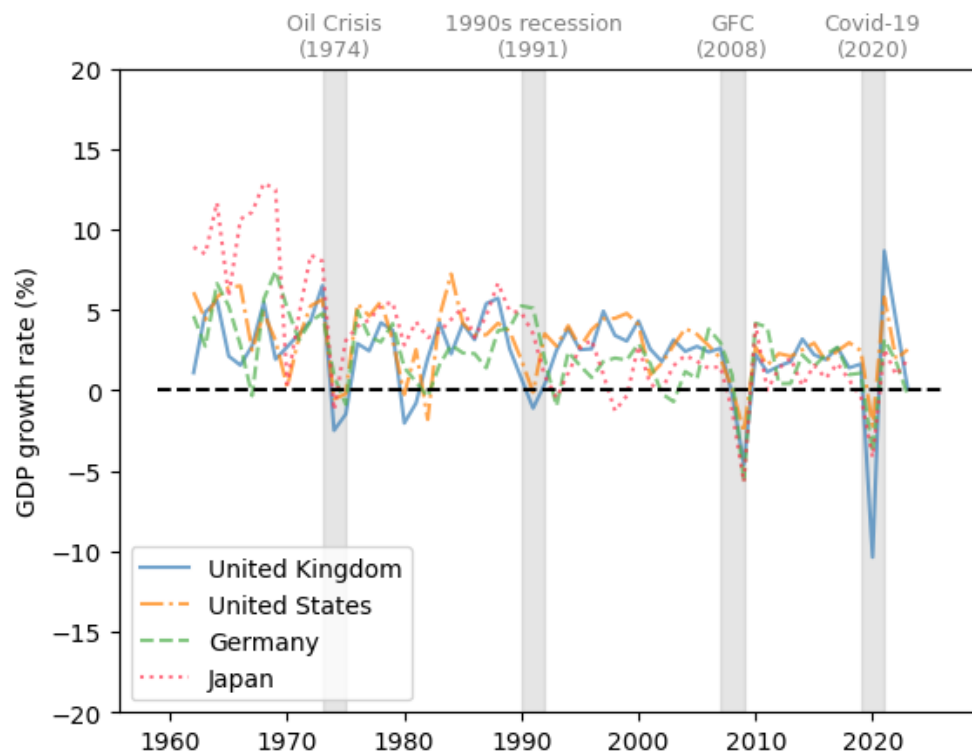

Fig. 3.7: Developed economies (GDP growth rate %)

We choose Brazil, China, Argentina, and Mexico as representative developing economies.

The comparison of GDP growth rates above suggests that business cycles are becoming more synchronized in 21st-century recessions.

However, emerging and less developed economies often experience more volatile changes throughout the economic cycles.
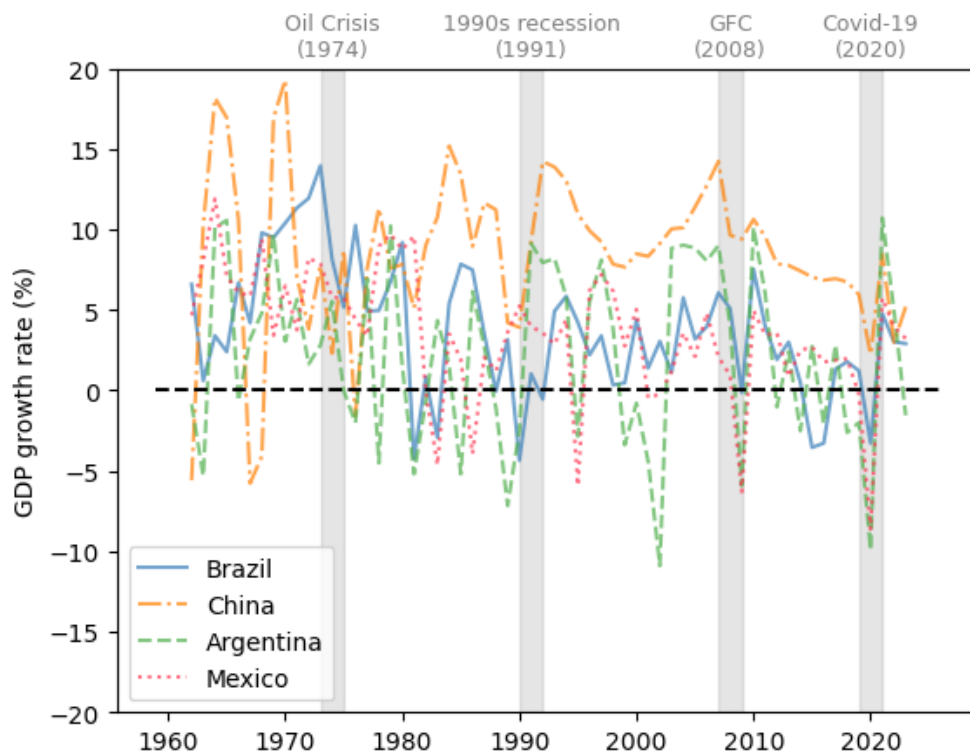
Fig. 3.8: Developing economies (GDP growth rate %)

Despite the synchronization in GDP growth, the experience of individual countries during the recession often differs.

We use the unemployment rate and the recovery of labor market conditions as another example.

Here we compare the unemployment rate of the United States, the United Kingdom, Japan, and France.

We see that France, with its strong labor unions, typically experiences relatively slow labor market recoveries after negative shocks.

We also notice that Japan has a history of very low and stable unemployment rates.

3.6 Leading indicators and correlated factors

Examining leading indicators and correlated factors helps policymakers to understand the causes and results of business cycles.

We will discuss potential leading indicators and correlated factors from three perspectives: consumption, production, and credit level.
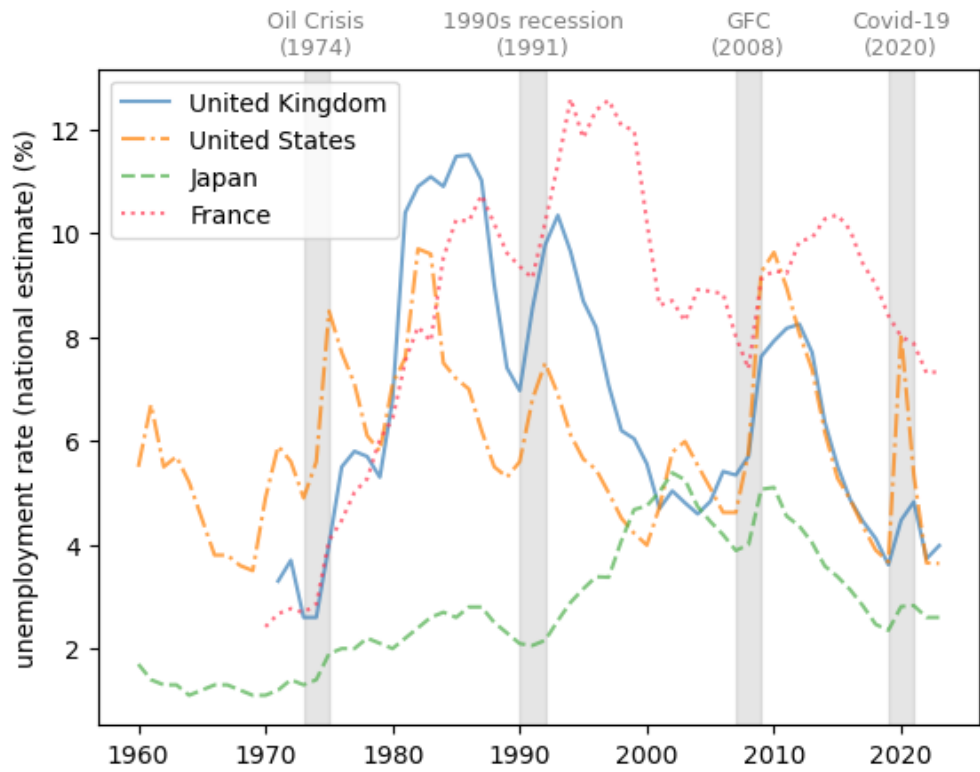
Fig. 3.9: Developed economies (unemployment rate %)

3.6.1 Consumption

Consumption depends on consumers'confidence towards their income and the overall performance of the economy in the future.

One widely cited indicator for consumer confidence is the consumer sentiment index published by the University of Michigan.

Here we plot the University of Michigan Consumer Sentiment Index and year-on-year core consumer price index (CPI) change from 1978-2022 in the US.
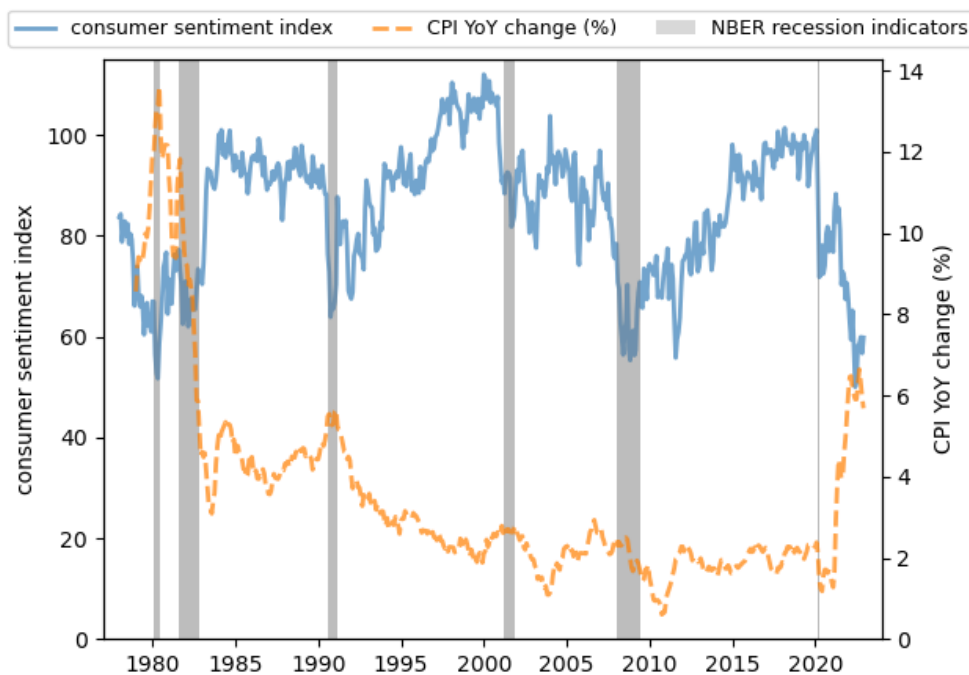

Fig. 3.10: Consumer sentiment index and YoY CPI change, US

We see that

- consumer sentiment often remains high during expansions and drops before recessions.

- there is a clear negative correlation between consumer sentiment and the CPI.

When the price of consumer commodities rises, consumer confidence diminishes.

This trend is more significant during stagflation.

3.6.2 Production

Real industrial output is highly correlated with recessions in the economy.

However, it is not a leading indicator, as the peak of contraction in production is delayed relative to consumer confidence and inflation.

We plot the real industrial output change from the previous year from 1919 to 2022 in the US to show this trend.
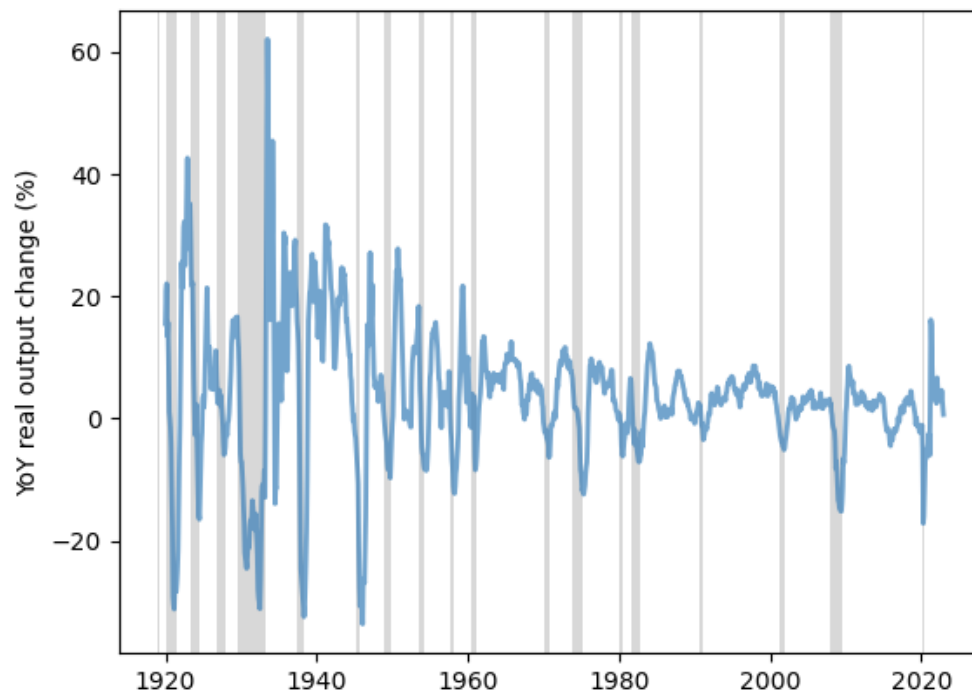
Fig. 3.11: YoY real output change, US (%)

We observe the delayed contraction in the plot across recessions.

3.6.3 Credit level

Credit contractions often occur during recessions, as lenders become more cautious and borrowers become more hesitant to take on additional debt.

This is due to factors such as a decrease in overall economic activity and gloomy expectations for the future.

One example is domestic credit to the private sector by banks in the UK.

The following graph shows the domestic credit to the private sector as a percentage of GDP by banks from 1970 to 2022 in the UK.

Fig. 3.12: Domestic credit to private sector by banks (% of GDP)

Note that the credit rises during economic expansions and stagnates or even contracts after recessions.

PRICE LEVEL HISTORIES

This lecture offers some historical evidence about fluctuations in levels of aggregate price indexes.

Let's start by installing the necessary Python packages.

The `xlrd` package is used by `pandas` to perform operations on Excel files.

```
!pip install xlrd
```
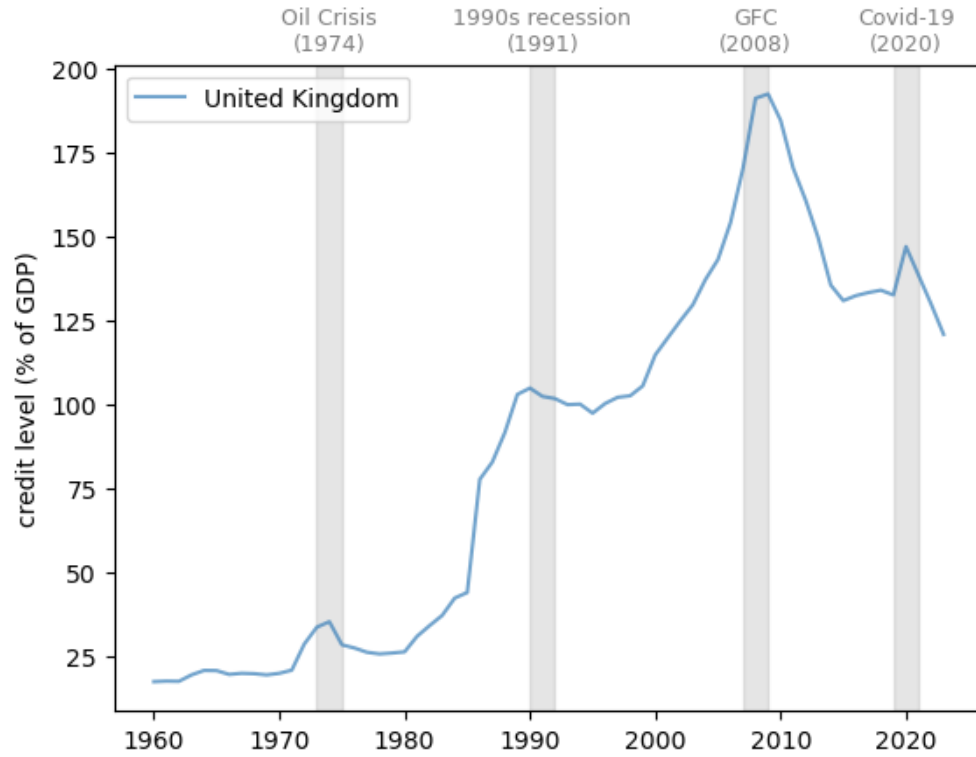
We can then import the Python modules we will use.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
```

The rate of growth of the price level is called **inflation** in the popular press and in discussions among central bankers and treasury officials.

The price level is measured in units of domestic currency per units of a representative bundle of consumption goods.

Thus, in the US, the price level at t is measured in dollars (month t or year t) per unit of the consumption bundle.

Until the early 20th century, in many western economies, price levels fluctuated from year to year but didn't have much of a trend.

Often the price levels ended a century near where they started.

Things were different in the 20th century, as we shall see in this lecture.

A widely believed explanation of this big difference is that countries'abandoning gold and silver standards in the early twentieth century.

Tip: This lecture sets the stage for some subsequent lectures about a theory that macro economists use to think about determinants of the price level, namely, *A Monetarist Theory of Price Levels* and *Monetarist Theory of Price Levels with*

4.1 Four centuries of price levels

We begin by displaying data that originally appeared on page 35 of [Sargent and Velde, 2002] that show price levels for four "hard currency" countries from 1600 to 1914.

- France

- Spain (Castile)

- United Kingdom

- United States

In the present context, the phrase "hard currency" means that the countries were on a commodity-money standard: money consisted of gold and silver coins that circulated at values largely determined by the weights of their gold and silver contents.

Note: Under a gold or silver standard, some money also consisted of "warehouse certificates" that represented paper claims on gold or silver coins. Bank notes issued by the government or private banks can be viewed as examples of such "warehouse certificates".

Let us bring the data into pandas from a spreadsheet that is hosted on github.

```python
# Import data and clean up the index
data_url = "https://github.com/QuantEcon/lecture-python-intro/raw/main/lectures/
 ↪datasets/longprices.xls"
df_fig5 = pd.read_excel(data_url,
                        sheet_name='all',
                        header=2,
                        index_col=0).iloc[1:]
df_fig5.index = df_fig5.index.astype(int)
```

We first plot price levels over the period 1600-1914.

During most years in this time interval, the countries were on a gold or silver standard.

```python
df_fig5_befe1914 = df_fig5[df_fig5.index <= 1914]

# Create plot
cols = ['UK', 'US', 'France', 'Castile']
```

(continues on next page)

```python
fig, ax = plt.subplots(figsize=(10,6))

for col in cols:
    ax.plot(df_fig5_befe1914.index,
            df_fig5_befe1914[col], label=col, lw=2)

ax.legend()
ax.set_ylabel('Index  1913 = 100')
ax.set_xlabel('Year')
ax.set_xlim(xmin=1600)
plt.tight_layout()
plt.show()
```
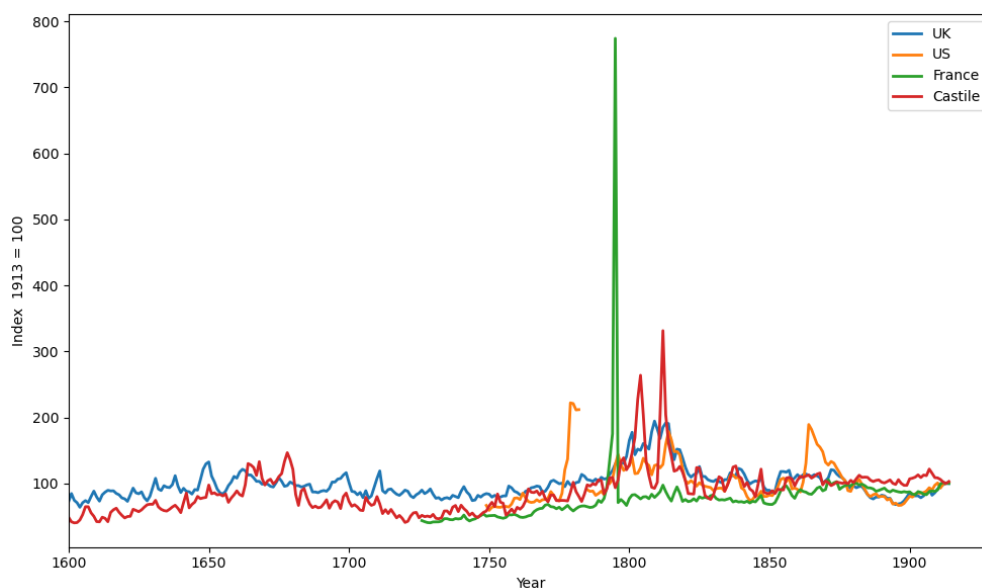

Fig. 4.1: Long run time series of the price level

We say "most years" because there were temporary lapses from the gold or silver standard.

By staring at Fig. 4.1 carefully, you might be able to guess when these temporary lapses occurred, because they were also times during which price levels temporarily rose markedly:

- 1791-1797 in France (French Revolution)

- 1776-1790 in the US (War for Independence from Great Britain)

- 1861-1865 in the US (Civil War)

During these episodes, the gold/silver standard was temporarily abandoned when a government printed paper money to pay for war expenditures.

Note: This quantecon lecture *Inflation During French Revolution* describes circumstances leading up to and during the big inflation that occurred during the French Revolution.

Despite these temporary lapses, a striking thing about the figure is that price levels were roughly constant over three centuries.

In the early century, two other features of this data attracted the attention of Irving Fisher of Yale University and John Maynard Keynes of Cambridge University.

- Despite being anchored to the same average level over long time spans, there were considerable year-to-year variations in price levels

- While using valuable gold and silver as coins succeeded in anchoring the price level by limiting the supply of money, it cost real resources.

- a country paid a high "opportunity cost" for using gold and silver coins as money –that gold and silver could instead have been made into valuable jewelry and other durable goods.

Keynes and Fisher proposed what they claimed would be a more efficient way to achieve a price level that

- would be at least as firmly anchored as achieved under a gold or silver standard, and

- would also exhibit less year-to-year short-term fluctuations.

They said that central bank could achieve price level stability by

- issuing **limited supplies** of paper currency

- refusing to print money to finance government expenditures

This logic prompted John Maynard Keynes to call a commodity standard a "barbarous relic."

A paper currency or "fiat money" system disposes of all reserves behind a currency.

But adhering to a gold or silver standard had provided an automatic mechanism for limiting the supply of money, thereby anchoring the price level.

To anchor the price level, a pure paper or fiat money system replaces that automatic mechanism with a central bank with the authority and determination to limit the supply of money (and to deter counterfeiters!)

Now let's see what happened to the price level in the four countries after 1914, when one after another of them left the gold/silver standard by showing the complete graph that originally appeared on page 35 of [Sargent and Velde, 2002].

Fig. 4.2 shows the logarithm of price levels over four "hard currency" countries from 1600 to 2000.

Note: Although we didn't have to use logarithms in our earlier graphs that had stopped in 1914, we now choose to use logarithms because we want to fit observations after 1914 in the same graph as the earlier observations.

After the outbreak of the Great War in 1914, the four countries left the gold standard and in so doing acquired the ability to print money to finance government expenditures.

```python
fig, ax = plt.subplots(dpi=200)

for col in cols:
    ax.plot(df_fig5.index, df_fig5[col], lw=2)
    ax.text(x=df_fig5.index[-1]+2,
            y=df_fig5[col].iloc[-1], s=col)

ax.set_yscale('log')
ax.set_ylabel('Logs of price levels (Index  1913 = 100)')
ax.set_ylim([10, 1e6])
ax.set_xlabel('year')
ax.set_xlim(xmin=1600)
plt.tight_layout()
plt.show()
```
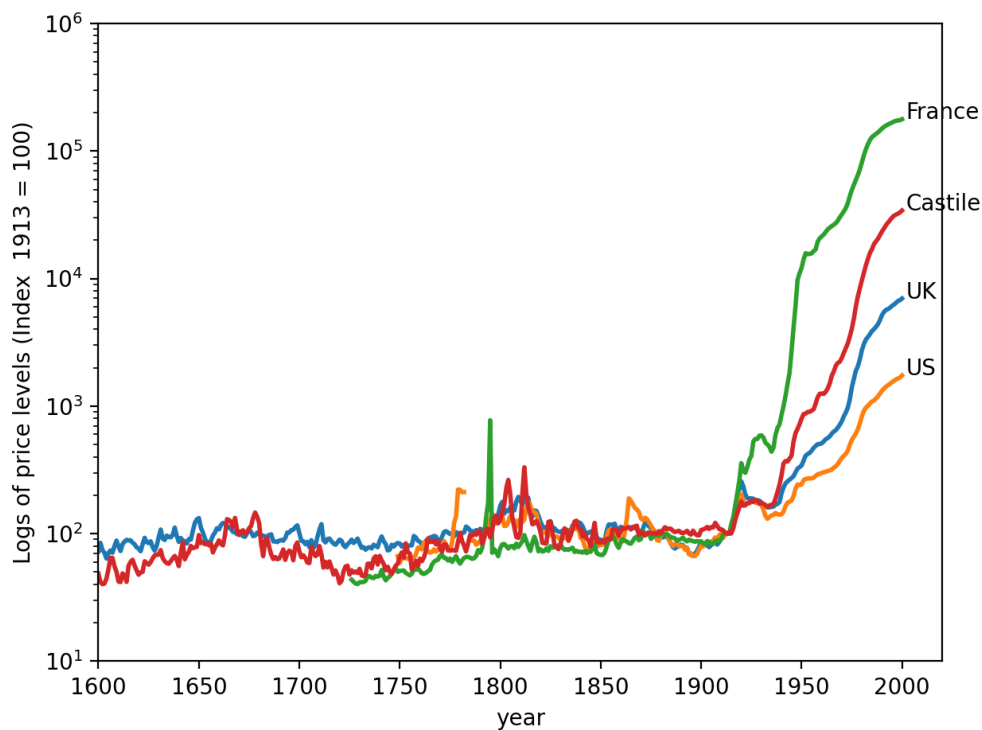

Fig. 4.2: Long run time series of the price level (log)

4.1. Four centuries of price levels 61

Fig. 4.2 shows that paper-money-printing central banks didn't do as well as the gold and standard silver standard in anchoring price levels.

That would probably have surprised or disappointed Irving Fisher and John Maynard Keynes.

Actually, earlier economists and statesmen knew about the possibility of fiat money systems long before Keynes and Fisher advocated them in the early 20th century.

Proponents of a commodity money system did not trust governments and central banks properly to manage a fiat money system.

They were willing to pay the resource costs associated with setting up and maintaining a commodity money system.

In light of the high and persistent inflation that many countries experienced after they abandoned commodity monies in the twentieth century, we hesitate to criticize advocates of a gold or silver standard for their preference to stay on the pre-1914 gold/silver standard.

The breadth and lengths of the inflationary experiences of the twentieth century under paper money fiat standards are historically unprecedented.

4.2 Four big inflations

In the wake of World War I, which ended in November 1918, monetary and fiscal authorities struggled to achieve price level stability without being on a gold or silver standard.

We present four graphs from "The Ends of Four Big Inflations" from chapter 3 of [Sargent, 2013].

The graphs depict logarithms of price levels during the early post World War I years for four countries:

- Figure 3.1, Retail prices Austria, 1921-1924 (page 42)

- Figure 3.2, Wholesale prices Hungary, 1921-1924 (page 43)

- Figure 3.3, Wholesale prices, Poland, 1921-1924 (page 44)

- Figure 3.4, Wholesale prices, Germany, 1919-1924 (page 45)

We have added logarithms of the exchange rates vis-à-vis the US dollar to each of the four graphs from chapter 3 of [Sargent, 2013].

Data underlying our graphs appear in tables in an appendix to chapter 3 of [Sargent, 2013]. We have transcribed all of these data into a spreadsheet `chapter_3.xlsx` that we read into pandas.

In the code cell below we clean the data and build a `pandas.dataframe`.

Now we write plotting functions `pe_plot` and `pr_plot` that will build figures that show the price level, exchange rates, and inflation rates, for each country of interest.

We prepare the data for each country

```python
# Import data
data_url = "https://github.com/QuantEcon/lecture-python-intro/raw/main/lectures/
 ↪datasets/chapter_3.xlsx"
xls = pd.ExcelFile(data_url)

# Select relevant sheets
sheet_index = [(2, 3, 4),
               (9, 10),
               (14, 15, 16),
               (21, 18, 19)]

# Remove redundant rows
remove_row = [(-2, -2, -2),
              (-7, -10),
              (-6, -4, -3),
              (-19, -3, -6)]

# Unpack and combine series for each country
df_list = []

for i in range(4):

    indices, rows = sheet_index[i], remove_row[i]

    # Apply process_entry on the selected sheet
    sheet_list = [
        pd.read_excel(xls, 'Table3.' + str(ind),
            header=1).iloc[:row].map(process_entry)
        for ind, row in zip(indices, rows)]

    sheet_list = [process_df(df) for df in sheet_list]
    df_list.append(pd.concat(sheet_list, axis=1))

df_aus, df_hun, df_pol, df_deu = df_list
```

Now let's construct graphs for our four countries.

For each country, we'll plot two graphs.

The first graph plots logarithms of

- price levels

- exchange rates vis-à-vis US dollars

For each country, the scale on the right side of a graph will pertain to the price level while the scale on the left side of a graph will pertain to the exchange rate.

For each country, the second graph plots a centered three-month moving average of the inflation rate defined as $\frac{p_{t-1}+p_t+p_{t+1}}{3}$.

4.2.1 Austria

The sources of our data are:

- Table 3.3, retail price level $\exp p$

- Table 3.4, exchange rate with US

```
p_seq = df_aus['Retail price index, 52 commodities']
e_seq = df_aus['Exchange Rate']

lab = ['Retail price index',
       'Austrian Krones (Crowns) per US cent']

# Create plot
fig, ax = plt.subplots(dpi=200)
_ = pe_plot(p_seq, e_seq, df_aus.index, lab, ax)

plt.show()
```

```
# Plot moving average
fig, ax = plt.subplots(dpi=200)
_ = pr_plot(p_seq, df_aus.index, ax)

plt.show()
```

Staring at Fig. 4.3 and Fig. 4.4 conveys the following impressions to the authors of this lecture at QuantEcon.

- an episode of "hyperinflation" with rapidly rising log price level and very high monthly inflation rates

- a sudden stop of the hyperinflation as indicated by the abrupt flattening of the log price level and a marked permanent drop in the three-month average of inflation

- a US dollar exchange rate that shadows the price level.

We'll see similar patterns in the next three episodes that we'll study now.
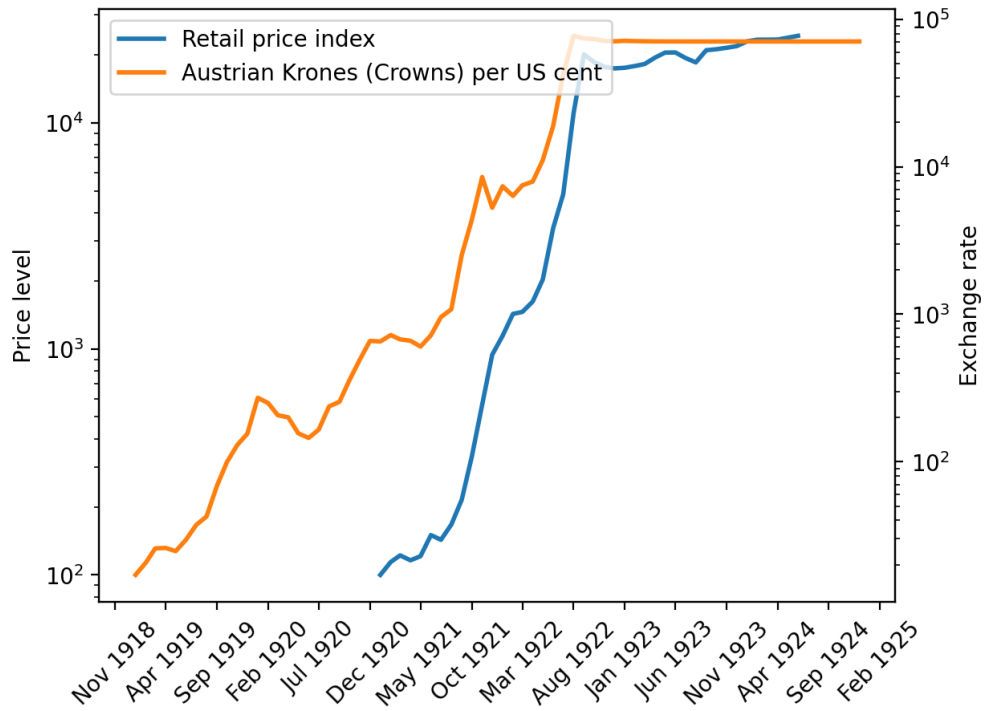
Fig. 4.3: Price index and exchange rate (Austria)
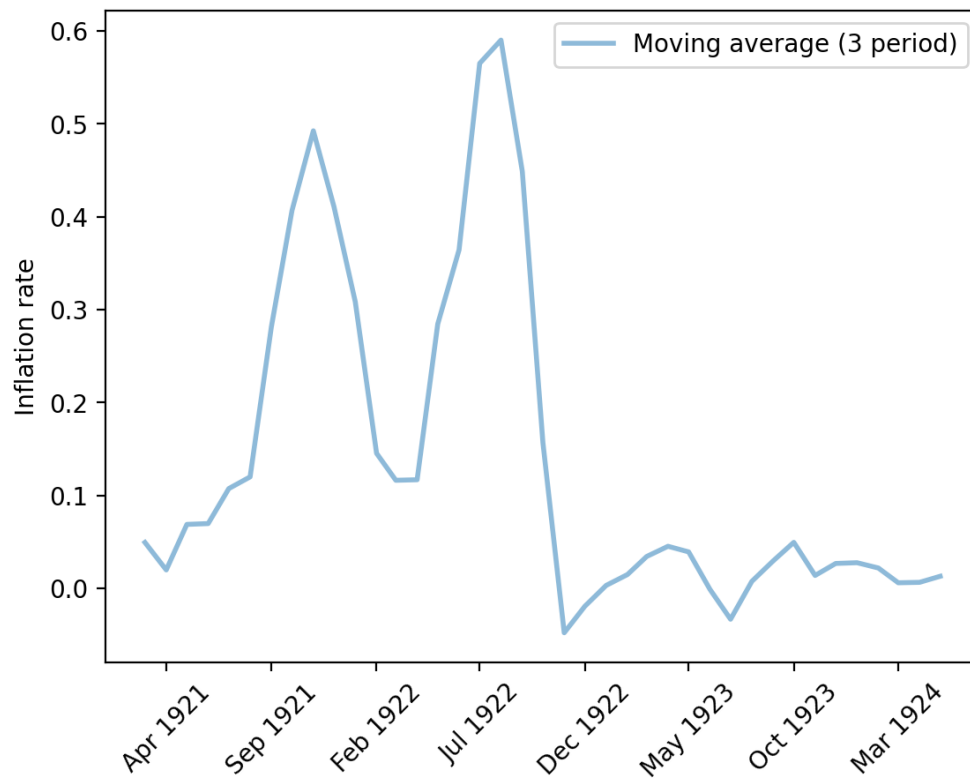
Fig. 4.4: Monthly inflation rate (Austria)

4.2.2 Hungary

The source of our data for Hungary is:

- Table 3.10, price level exp p and exchange rate

```python
p_seq = df_hun['Hungarian index of prices']
e_seq = 1 / df_hun['Cents per crown in New York']

lab = ['Hungarian index of prices',
       'Hungarian Koronas (Crowns) per US cent']

# Create plot
fig, ax = plt.subplots(dpi=200)
_ = pe_plot(p_seq, e_seq, df_hun.index, lab, ax)

plt.show()
```
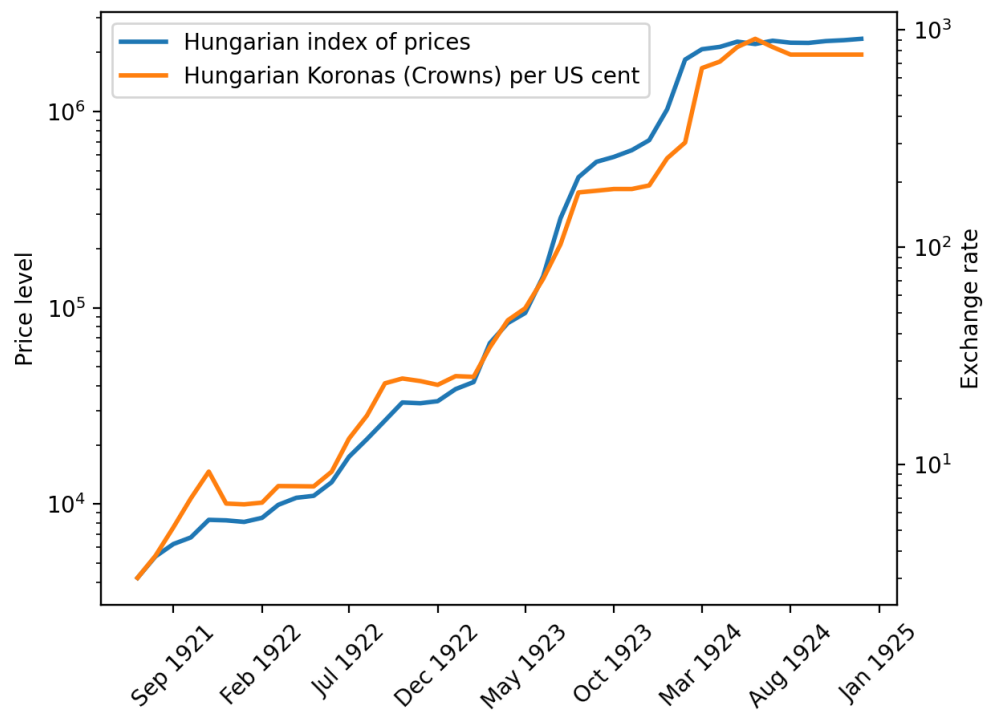

Fig. 4.5: Price index and exchange rate (Hungary)

```python
# Plot moving average
fig, ax = plt.subplots(dpi=200)
```

```
_ = pr_plot(p_seq, df_hun.index, ax)

plt.show()
```
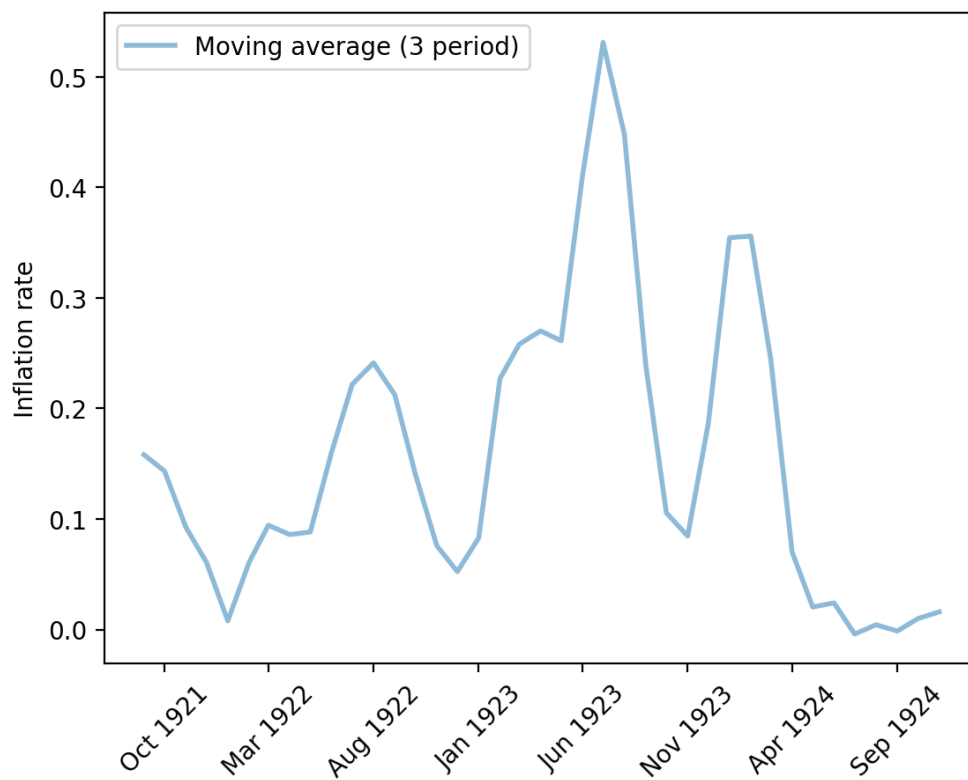

Fig. 4.6: Monthly inflation rate (Hungary)

4.2.3 Poland

The sources of our data for Poland are:

- Table 3.15, price level exp p

- Table 3.15, exchange rate

Note: To construct the price level series from the data in the spreadsheet, we instructed Pandas to follow the same procedures implemented in chapter 3 of [Sargent, 2013]. We spliced together three series - Wholesale price index, Wholesale Price Index: On paper currency basis, and Wholesale Price Index: On zloty basis. We adjusted the sequence based on the price level ratio at the last period of the available previous series and glued them to construct a single series.

We dropped the exchange rate after June 1924, when the zloty was adopted. We did this because we don't have the price measured in zloty. We used the old currency in June to compute the exchange rate adjustment.

```python
# Splice three price series in different units
p_seq1 = df_pol['Wholesale price index'].copy()
p_seq2 = df_pol['Wholesale Price Index: '
                'On paper currency basis'].copy()
p_seq3 = df_pol['Wholesale Price Index: '
                'On zloty basis'].copy()

# Non-nan part
mask_1 = p_seq1[~p_seq1.isna()].index[-1]
mask_2 = p_seq2[~p_seq2.isna()].index[-2]

adj_ratio12 = (p_seq1[mask_1] / p_seq2[mask_1])
adj_ratio23 = (p_seq2[mask_2] / p_seq3[mask_2])

# Glue three series
p_seq = pd.concat([p_seq1[:mask_1],
                   adj_ratio12 * p_seq2[mask_1:mask_2],
                   adj_ratio23 * p_seq3[mask_2:]])
p_seq = p_seq[~p_seq.index.duplicated(keep='first')]

# Exchange rate
e_seq = 1/df_pol['Cents per Polish mark (zloty after May 1924)']
e_seq[e_seq.index > '05-01-1924'] = np.nan
```

```python
lab = ['Wholesale price index',
       'Polish marks per US cent']

# Create plot
fig, ax = plt.subplots(dpi=200)
ax1 = pe_plot(p_seq, e_seq, df_pol.index, lab, ax)

plt.show()
```
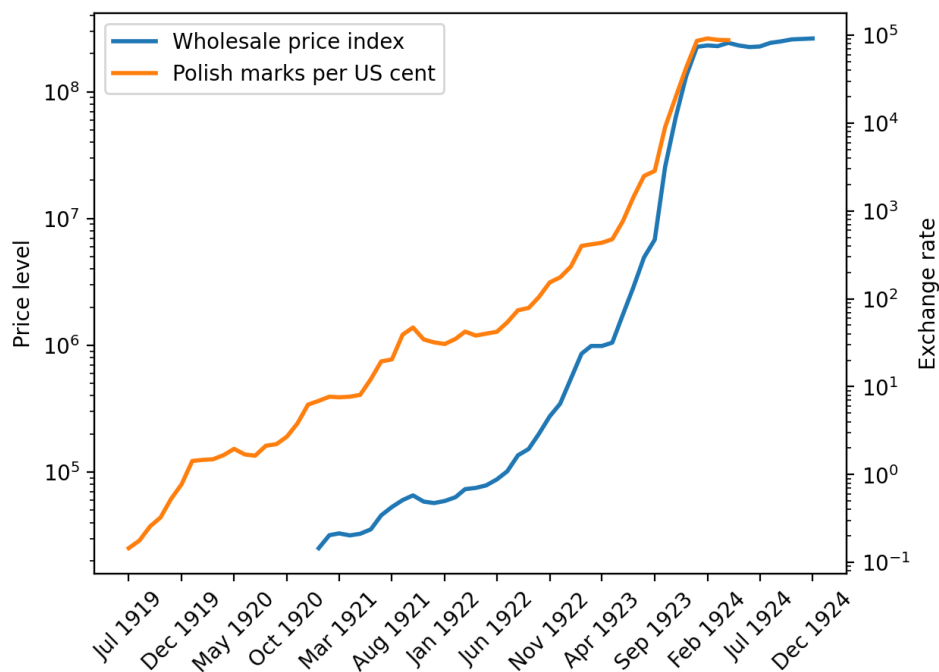
```
# Plot moving average
fig, ax = plt.subplots(dpi=200)
_ = pr_plot(p_seq, df_pol.index, ax)


plt.show()
```

4.2.4 Germany

The sources of our data for Germany are the following tables from chapter 3 of [Sargent, 2013]:

- Table 3.18, wholesale price level $\exp p$

- Table 3.19, exchange rate

```
p_seq = df_deu['Price index (on basis of marks before July 1924,'
                '  reichsmarks after)'].copy()
e_seq = 1/df_deu['Cents per mark']


lab = ['Price index',
        'Marks per US cent']


# Create plot
fig, ax = plt.subplots(dpi=200)
```

(continues on next page)
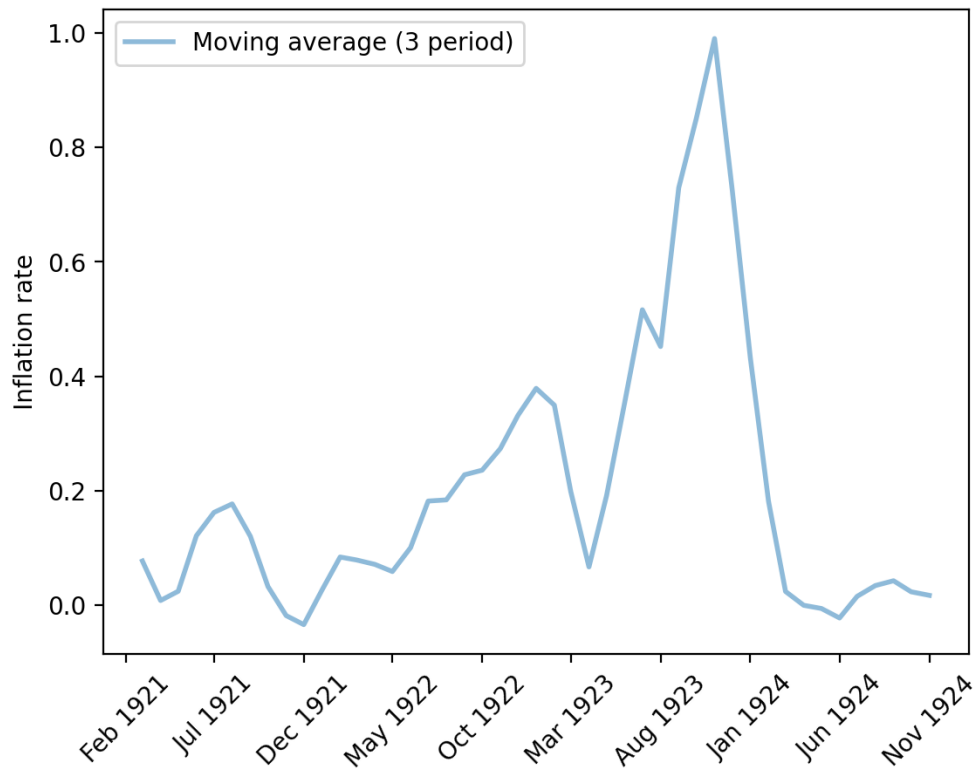
Fig. 4.7: Monthly inflation rate (Poland)

```
ax1 = pe_plot(p_seq, e_seq, df_deu.index, lab, ax)

plt.show()
```
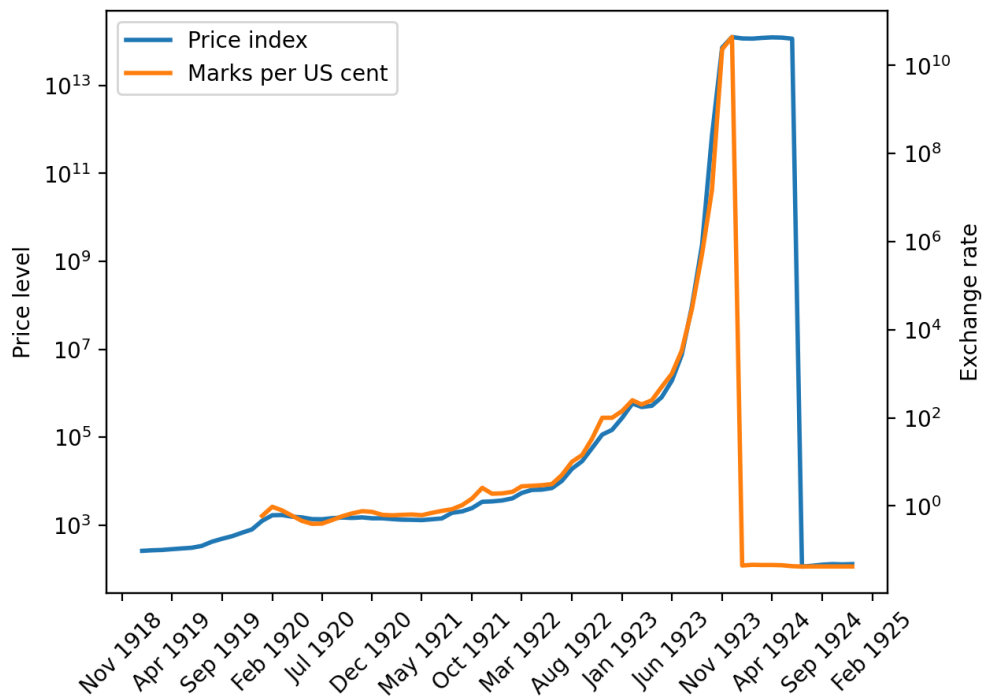

Fig. 4.8: Price index and exchange rate (Germany)

```
p_seq = df_deu['Price index (on basis of marks before July 1924,'
               '  reichsmarks after)'].copy()
e_seq = 1/df_deu['Cents per mark'].copy()

# Adjust the price level/exchange rate after the currency reform
p_seq[p_seq.index > '06-01-1924'] = p_seq[p_seq.index
                                        > '06-01-1924'] * 1e12
e_seq[e_seq.index > '12-01-1923'] = e_seq[e_seq.index
                                        > '12-01-1923'] * 1e12


lab = ['Price index (marks or converted to marks)',
       'Marks per US cent(or reichsmark converted to mark)']

# Create plot
fig, ax = plt.subplots(dpi=200)
```

```
ax1 = pe_plot(p_seq, e_seq, df_deu.index, lab, ax)

plt.show()
```


Fig. 4.9: Price index (adjusted) and exchange rate (Germany)

```
# Plot moving average
fig, ax = plt.subplots(dpi=200)
_ = pr_plot(p_seq, df_deu.index, ax)

plt.show()
```
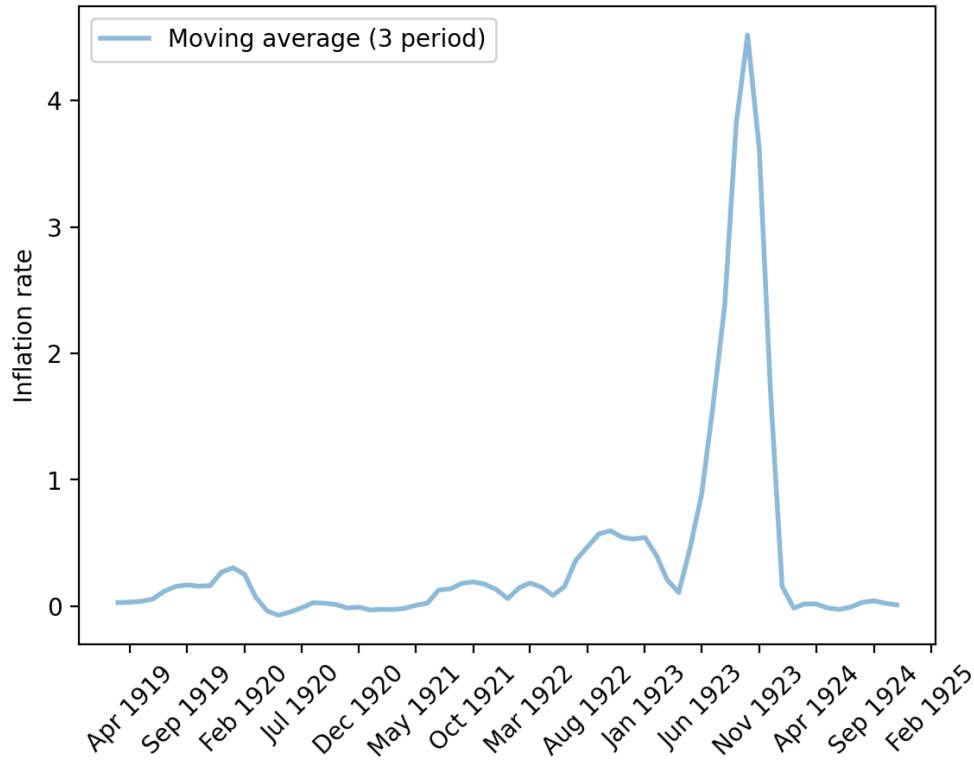
Fig. 4.10: Monthly inflation rate (Germany)

4.3 Starting and stopping big inflations

It is striking how *quickly* (log) price levels in Austria, Hungary, Poland, and Germany leveled off after rising so quickly.

These "sudden stops" are also revealed by the permanent drops in three-month moving averages of inflation for the four countries plotted above.

In addition, the US dollar exchange rates for each of the four countries shadowed their price levels.

Note: This pattern is an instance of a force featured in the purchasing power parity theory of exchange rates.

Each of these big inflations seemed to have "stopped on a dime".

Chapter 3 of [Sargent and Velde, 2002] offers an explanation for this remarkable pattern.

In a nutshell, here is the explanation offered there.

After World War I, the United States was on a gold standard.

The US government stood ready to convert a dollar into a specified amount of gold on demand.

Immediately after World War I, Hungary, Austria, Poland, and Germany were not on the gold standard.

Their currencies were "fiat" or "unbacked", meaning that they were not backed by credible government promises to convert them into gold or silver coins on demand.

The governments printed new paper notes to pay for goods and services.

Note: Technically the notes were "backed" mainly by treasury bills. But people could not expect that those treasury bills would be paid off by levying taxes, but instead by printing more notes or treasury bills.

This was done on such a scale that it led to a depreciation of the currencies of spectacular proportions.

In the end, the German mark stabilized at 1 trillion (10^{12}) paper marks to the prewar gold mark, the Polish mark at 1.8 million paper marks to the gold zloty, the Austrian crown at 14,400 paper crowns to the prewar Austro-Hungarian crown, and the Hungarian krone at 14,500 paper crowns to the prewar Austro-Hungarian crown.

Chapter 3 of [Sargent and Velde, 2002] described deliberate changes in policy that Hungary, Austria, Poland, and Germany made to end their hyperinflations.

Each government stopped printing money to pay for goods and services once again and made its currency convertible to the US dollar or the UK pound.

The story told in [Sargent and Velde, 2002] is grounded in a *monetarist theory of the price level* described in *A Monetarist Theory of Price Levels* and *Monetarist Theory of Price Levels with Adaptive Expectations*.

Those lectures discuss theories about what owners of those rapidly depreciating currencies were thinking and how their beliefs shaped responses of inflation to government monetary and fiscal policies.

FIVE

INFLATION DURING FRENCH REVOLUTION

5.1 Overview

This lecture describes some of the monetary and fiscal features of the French Revolution (1789-1799) described by [Sargent and Velde, 1995].

To finance public expenditures and service its debts, the French government embarked on policy experiments.

The authors of these experiments had in mind theories about how government monetary and fiscal policies affected economic outcomes.

Some of those theories about monetary and fiscal policies still interest us today.

- a **tax-smoothing** model like Robert Barro's [Barro, 1979]

  - this normative (i.e., prescriptive model) advises a government to finance temporary war-time surges in expenditures mostly by issuing government debt, raising taxes by just enough to service the additional debt issued during the wary; then, after the war, to roll over whatever debt the government had accumulated during the war; and to increase taxes after the war permanently by just enough to finance interest payments on that post-war government debt

- **unpleasant monetarist arithmetic** like that described in this quanteon lecture *Some Unpleasant Monetarist Arithmetic*

  - mathematics involving compound interest governed French government debt dynamics in the decades preceding 1789; according to leading historians, that arithmetic set the stage for the French Revolution

- a *real bills* theory of the effects of government open market operations in which the government *backs* new issues of paper money with government holdings of valuable real property or financial assets that holders of money can purchase from the government in exchange for their money.

  - The Revolutionaries learned about this theory from Adam Smith's 1776 book The Wealth of Nations [Smith, 2010] and other contemporary sources

  - It shaped how the Revolutionaries issued a paper money called **assignats** from 1789 to 1791

- a classical **gold** or **silver standard**

  - Napoleon Bonaparte became head of the French government in 1799. He used this theory to guide his monetary and fiscal policies

- a classical **inflation-tax** theory of inflation in which Philip Cagan's ([Cagan, 1956]) demand for money studied in this lecture *A Monetarist Theory of Price Levels* is a key component

  - This theory helps explain French price level and money supply data from 1794 to 1797

- a **legal restrictions** or **financial repression** theory of the demand for real balances

  - The Twelve Members comprising the Committee of Public Safety who adminstered the Terror from June 1793 to July 1794 used this theory to shape their monetary policy

We use matplotlib to replicate several of the graphs with which [Sargent and Velde, 1995] portrayed outcomes of these experiments

5.2 Data Sources

This lecture uses data from three spreadsheets assembled by [Sargent and Velde, 1995]:

- datasets/fig_3.xlsx

- datasets/dette.xlsx

- datasets/assignat.xlsx

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
plt.rcParams.update({'font.size': 12})


base_url = 'https://github.com/QuantEcon/lecture-python-intro/raw/'\
           + 'main/lectures/datasets/'


fig_3_url = f'{base_url}fig_3.xlsx'
dette_url = f'{base_url}dette.xlsx'
assignat_url = f'{base_url}assignat.xlsx'
```

5.3 Government Expenditures and Taxes Collected

We'll start by using `matplotlib` to construct several graphs that will provide important historical context.

These graphs are versions of ones that appear in [Sargent and Velde, 1995].

These graphs show that during the 18th century

- government expenditures in France and Great Britain both surged during four big wars, and by comparable amounts

- In Britain, tax revenues were approximately equal to government expenditures during peace times, but were substantially less than government expenditures during wars

- In France, even in peace time, tax revenues were substantially less than government expenditures

```python
# Read the data from Excel file
data2 = pd.read_excel(dette_url,
        sheet_name='Militspe', usecols='M:X',
        skiprows=7, nrows=102, header=None)

# French military spending, 1685-1789, in 1726 livres
data4 = pd.read_excel(dette_url,
        sheet_name='Militspe', usecols='D',
        skiprows=3, nrows=105, header=None).squeeze()

years = range(1685, 1790)

plt.figure()
plt.plot(years, data4, '*-', linewidth=0.8)

plt.plot(range(1689, 1791), data2.iloc[:, 4], linewidth=0.8)

plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)
plt.gca().tick_params(labelsize=12)
plt.xlim([1689, 1790])
plt.xlabel('*: France')
plt.ylabel('Millions of livres')
plt.ylim([0, 475])

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

During the 18th century, Britain and France fought four large wars.
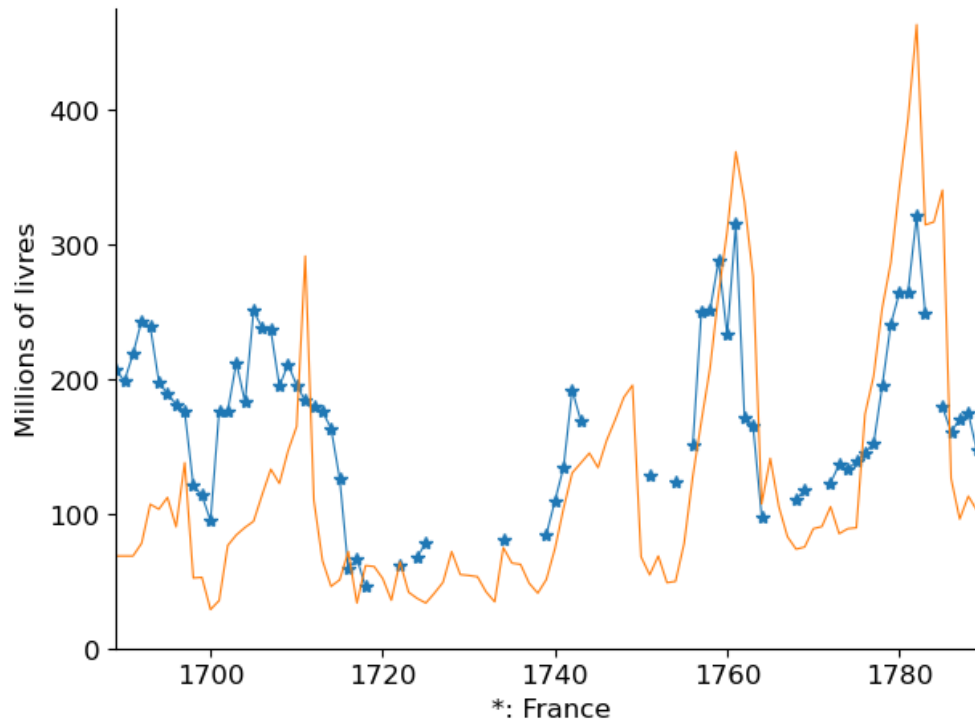
Fig. 5.1: Military Spending in Britain and France

Britain won the first three wars and lost the fourth.

Each of those wars produced surges in both countries'government expenditures that each country somehow had to finance.

Figure Fig. 5.1 shows surges in military expenditures in France (in blue) and Great Britain. during those four wars.

A remarkable aspect of figure Fig. 5.1 is that despite having a population less than half of France's, Britain was able to finance military expenses of about the same amounts as France's.

This testifies to Britain's having created state institutions that could sustain high tax collections, government spending , and government borrowing. See [North and Weingast, 1989].

```python
# Read the data from Excel file
data2 = pd.read_excel(dette_url, sheet_name='Militspe', usecols='M:X',
                      skiprows=7, nrows=102, header=None)

# Plot the data
plt.figure()
plt.plot(range(1689, 1791), data2.iloc[:, 5], linewidth=0.8)
plt.plot(range(1689, 1791), data2.iloc[:, 11], linewidth=0.8, color='red')
plt.plot(range(1689, 1791), data2.iloc[:, 9], linewidth=0.8, color='orange')
plt.plot(range(1689, 1791), data2.iloc[:, 8], 'o-',
         markerfacecolor='none', linewidth=0.8, color='purple')
```

```
# Customize the plot
plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)
plt.gca().tick_params(labelsize=12)
plt.xlim([1689, 1790])
plt.ylabel('millions of pounds', fontsize=12)

# Add text annotations
plt.text(1765, 1.5, 'civil', fontsize=10)
plt.text(1760, 4.2, 'civil plus debt service', fontsize=10)
plt.text(1708, 15.5, 'total govt spending', fontsize=10)
plt.text(1759, 7.3, 'revenues', fontsize=10)

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


Fig. 5.2: Government Expenditures and Tax Revenues in Britain

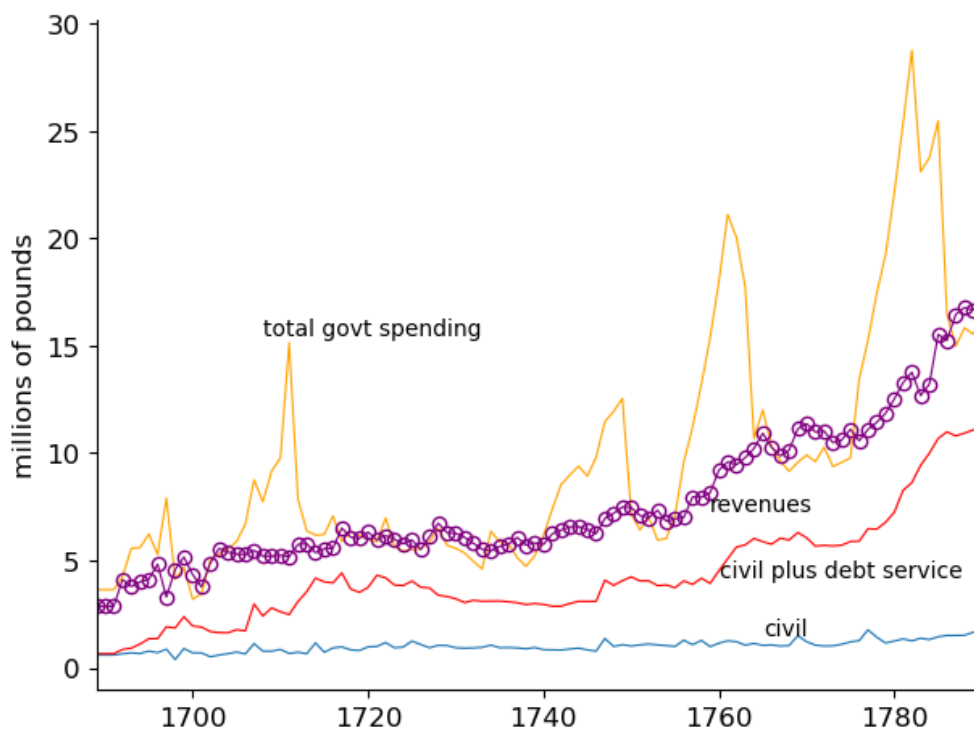Figures Fig. 5.2 and Fig. 5.4 summarize British and French government fiscal policies during the century before the start of the French Revolution in 1789.

Before 1789, progressive forces in France admired how Britain had financed its government expenditures and wanted to redesign French fiscal arrangements to make them more like Britain's.

Figure Fig. 5.2 shows government expenditures and how it was distributed among expenditures for

- civil (non-military) activities

- debt service, i.e., interest payments

- military expenditures (the yellow line minus the red line)

Figure Fig. 5.2 also plots total government revenues from tax collections (the purple circled line)

Notice the surges in total government expenditures associated with surges in military expenditures in these four wars

- Wars against France's King Louis XIV early in the 18th century

- The War of the Austrian Succession in the 1740s

- The French and Indian War in the 1750's and 1760s

- The American War for Independence from 1775 to 1783

Figure Fig. 5.2 indicates that

- during times of peace, government expenditures approximately equal taxes and debt service payments neither grow nor decline over time

- during times of wars, government expenditures exceed tax revenues

  - the government finances the deficit of revenues relative to expenditures by issuing debt

- after a war is over, the government's tax revenues exceed its non-interest expenditures by just enough to service the debt that the government issued to finance earlier deficits

  - thus, after a war, the government does *not* raise taxes by enough to pay off its debt

  - instead, it just rolls over whatever debt it inherits, raising taxes by just enough to service the interest payments on that debt

Eighteenth-century British fiscal policy portrayed Figure Fig. 5.2 thus looks very much like a text-book example of a *tax-smoothing* model like Robert Barro's [Barro, 1979].

A striking feature of the graph is what we'll label a *law of gravity* between tax collections and government expenditures.

- levels of government expenditures at taxes attract each other

- while they can temporarily differ –as they do during wars –they come back together when peace returns

Next we'll plot data on debt service costs as fractions of government revenues in Great Britain and France during the 18th century.

```python
# Read the data from the Excel file
data1 = pd.read_excel(dette_url, sheet_name='Debt',
            usecols='R:S', skiprows=5, nrows=99, header=None)
data1a = pd.read_excel(dette_url, sheet_name='Debt',
            usecols='P', skiprows=89, nrows=15, header=None)

# Plot the data
plt.figure()
plt.plot(range(1690, 1789), 100 * data1.iloc[:, 1], linewidth=0.8)

date = np.arange(1690, 1789)
index = (date < 1774) & (data1.iloc[:, 0] > 0)
plt.plot(date[index], 100 * data1[index].iloc[:, 0],
        '*:', color='r', linewidth=0.8)

# Plot the additional data
plt.plot(range(1774, 1789), 100 * data1a, '*:', color='orange')

# Note about the data
# The French data before 1720 don't match up with the published version
# Set the plot properties
plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)
plt.gca().set_facecolor('white')
plt.gca().set_xlim([1688, 1788])
plt.ylabel('% of Taxes')

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

Figure Fig. 5.3 shows that interest payments on government debt (i.e., so-called "debt service") were high fractions of government tax revenues in both Great Britain and France.

Fig. 5.2 showed us that in peace times Britain managed to balance its budget despite those large interest costs.

But as we'll see in our next graph, on the eve of the French Revolution in 1788, the fiscal *law of gravity* that worked so well in Britain did not working very well in France.

```python
# Read the data from the Excel file
data1 = pd.read_excel(fig_3_url, sheet_name='Sheet1',
        usecols='C:F', skiprows=5, nrows=30, header=None)

data1.replace(0, np.nan, inplace=True)
```
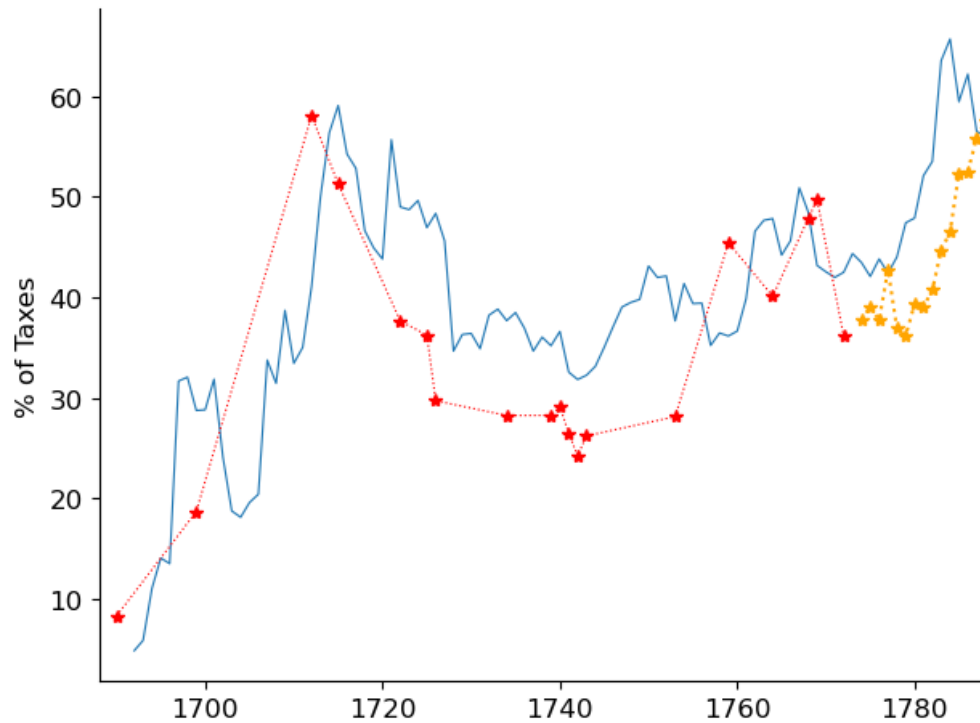
Fig. 5.3: Ratio of debt service to taxes, Britain and France

```
# Plot the data
plt.figure()

plt.plot(range(1759, 1789, 1), data1.iloc[:, 0], '-x', linewidth=0.8)
plt.plot(range(1759, 1789, 1), data1.iloc[:, 1], '--*', linewidth=0.8)
plt.plot(range(1759, 1789, 1), data1.iloc[:, 2],
         '-o', linewidth=0.8, markerfacecolor='none')
plt.plot(range(1759, 1789, 1), data1.iloc[:, 3], '-*', linewidth=0.8)

plt.text(1775, 610, 'total spending', fontsize=10)
plt.text(1773, 325, 'military', fontsize=10)
plt.text(1773, 220, 'civil plus debt service', fontsize=10)
plt.text(1773, 80, 'debt service', fontsize=10)
plt.text(1785, 500, 'revenues', fontsize=10)

plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)
plt.ylim([0, 700])
plt.ylabel('millions of livres')

plt.tight_layout()
```

```
plt.show()
```
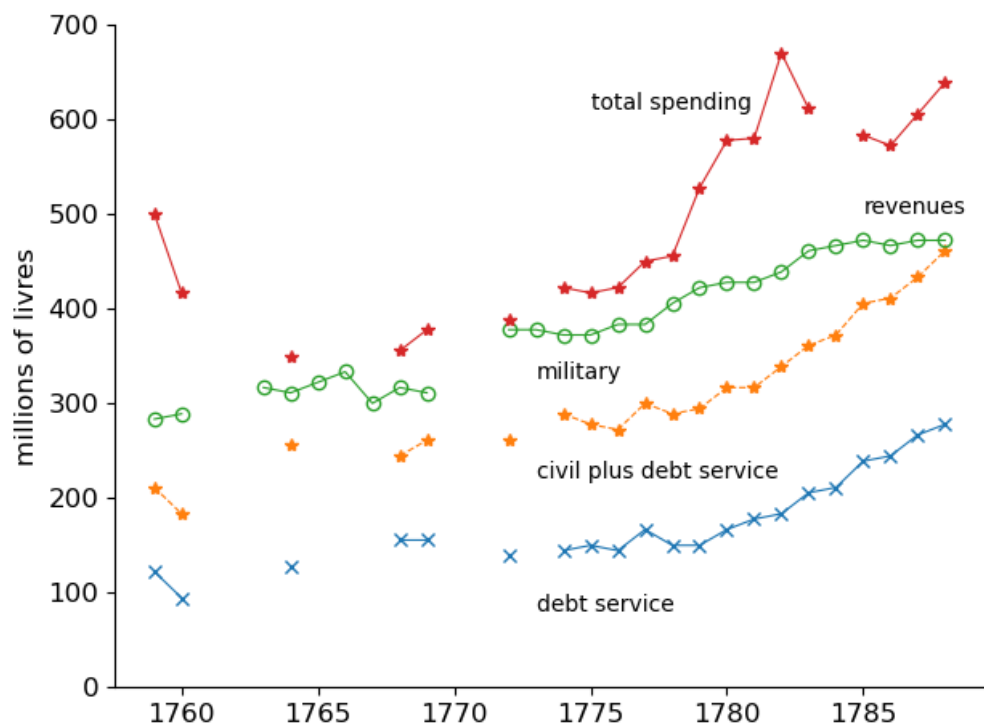

Fig. 5.4: Government Spending and Tax Revenues in France

Fig. 5.4 shows that on the eve of the French Revolution in 1788, government expenditures exceeded tax revenues.

Especially during and after France's expenditures to help the Americans in their War of Independence from Great Britain, growing government debt service (i.e., interest payments) contributed to this situation.

This was partly a consequence of the unfolding of the debt dynamics that underlies the Unpleasant Arithmetic discussed in this quantecon lecture *Some Unpleasant Monetarist Arithmetic*.

[Sargent and Velde, 1995] describe how the Ancient Regime that until 1788 had governed France had stable institutional features that made it difficult for the government to balance its budget.

Powerful contending interests had prevented from the government from closing the gap between its total expenditures and its tax revenues by either

- raising taxes, or

- lowering government's non-debt service (i.e., non-interest) expenditures, or

- lowering debt service (i.e., interest) costs by rescheduling, i.e., defaulting on some debts

Precedents and prevailing French arrangements had empowered three constituencies to block adjustments to components of the government budget constraint that they cared especially about

- tax payers

- beneficiaries of government expenditures

- government creditors (i.e., owners of government bonds)

When the French government had confronted a similar situation around 1720 after King Louis XIV's Wars had left it with a debt crisis, it had sacrificed the interests of
government creditors, i.e., by defaulting enough of its debt to bring reduce interest payments down enough to balance the budget.

Somehow, in 1789, creditors of the French government were more powerful than they had been in 1720.

Therefore, King Louis XVI convened the Estates General together to ask them to redesign the French constitution in a way that would lower government expenditures or increase taxes, thereby allowing him to balance the budget while also honoring his promises to creditors of the French government.

The King called the Estates General together in an effort to promote the reforms that would would bring sustained budget balance.

[Sargent and Velde, 1995] describe how the French Revolutionaries set out to accomplish that.

5.4 Nationalization, Privatization, Debt Reduction

In 1789, the Revolutionaries quickly reorganized the Estates General into a National Assembly.

A first piece of business was to address the fiscal crisis, the situation that had motivated the King to convene the Estates General.

The Revolutionaries were not socialists or communists.

To the contrary, they respected private property and knew state-of-the-art economics.

They knew that to honor government debts, they would have to raise new revenues or reduce expenditures.

A coincidence was that the Catholic Church owned vast income-producing properties.

Indeed, the capitalized value of those income streams put estimates of the value of church lands at about the same amount as the entire French government debt.

This coincidence fostered a three step plan for servicing the French government debt

- nationalize the church lands –i.e., sequester or confiscate it without paying for it

- sell the church lands

- use the proceeds from those sales to service or even retire French government debt

The monetary theory underlying this plan had been set out by Adam Smith in his analysis of what he called *real bills* in his 1776 book **The Wealth of Nations** [Smith, 2010], which many of the revolutionaries had read.

Adam Smith defined a *real bill* as a paper money note that is backed by a claims on a real asset like productive capital or inventories.

The National Assembly put together an ingenious institutional arrangement to implement this plan.

In response to a motion by Catholic Bishop Talleyrand (an atheist), the National Assembly confiscated and nationalized Church lands.

The National Assembly intended to use earnings from Church lands to service its national debt.

To do this, it began to implement a "privatization plan" that would let it service its debt while not raising taxes.

Their plan involved issuing paper notes called "assignats" that entitled bearers to use them to purchase state lands.

These paper notes would be "as good as silver coins" in the sense that both were acceptable means of payment in exchange for those (formerly) church lands.

Finance Minister Necker and the Constituents of the National Assembly thus planned to solve the privatization problem *and* the debt problem simultaneously by creating a new currency.

They devised a scheme to raise revenues by auctioning the confiscated lands, thereby withdrawing paper notes issued on the security of the lands sold by the government.

This "tax-backed money" scheme propelled the National Assembly into the domains of then modern monetary theories.

Records of debates show how members of the Assembly marshaled theory and evidence to assess the likely effects of their innovation.

- Members of the National Assembly quoted David Hume and Adam Smith

- They cited John Law's System of 1720 and the American experiences with paper money fifteen years earlier as examples of how paper money schemes can go awry

- Knowing pitfalls, they set out to avoid them

They succeeded for two or three years.

But after that, France entered a big War that disrupted the plan in ways that completely altered the character of France's paper money. [Sargent and Velde, 1995] describe what happened.

5.5 Remaking the tax code and tax administration

In 1789 the French Revolutionaries formed a National Assembly and set out to remake French fiscal policy.

They wanted to honor government debts –interests of French government creditors were well represented in the National Assembly.

But they set out to remake the French tax code and the administrative machinery for collecting taxes.

- they abolished many taxes

- they abolished the Ancient Regimes scheme for *tax farming*

    – tax farming meant that the government had privatized tax collection by hiring private citizens –so-called tax farmers to collect taxes, while retaining a fraction of them as payment for their services

    – the great chemist Lavoisier was also a tax farmer, one of the reasons that the Committee for Public Safety sent him to the guillotine in 1794

As a consequence of these tax reforms, government tax revenues declined

The next figure shows this

```python
# Read data from Excel file
data5 = pd.read_excel(dette_url, sheet_name='Debt', usecols='K',
                      skiprows=41, nrows=120, header=None)

# Plot the data
plt.figure()
plt.plot(range(1726, 1846), data5.iloc[:, 0], linewidth=0.8)

plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)
plt.gca().set_facecolor('white')
plt.gca().tick_params(labelsize=12)
plt.xlim([1726, 1845])
plt.ylabel('1726 = 1', fontsize=12)

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

According to Fig. 5.5, tax revenues per capita did not rise to their pre 1789 levels until after 1815, when Napoleon Bonaparte was exiled to St Helena and King Louis XVIII was restored to the French Crown.

- from 1799 to 1814, Napoleon Bonaparte had other sources of revenues –booty and reparations from provinces and

Fig. 5.5: Index of real per capital revenues, France

nations that he defeated in war

- from 1789 to 1799, the French Revolutionaries turned to another source to raise resources to pay for government purchases of goods and services and to service French government debt.

And as the next figure shows, government expenditures exceeded tax revenues by substantial amounts during the period form 1789 to 1799.

```python
# Read data from Excel file
data11 = pd.read_excel(assignat_url, sheet_name='Budgets',
        usecols='J:K', skiprows=22, nrows=52, header=None)

# Prepare the x-axis data
x_data = np.concatenate([
    np.arange(1791, 1794 + 8/12, 1/12),
    np.arange(1794 + 9/12, 1795 + 3/12, 1/12)
])

# Remove NaN values from the data
data11_clean = data11.dropna()

# Plot the data
```

(continues on next page)

```python
plt.figure()
h = plt.plot(x_data, data11_clean.values[:, 0], linewidth=0.8)
h = plt.plot(x_data, data11_clean.values[:, 1], '--', linewidth=0.8)

# Set plot properties
plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)
plt.gca().set_facecolor('white')
plt.gca().tick_params(axis='both', which='major', labelsize=12)
plt.xlim([1791, 1795 + 3/12])
plt.xticks(np.arange(1791, 1796))
plt.yticks(np.arange(0, 201, 20))

# Set the y-axis label
plt.ylabel('millions of livres', fontsize=12)

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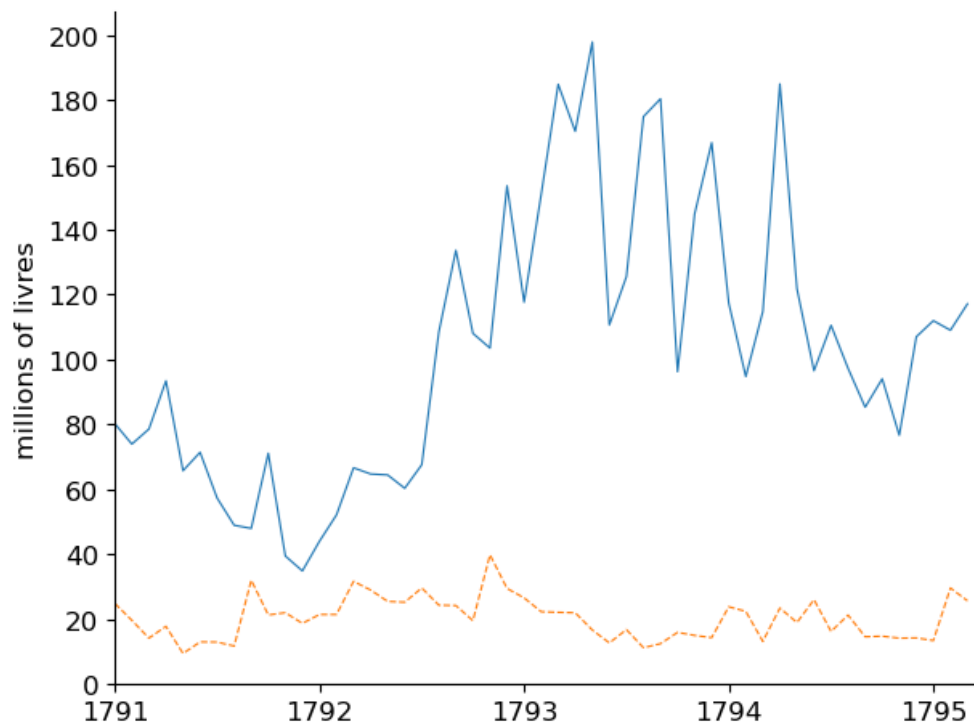

Fig. 5.6: Spending (blue) and Revenues (orange), (real values)

To cover the discrepancies between government expenditures and tax revenues revealed in Fig. 5.6, the French revolu-

tionaries printed paper money and spent it.

The next figure shows that by printing money, they were able to finance substantial purchases of goods and services, including military goods and soldiers'pay.

```python
# Read data from Excel file
data12 = pd.read_excel(assignat_url, sheet_name='seignor',
        usecols='F', skiprows=6, nrows=75, header=None).squeeze()


# Create a figure and plot the data
plt.figure()
plt.plot(pd.date_range(start='1790', periods=len(data12), freq='M'),
        data12, linewidth=0.8)

plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)

plt.axhline(y=472.42/12, color='r', linestyle=':')
plt.xticks(ticks=pd.date_range(start='1790',
            end='1796', freq='AS'), labels=range(1790, 1797))
plt.xlim(pd.Timestamp('1791'),
        pd.Timestamp('1796-02') + pd.DateOffset(months=2))
plt.ylabel('millions of livres', fontsize=12)
plt.text(pd.Timestamp('1793-11'), 39.5, 'revenues in 1788',
        verticalalignment='top', fontsize=12)

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

Fig. 5.7 compares the revenues raised by printing money from 1789 to 1796 with tax revenues that the Ancient Regime had raised in 1788.

Measured in goods, revenues raised at time t by printing new money equal

$$\frac{M_{t+1} - M_t}{p_t}$$

where

- M_t is the stock of paper money at time t measured in livres

- p_t is the price level at time t measured in units of goods per livre at time t

- $M_{t+1} - M_t$ is the amount of new money printed at time t

Notice the 1793-1794 surge in revenues raised by printing money.
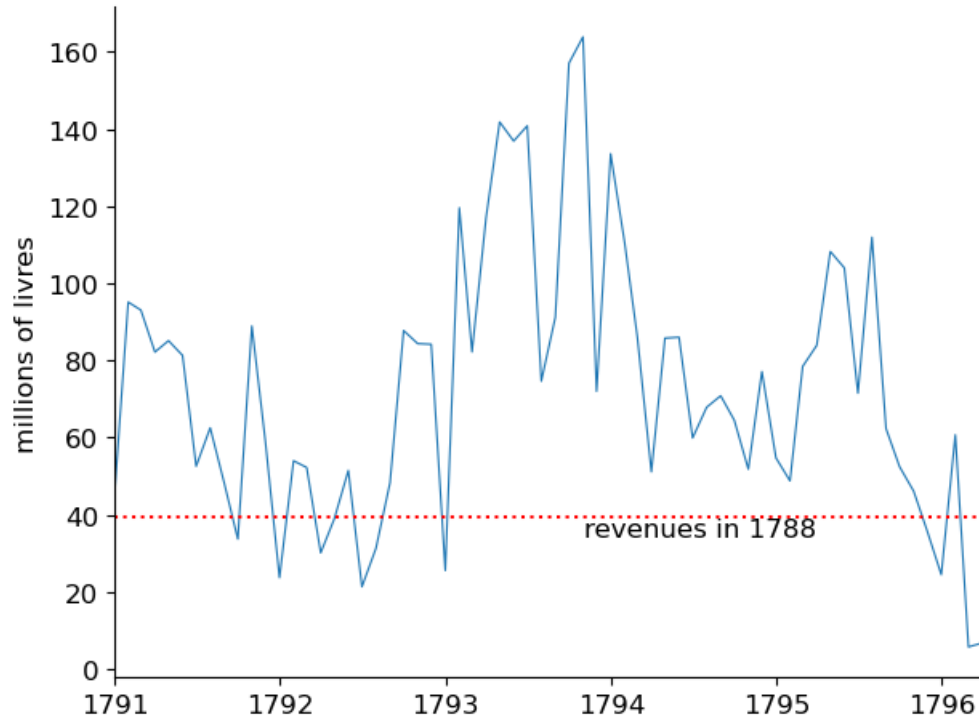
Fig. 5.7: Revenues raised by printing paper money notes

- This reflects extraordinary measures that the Committee for Public Safety adopted to force citizens to accept paper money, or else.

Also note the abrupt fall off in revenues raised by 1797 and the absence of further observations after 1797.

- This reflects the end of using the printing press to raise revenues.

What French paper money entitled its holders to changed over time in interesting ways.

These led to outcomes that vary over time and that illustrate the playing out in practice of theories that guided the Revolutionaries' monetary policy decisions.

The next figure shows the price level in France during the time that the Revolutionaries used paper money to finance parts of their expenditures.

Note that we use a log scale because the price level rose so much.

```python
# Read the data from Excel file
data7 = pd.read_excel(assignat_url, sheet_name='Data',
        usecols='P:Q', skiprows=4, nrows=80, header=None)
data7a = pd.read_excel(assignat_url, sheet_name='Data',
        usecols='L', skiprows=4, nrows=80, header=None)
# Create the figure and plot
plt.figure()
```

```python
x = np.arange(1789 + 10/12, 1796 + 5/12, 1/12)
h, = plt.plot(x, 1. / data7.iloc[:, 0], linestyle='--')
h, = plt.plot(x, 1. / data7.iloc[:, 1], color='r')


# Set properties of the plot
plt.gca().tick_params(labelsize=12)
plt.yscale('log')
plt.xlim([1789 + 10/12, 1796 + 5/12])
plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)


# Add vertical lines
plt.axvline(x=1793 + 6.5/12, linestyle='-', linewidth=0.8, color='orange')
plt.axvline(x=1794 + 6.5/12, linestyle='-', linewidth=0.8, color='purple')


# Add text
plt.text(1793.75, 120, 'Terror', fontsize=12)
plt.text(1795, 2.8, 'price level', fontsize=12)
plt.text(1794.9, 40, 'gold', fontsize=12)



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

We have partioned Fig. 5.8 that shows the log of the price level and Fig. 5.9 below that plots real balances $\frac{M_t}{p_t}$ into three periods that correspond to different monetary experiments or *regimes*.

The first period ends in the late summer of 1793, and is characterized by growing real balances and moderate inflation.

The second period begins and ends with the Terror. It is marked by high real balances, around 2,500 million, and roughly stable prices. The fall of Robespierre in late July 1794 begins the third of our episodes, in which real balances decline and prices rise rapidly.

We interpret these three episodes in terms of distinct theories

- a *backing* or *real bills* theory (the classic text for this theory is Adam Smith [Smith, 2010])

- a legal restrictions theory ([Keynes, 1940], [Bryant and Wallace, 1984])

- a classical hyperinflation theory ([Cagan, 1956])

-

Note: According to the empirical definition of hyperinflation adopted by [Cagan, 1956], beginning in the month that

---
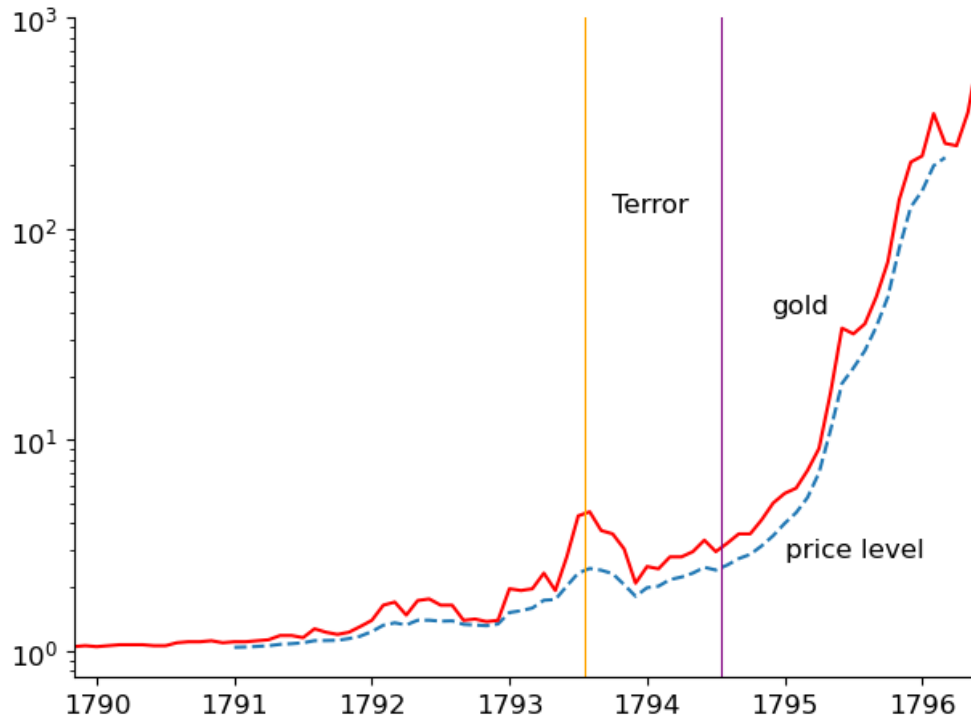
Fig. 5.8: Price Level and Price of Gold (log scale)

inflation exceeds 50 percent per month and ending in the month before inflation drops below 50 percent per month for at least a year, the *assignat* experienced a hyperinflation from May to December 1795.

We view these theories not as competitors but as alternative collections of "if-then" statements about government note issues, each of which finds its conditions more nearly met in one of these episodes than in the other two.

```python
# Read the data from Excel file
data7 = pd.read_excel(assignat_url, sheet_name='Data',
        usecols='P:Q', skiprows=4, nrows=80, header=None)
data7a = pd.read_excel(assignat_url, sheet_name='Data',
        usecols='L', skiprows=4, nrows=80, header=None)

# Create the figure and plot
plt.figure()
h = plt.plot(pd.date_range(start='1789-11-01', periods=len(data7), freq='M'),
             (data7a.values * [1, 1]) * data7.values, linewidth=1.)
plt.setp(h[1], linestyle='--', color='red')

plt.vlines([pd.Timestamp('1793-07-15'), pd.Timestamp('1793-07-15')],
           0, 3000, linewidth=0.8, color='orange')
plt.vlines([pd.Timestamp('1794-07-15'), pd.Timestamp('1794-07-15')],
```

```
            0, 3000, linewidth=0.8, color='purple')

plt.ylim([0, 3000])

# Set properties of the plot
plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)
plt.gca().set_facecolor('white')
plt.gca().tick_params(labelsize=12)
plt.xlim(pd.Timestamp('1789-11-01'), pd.Timestamp('1796-06-01'))
plt.ylabel('millions of livres', fontsize=12)

# Add text annotations
plt.text(pd.Timestamp('1793-09-01'), 200, 'Terror', fontsize=12)
plt.text(pd.Timestamp('1791-05-01'), 750, 'gold value', fontsize=12)
plt.text(pd.Timestamp('1794-10-01'), 2500, 'real value', fontsize=12)


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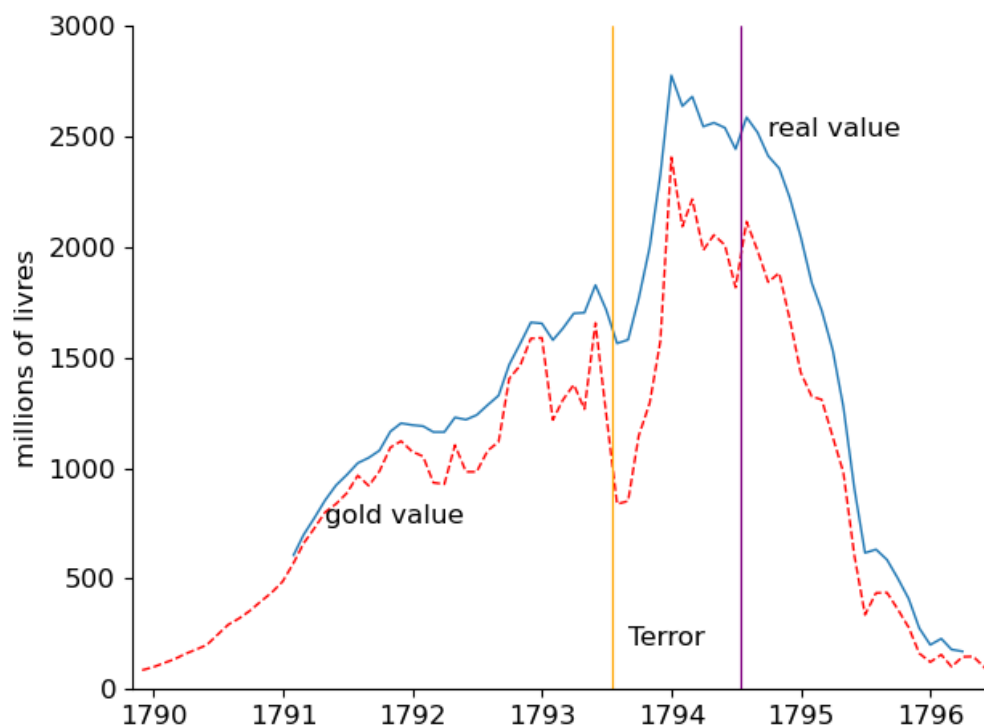

Fig. 5.9: Real balances of assignats (in gold and goods)

The three clouds of points in Figure Fig. 5.10 depict different real balance-inflation relationships.

Only the cloud for the third period has the inverse relationship familiar to us now from twentieth-century hyperinflations.

- subperiod 1: ("*real bills* period): January 1791 to July 1793

- subperiod 2: ("terror"): August 1793 - July 1794

- subperiod 3: ("classic Cagan hyperinflation"): August 1794 - March 1796

```python
def fit(x, y):

    b = np.cov(x, y)[0, 1] / np.var(x)
    a = y.mean() - b * x.mean()

    return a, b
```

```python
# Load data
caron = np.load('datasets/caron.npy')
nom_balances = np.load('datasets/nom_balances.npy')

infl = np.concatenate(([np.nan],
      -np.log(caron[1:63, 1] / caron[0:62, 1])))
bal = nom_balances[14:77, 1] * caron[:, 1] / 1000
```

```python
# Regress y on x for three periods
a1, b1 = fit(bal[1:31], infl[1:31])
a2, b2 = fit(bal[31:44], infl[31:44])
a3, b3 = fit(bal[44:63], infl[44:63])

# Regress x on y for three periods
a1_rev, b1_rev = fit(infl[1:31], bal[1:31])
a2_rev, b2_rev = fit(infl[31:44], bal[31:44])
a3_rev, b3_rev = fit(infl[44:63], bal[44:63])
```

```python
plt.figure()
plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)

# First subsample
plt.plot(bal[1:31], infl[1:31], 'o', markerfacecolor='none',
         color='blue', label='real bills period')
```

```
# Second subsample
plt.plot(bal[31:44], infl[31:44], '+', color='red', label='terror')

# Third subsample
plt.plot(bal[44:63], infl[44:63], '*',
        color='orange', label='classic Cagan hyperinflation')

plt.xlabel('real balances')
plt.ylabel('inflation')
plt.legend()

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


Fig. 5.10: Inflation and Real Balances

The three clouds of points in Fig. 5.10 evidently depict different real balance-inflation relationships.

Only the cloud for the third period has the inverse relationship familiar to us now from twentieth-century hyperinflations.

To bring this out, we'll use linear regressions to draw straight lines that compress the inflation-real balance relationship for our three sub-periods.

Before we do that, we'll drop some of the early observations during the terror period to obtain the following graph.

```python
# Regress y on x for three periods
a1, b1 = fit(bal[1:31], infl[1:31])
a2, b2 = fit(bal[31:44], infl[31:44])
a3, b3 = fit(bal[44:63], infl[44:63])

# Regress x on y for three periods
a1_rev, b1_rev = fit(infl[1:31], bal[1:31])
a2_rev, b2_rev = fit(infl[31:44], bal[31:44])
a3_rev, b3_rev = fit(infl[44:63], bal[44:63])
```

```python
plt.figure()
plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)

# First subsample
plt.plot(bal[1:31], infl[1:31], 'o', markerfacecolor='none', color='blue', label=
 ↪'real bills period')

# Second subsample
plt.plot(bal[34:44], infl[34:44], '+', color='red', label='terror')

# Third subsample
plt.plot(bal[44:63], infl[44:63], '*', color='orange', label='classic Cagan␣
 ↪hyperinflation')

plt.xlabel('real balances')
plt.ylabel('inflation')
plt.legend()

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

Now let's regress inflation on real balances during the *real bills* period and plot the regression line.

```python
plt.figure()
plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)

# First subsample
plt.plot(bal[1:31], infl[1:31], 'o', markerfacecolor='none',
```
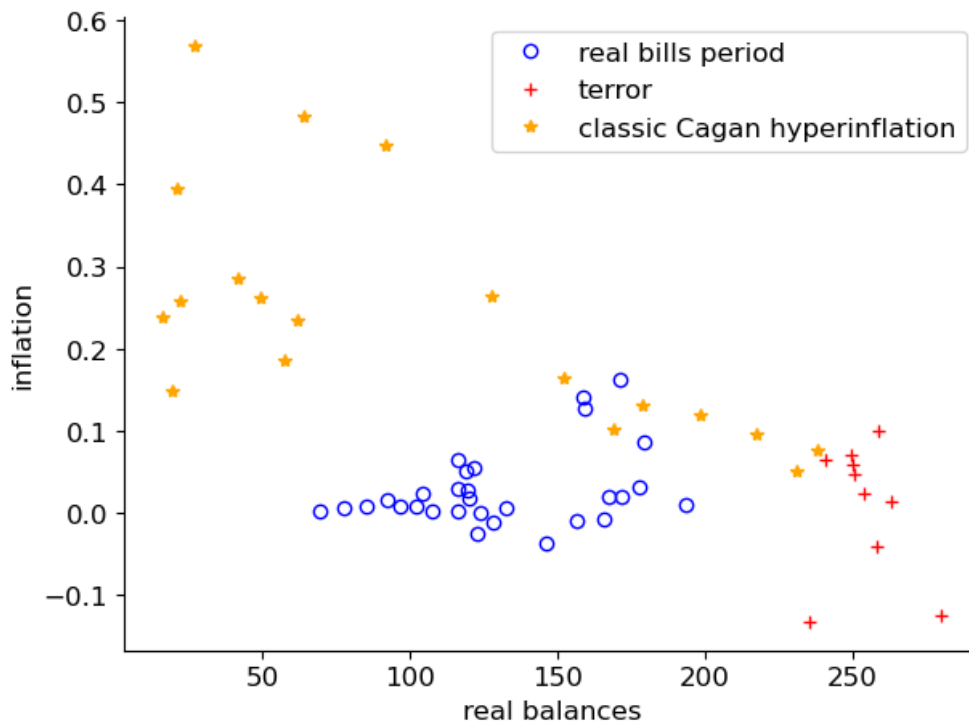
Fig. 5.11: Inflation and Real Balances

```
        color='blue', label='real bills period')
plt.plot(bal[1:31], a1 + bal[1:31] * b1, color='blue')


# Second subsample
plt.plot(bal[31:44], infl[31:44], '+', color='red', label='terror')


# Third subsample
plt.plot(bal[44:63], infl[44:63], '*',
        color='orange', label='classic Cagan hyperinflation')

plt.xlabel('real balances')
plt.ylabel('inflation')
plt.legend()

plt.tight_layout()
plt.show()
```
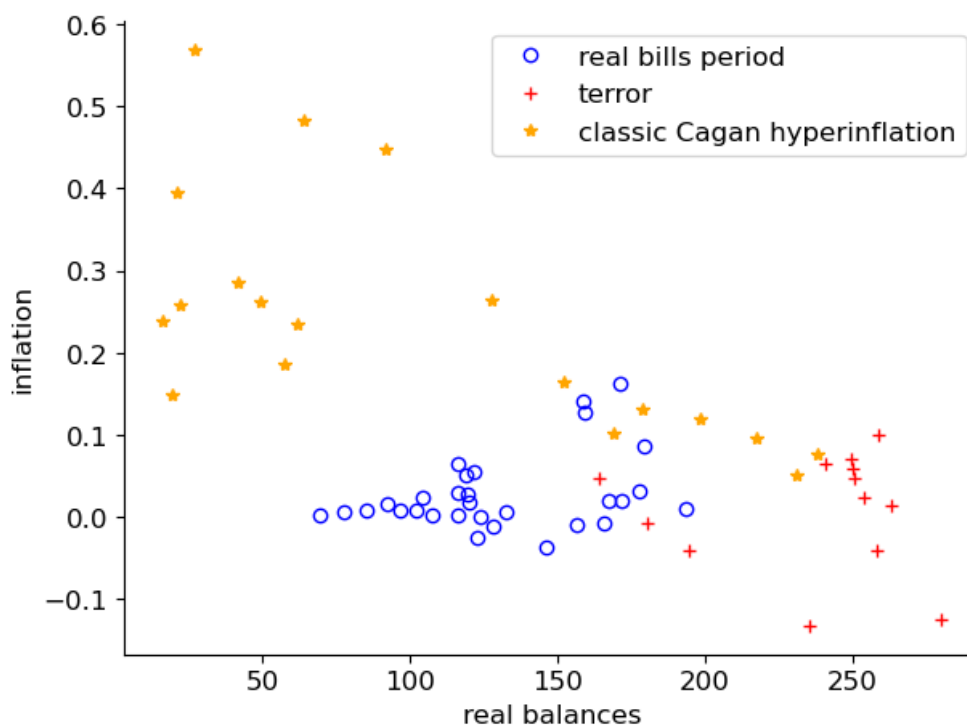
The regression line in Fig. 5.12 shows that large increases in real balances of assignats (paper money) were accompanied by only modest rises in the price level, an outcome in line with the *real bills* theory.
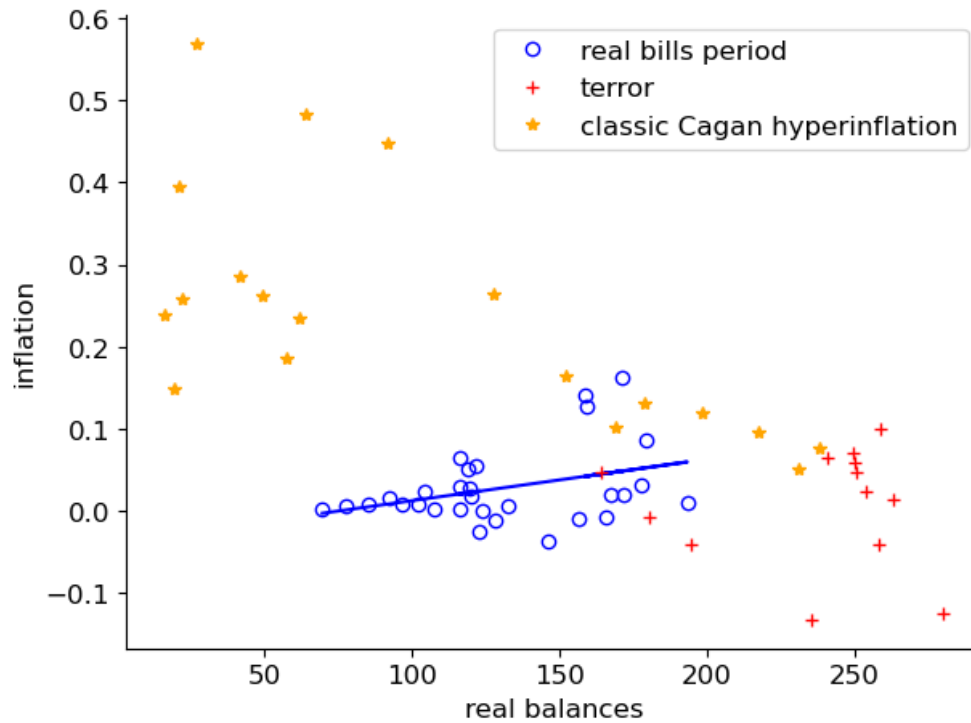
Fig. 5.12: Inflation and Real Balances

During this period, assignats were claims on church lands.

But towards the end of this period, the price level started to rise and real balances to fall as the government continued to print money but stopped selling church land.

To get people to hold that paper money, the government forced people to hold it by using legal restrictions.

Now let's regress real balances on inflation during the terror and plot the regression line.

```python
plt.figure()
plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)

# First subsample
plt.plot(bal[1:31], infl[1:31], 'o', markerfacecolor='none',
         color='blue', label='real bills period')

# Second subsample
plt.plot(bal[31:44], infl[31:44], '+', color='red', label='terror')
plt.plot(a2_rev + b2_rev * infl[31:44], infl[31:44], color='red')

# Third subsample
plt.plot(bal[44:63], infl[44:63], '*',
```

```
        color='orange', label='classic Cagan hyperinflation')

plt.xlabel('real balances')
plt.ylabel('inflation')
plt.legend()

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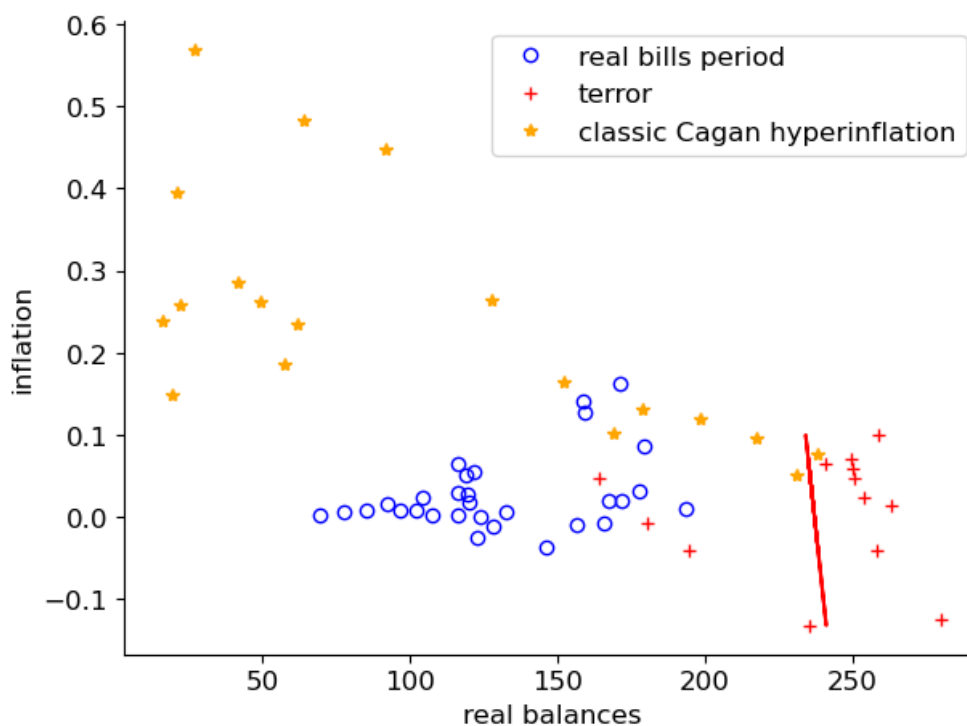

Fig. 5.13: Inflation and Real Balances

The regression line in Fig. 5.13 shows that large increases in real balances of assignats (paper money) were accompanied by little upward price level pressure, even some declines in prices.

This reflects how well legal restrictions –financial repression –was working during the period of the Terror.

But the Terror ended in July 1794. That unleashed a big inflation as people tried to find other ways to transact and store values.

The following two graphs are for the classical hyperinflation period.

One regresses inflation on real balances, the other regresses real balances on inflation.

Both show a prounced inverse relationship that is the hallmark of the hyperinflations studied by Cagan [Cagan, 1956].

```python
plt.figure()
plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)

# First subsample
plt.plot(bal[1:31], infl[1:31], 'o', markerfacecolor='none',
         color='blue', label='real bills period')

# Second subsample
plt.plot(bal[31:44], infl[31:44], '+', color='red', label='terror')

# Third subsample
plt.plot(bal[44:63], infl[44:63], '*',
    color='orange', label='classic Cagan hyperinflation')
plt.plot(bal[44:63], a3 + bal[44:63] * b3, color='orange')

plt.xlabel('real balances')
plt.ylabel('inflation')
plt.legend()

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

Fig. 5.14 shows the results of regressing inflation on real balances during the period of the hyperinflation.

```python
plt.figure()
plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)

# First subsample
plt.plot(bal[1:31], infl[1:31], 'o',
    markerfacecolor='none', color='blue', label='real bills period')

# Second subsample
plt.plot(bal[31:44], infl[31:44], '+', color='red', label='terror')

# Third subsample
plt.plot(bal[44:63], infl[44:63], '*',
        color='orange', label='classic Cagan hyperinflation')
plt.plot(a3_rev + b3_rev * infl[44:63], infl[44:63], color='orange')
```
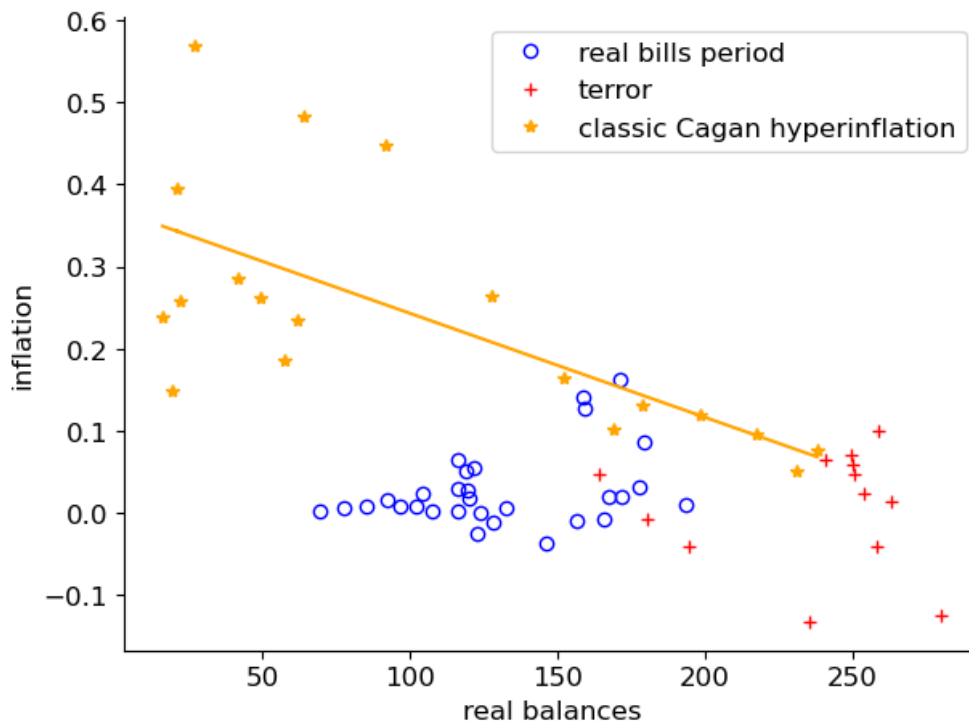
Fig. 5.14: Inflation and Real Balances

```python
plt.xlabel('real balances')
plt.ylabel('inflation')
plt.legend()

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

Fig. 5.14 shows the results of regressing real money balances on inflation during the period of the hyperinflation.

5.6 Hyperinflation Ends

[Sargent and Velde, 1995] tell how in 1797 the Revolutionary government abruptly ended the inflation by

- repudiating 2/3 of the national debt, and thereby

- eliminating the net-of-interest government deficit

- no longer printing money, but instead
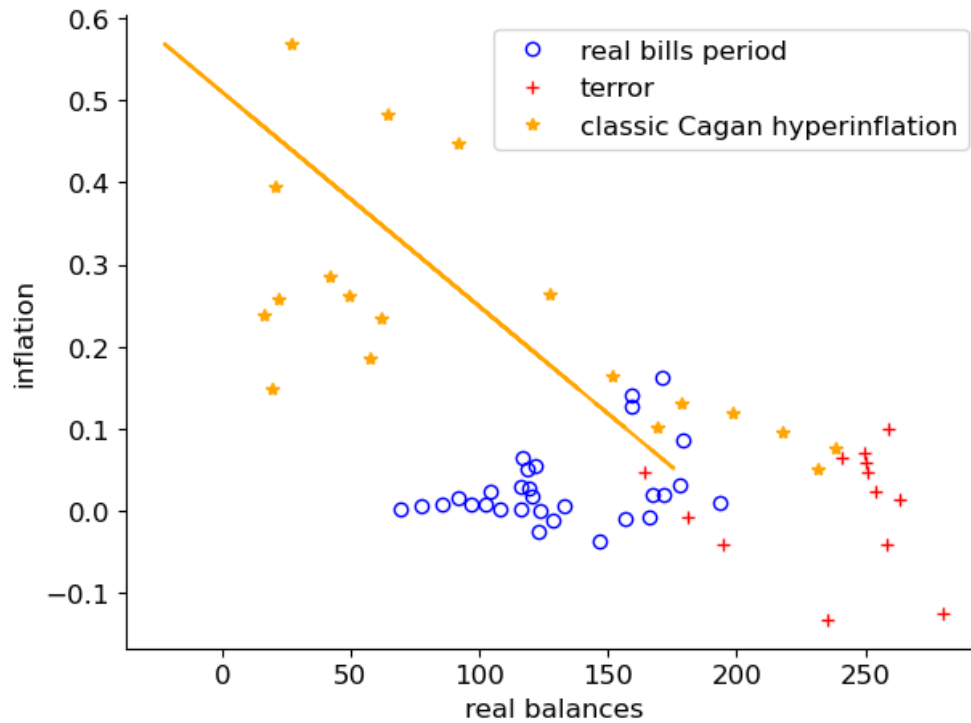
Fig. 5.15: Inflation and Real Balances

- using gold and silver coins as money

In 1799, Napoleon Bonaparte became first consul and for the next 15 years used resources confiscated from conquered territories to help pay for French government expenditures.

5.7 Underlying Theories

This lecture sets the stage for studying theories of inflation and the government monetary and fiscal policies that bring it about.

A *monetarist theory of the price level* is described in this quantecon lecture *A Monetarist Theory of Price Levels*.

That lecture sets the stage for these quantecon lectures *Money Financed Government Deficits and Price Levels* and *Some Unpleasant Monetarist Arithmetic*.

INCOME AND WEALTH INEQUALITY

6.1 Overview

In the lecture 长期增长 we studied how GDP per capita has changed for certain countries and regions.

Per capita GDP is important because it gives us an idea of average income for households in a given country.

However, when we study income and wealth, averages are only part of the story.

Example 6.1.1

For example, imagine two societies, each with one million people, where

- in the first society, the yearly income of one man is $100,000,000 and the income of the others are zero

- in the second society, the yearly income of everyone is $100

These countries have the same income per capita (average income is $100) but the lives of the people will be very different (e.g., almost everyone in the first society is starving, even though one person is fabulously rich).

The example above suggests that we should go beyond simple averages when we study income and wealth.

This leads us to the topic of economic inequality, which examines how income and wealth (and other quantities) are distributed across a population.

In this lecture we study inequality, beginning with measures of inequality and then applying them to wealth and income data from the US and other countries.

6.1.1 Some history

Many historians argue that inequality played a role in the fall of the Roman Republic (see, e.g., [Levitt, 2019]).

Following the defeat of Carthage and the invasion of Spain, money flowed into Rome from across the empire, greatly enriched those in power.

Meanwhile, ordinary citizens were taken from their farms to fight for long periods, diminishing their wealth.

The resulting growth in inequality was a driving factor behind political turmoil that shook the foundations of the republic.

Eventually, the Roman Republic gave way to a series of dictatorships, starting with Octavian (Augustus) in 27 BCE.

This history tells us that inequality matters, in the sense that it can drive major world events.

There are other reasons that inequality might matter, such as how it affects human welfare.

With this motivation, let us start to think about what inequality is and how we can quantify and analyze it.

6.1.2 Measurement

In politics and popular media, the word "inequality" is often used quite loosely, without any firm definition.

To bring a scientific perspective to the topic of inequality we must start with careful definitions.

Hence we begin by discussing ways that inequality can be measured in economic research.

We will need to install the following packages

```
!pip install wbgapi plotly
```

We will also use the following imports.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random as rd
import wbgapi as wb
import plotly.express as px
```

6.2 The Lorenz curve

One popular measure of inequality is the Lorenz curve.

In this section we define the Lorenz curve and examine its properties.

6.2.1 Definition

The Lorenz curve takes a sample $w_1, \dots, w_n$ and produces a curve L.

We suppose that the sample has been sorted from smallest to largest.

To aid our interpretation, suppose that we are measuring wealth

- w_1 is the wealth of the poorest member of the population, and

- w_n is the wealth of the richest member of the population.

The curve L is just a function $y = L(x)$ that we can plot and interpret.

To create it we first generate data points (x_i, y_i) according to

---

Definition 6.2.1

$$x_i = \frac{i}{n}, \qquad y_i = \frac{\sum_{j \leq i} w_j}{\sum_{j \leq n} w_j}, \qquad i = 1, \dots, n$$

Now the Lorenz curve L is formed from these data points using interpolation.

If we use a line plot in `matplotlib`, the interpolation will be done for us.

The meaning of the statement $y = L(x)$ is that the lowest $(100 \times x)\%$ of people have $(100 \times y)\%$ of all wealth.

- if $x = 0.5$ and $y = 0.1$, then the bottom 50% of the population owns 10% of the wealth.

In the discussion above we focused on wealth but the same ideas apply to income, consumption, etc.

6.2.2 Lorenz curves of simulated data

Let's look at some examples and try to build understanding.

First let us construct a `lorenz_curve` function that we can use in our simulations below.

It is useful to construct a function that translates an array of income or wealth data into the cumulative share of individuals (or households) and the cumulative share of income (or wealth).

```python
def lorenz_curve(y):
    """
    Calculates the Lorenz Curve, a graphical representation of
    the distribution of income or wealth.

    It returns the cumulative share of people (x-axis) and
    the cumulative share of income earned.

    Parameters
    ----------
    y : array_like(float or int, ndim=1)
        Array of income/wealth for each individual.
        Unordered or ordered is fine.

    Returns
    -------
    cum_people : array_like(float, ndim=1)
        Cumulative share of people for each person index (i/n)
    cum_income : array_like(float, ndim=1)
        Cumulative share of income for each person index


    References
    ----------
    .. [1] https://en.wikipedia.org/wiki/Lorenz_curve

    Examples
    --------
    >>> a_val, n = 3, 10_000
    >>> y = np.random.pareto(a_val, size=n)
    >>> f_vals, l_vals = lorenz(y)


    """

    n = len(y)
    y = np.sort(y)
```

```
    s = np.zeros(n + 1)
    s[1:] = np.cumsum(y)
    cum_people = np.zeros(n + 1)
    cum_income = np.zeros(n + 1)
    for i in range(1, n + 1):
        cum_people[i] = i / n
        cum_income[i] = s[i] / s[n]
    return cum_people, cum_income
```

In the next figure, we generate $n = 2000$ draws from a lognormal distribution and treat these draws as our population.

The straight 45-degree line ($x = L(x)$ for all x) corresponds to perfect equality.

The log-normal draws produce a less equal distribution.

For example, if we imagine these draws as being observations of wealth across a sample of households, then the dashed lines show that the bottom 80% of households own just over 40% of total wealth.

```
n = 2000
sample = np.exp(np.random.randn(n))


fig, ax = plt.subplots()


f_vals, l_vals = lorenz_curve(sample)
ax.plot(f_vals, l_vals, label=f'lognormal sample', lw=2)
ax.plot(f_vals, f_vals, label='equality', lw=2)

ax.vlines([0.8], [0.0], [0.43], alpha=0.5, colors='k', ls='--')
ax.hlines([0.43], [0], [0.8], alpha=0.5, colors='k', ls='--')
ax.set_xlim((0, 1))
ax.set_xlabel("share of households")
ax.set_ylim((0, 1))
ax.set_ylabel("share of wealth")
ax.legend()
plt.show()
```
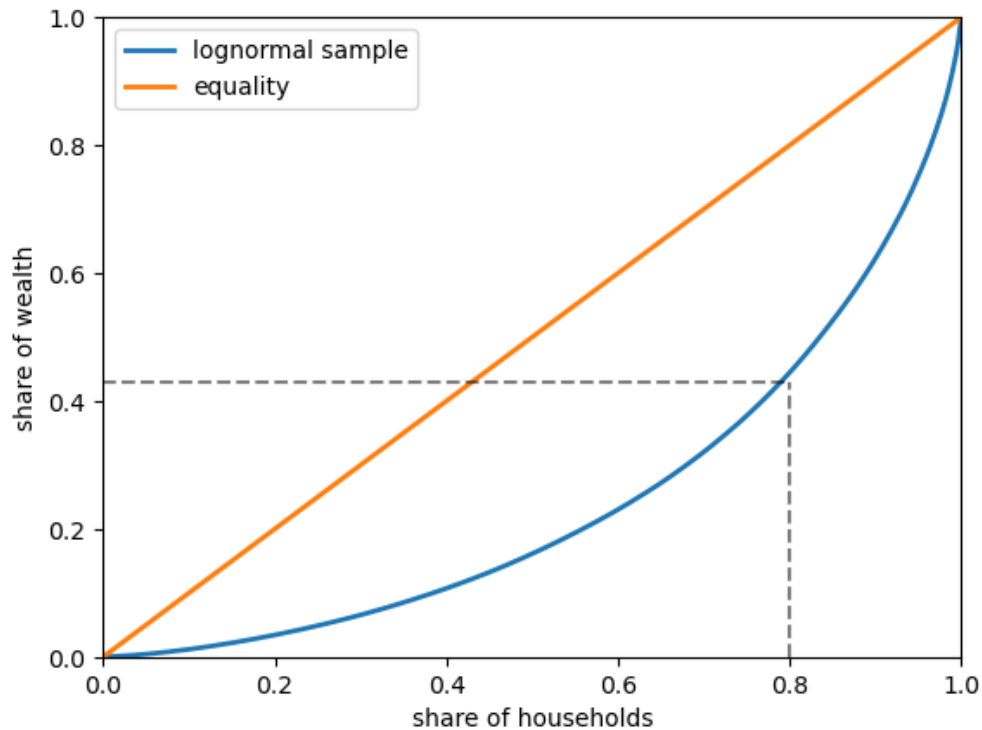
Fig. 6.1: Lorenz curve of simulated wealth data

6.2.3 Lorenz curves for US data

Next let's look at US data for both income and wealth.

The following code block imports a subset of the dataset `SCF_plus` for 2016, which is derived from the Survey of Consumer Finances (SCF).

```
url = 'https://github.com/QuantEcon/high_dim_data/raw/main/SCF_plus/SCF_plus_mini.csv'
df = pd.read_csv(url)
df_income_wealth = df.dropna()
```

```
df_income_wealth.head(n=5)
```

	year	n_wealth	t_income	l_income	weights	nw_groups	ti_groups
0	1950	266933.75	55483.027	0.0	0.998732	50-90%	50-90%
1	1950	87434.46	55483.027	0.0	0.998732	50-90%	50-90%
2	1950	795034.94	55483.027	0.0	0.998732	Top 10%	50-90%
3	1950	94531.78	55483.027	0.0	0.998732	50-90%	50-90%
4	1950	166081.03	55483.027	0.0	0.998732	50-90%	50-90%

The next code block uses data stored in dataframe `df_income_wealth` to generate the Lorenz curves.

(The code is somewhat complex because we need to adjust the data according to population weights supplied by the SCF.)

Now we plot Lorenz curves for net wealth, total income and labor income in the US in 2016.

Total income is the sum of households'all income sources, including labor income but excluding capital gains.

(All income measures are pre-tax.)

```
fig, ax = plt.subplots()
ax.plot(f_vals_nw[-1], l_vals_nw[-1], label=f'net wealth')
ax.plot(f_vals_ti[-1], l_vals_ti[-1], label=f'total income')
ax.plot(f_vals_li[-1], l_vals_li[-1], label=f'labor income')
ax.plot(f_vals_nw[-1], f_vals_nw[-1], label=f'equality')
ax.set_xlabel("share of households")
ax.set_ylabel("share of income/wealth")
ax.legend()
plt.show()
```
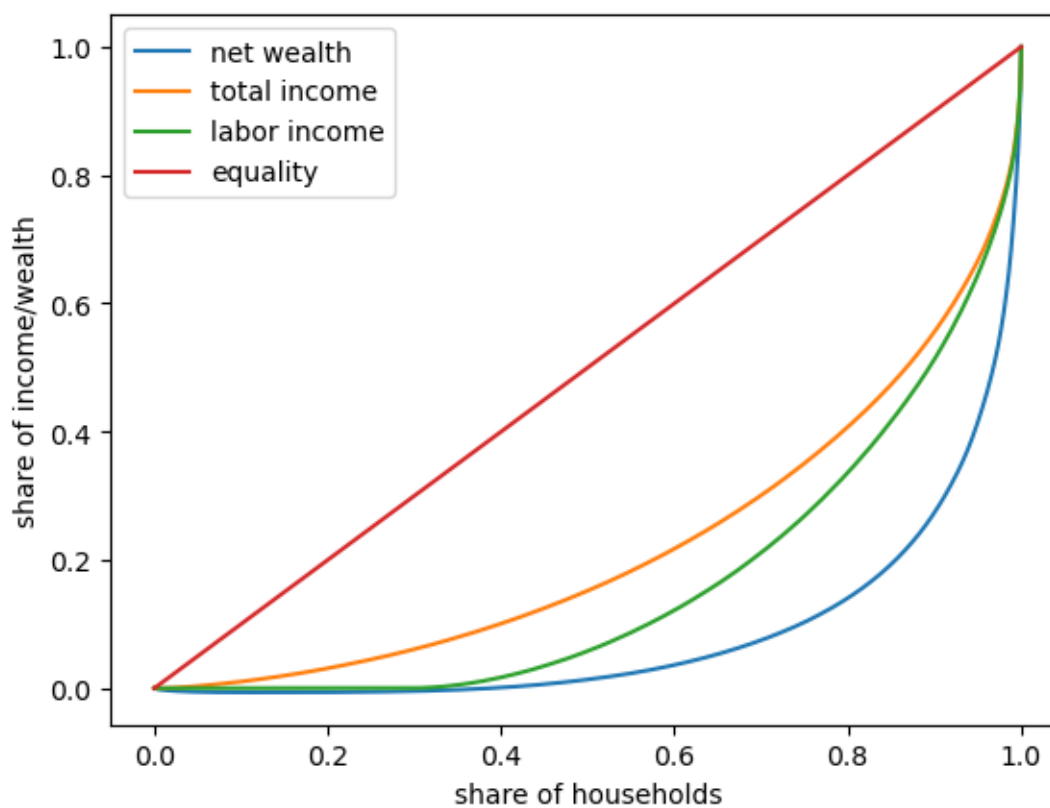

Fig. 6.2: 2016 US Lorenz curves

One key finding from this figure is that wealth inequality is more extreme than income inequality.

6.3 The Gini coefficient

The Lorenz curve provides a visual representation of inequality in a distribution.

Another way to study income and wealth inequality is via the Gini coefficient.

In this section we discuss the Gini coefficient and its relationship to the Lorenz curve.

6.3.1 Definition

As before, suppose that the sample $w_1, \ldots, w_n$ has been sorted from smallest to largest.

The Gini coefficient is defined for the sample above as

Definition 6.3.1

$$G := \frac{\sum_{i=1}^{n} \sum_{j=1}^{n} |w_j - w_i|}{2n \sum_{i=1}^{n} w_i}.$$

The Gini coefficient is closely related to the Lorenz curve.

In fact, it can be shown that its value is twice the area between the line of equality and the Lorenz curve (e.g., the shaded area in Fig. 6.3).

The idea is that $G = 0$ indicates complete equality, while $G = 1$ indicates complete inequality.

```
fig, ax = plt.subplots()
f_vals, l_vals = lorenz_curve(sample)
ax.plot(f_vals, l_vals, label=f'lognormal sample', lw=2)
ax.plot(f_vals, f_vals, label='equality', lw=2)
ax.fill_between(f_vals, l_vals, f_vals, alpha=0.06)
ax.set_ylim((0, 1))
ax.set_xlim((0, 1))
ax.text(0.04, 0.5, r'$G = 2 \times$ shaded area')
ax.set_xlabel("share of households (%)")
ax.set_ylabel("share of wealth (%)")
ax.legend()
plt.show()
```
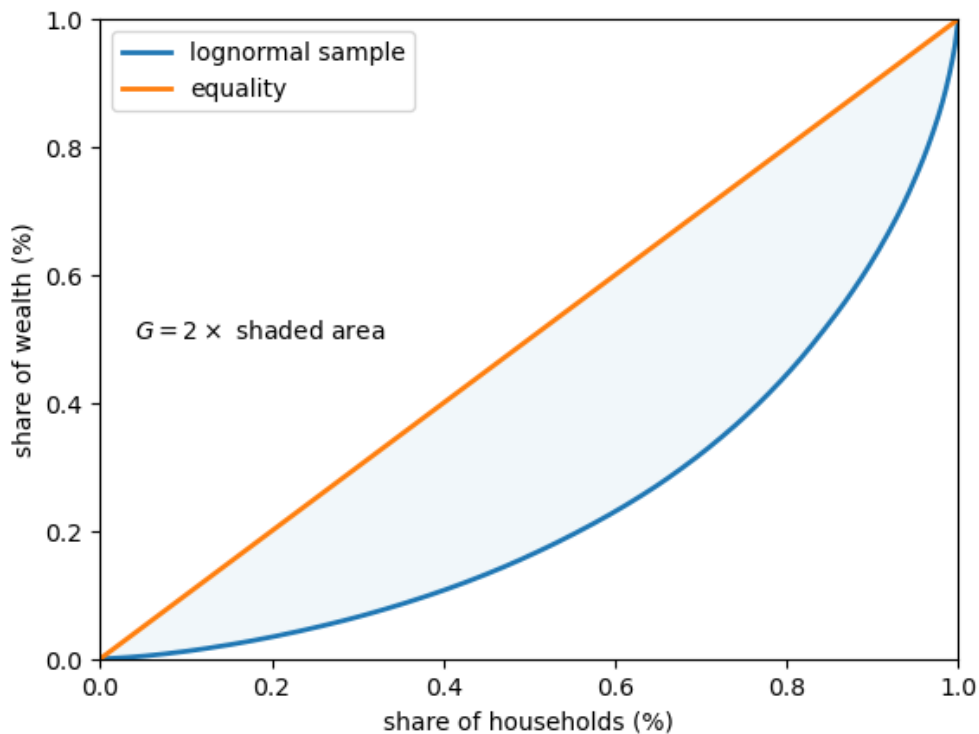
Fig. 6.3: Gini coefficient (simulated wealth data)

In fact the Gini coefficient can also be expressed as

$$G = \frac{A}{A + B}$$

where A is the area between the 45-degree line of perfect equality and the Lorenz curve, while B is the area below the Lorenze curve –see Fig. 6.4.

```
fig, ax = plt.subplots()
f_vals, l_vals = lorenz_curve(sample)
ax.plot(f_vals, l_vals, label='lognormal sample', lw=2)
ax.plot(f_vals, f_vals, label='equality', lw=2)
ax.fill_between(f_vals, l_vals, f_vals, alpha=0.06)
ax.fill_between(f_vals, l_vals, np.zeros_like(f_vals), alpha=0.06)
ax.set_ylim((0, 1))
ax.set_xlim((0, 1))
ax.text(0.55, 0.4, 'A')
ax.text(0.75, 0.15, 'B')
ax.set_xlabel("share of households")
ax.set_ylabel("share of wealth")
ax.legend()
plt.show()
```
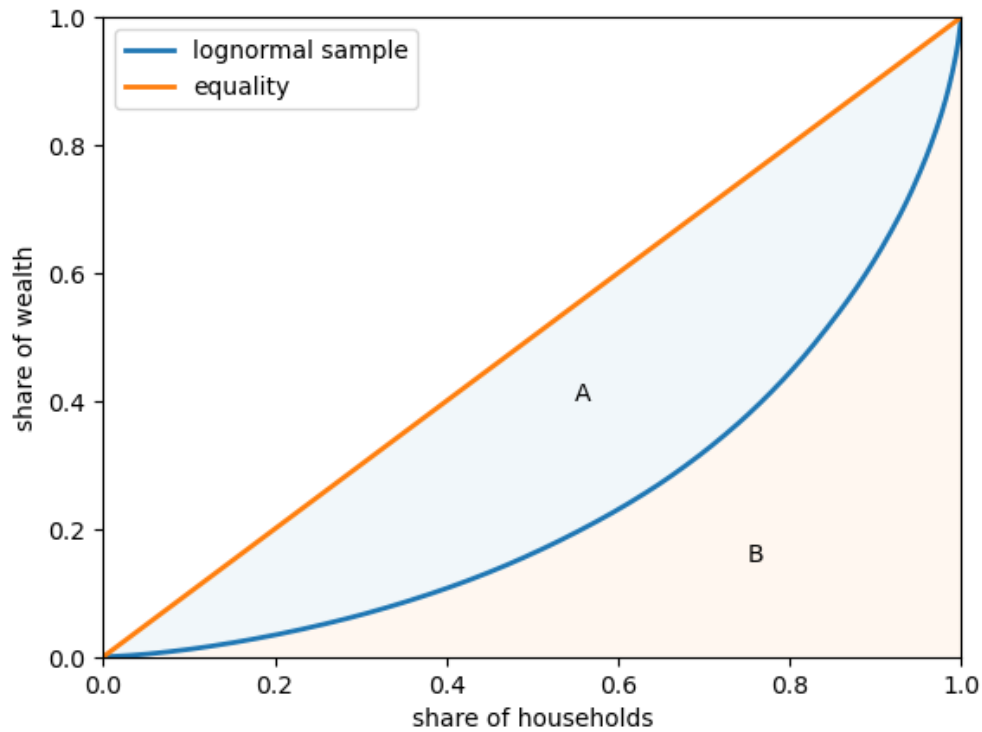
Fig. 6.4: Lorenz curve and Gini coefficient

See also:

The World in Data project has a graphical exploration of the Lorenz curve and the Gini coefficient

6.3.2 Gini coefficient of simulated data

Let's examine the Gini coefficient in some simulations.

The code below computes the Gini coefficient from a sample.

```
def gini_coefficient(y):
    r"""
    Implements the Gini inequality index

    Parameters
    ----------
    y : array_like(float)
        Array of income/wealth for each individual.
        Ordered or unordered is fine

    Returns
```

```
    -------
    Gini index: float
        The gini index describing the inequality of the array of income/wealth

    References
    ----------

    https://en.wikipedia.org/wiki/Gini_coefficient
    """
    n = len(y)
    i_sum = np.zeros(n)
    for i in range(n):
        for j in range(n):
            i_sum[i] += abs(y[i] - y[j])
    return np.sum(i_sum) / (2 * n * np.sum(y))
```

Now we can compute the Gini coefficients for five different populations.

Each of these populations is generated by drawing from a lognormal distribution with parameters μ (mean) and σ (standard deviation).

To create the five populations, we vary σ over a grid of length 5 between 0.2 and 4.

In each case we set $\mu = -\sigma^2/2$.

This implies that the mean of the distribution does not change with σ.

You can check this by looking up the expression for the mean of a lognormal distribution.

```
%%time
k = 5
σ_vals = np.linspace(0.2, 4, k)
n = 2_000

ginis = []

for σ in σ_vals:
    μ = -σ**2 / 2
    y = np.exp(μ + σ * np.random.randn(n))
    ginis.append(gini_coefficient(y))
```

```
CPU times: user 5.1 s, sys: 36.6 ms, total: 5.14 s
Wall time: 4.81 s
```

Let's build a function that returns a figure (so that we can use it later in the lecture).

6.3. The Gini coefficient

```
def plot_inequality_measures(x, y, legend, xlabel, ylabel):
    fig, ax = plt.subplots()
    ax.plot(x, y, marker='o', label=legend)
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    ax.legend()
    return fig, ax
```

```
fix, ax = plot_inequality_measures(σ_vals,
                                   ginis,
                                   'simulated',
                                   '$\sigma$',
                                   'Gini coefficients')
plt.show()
```
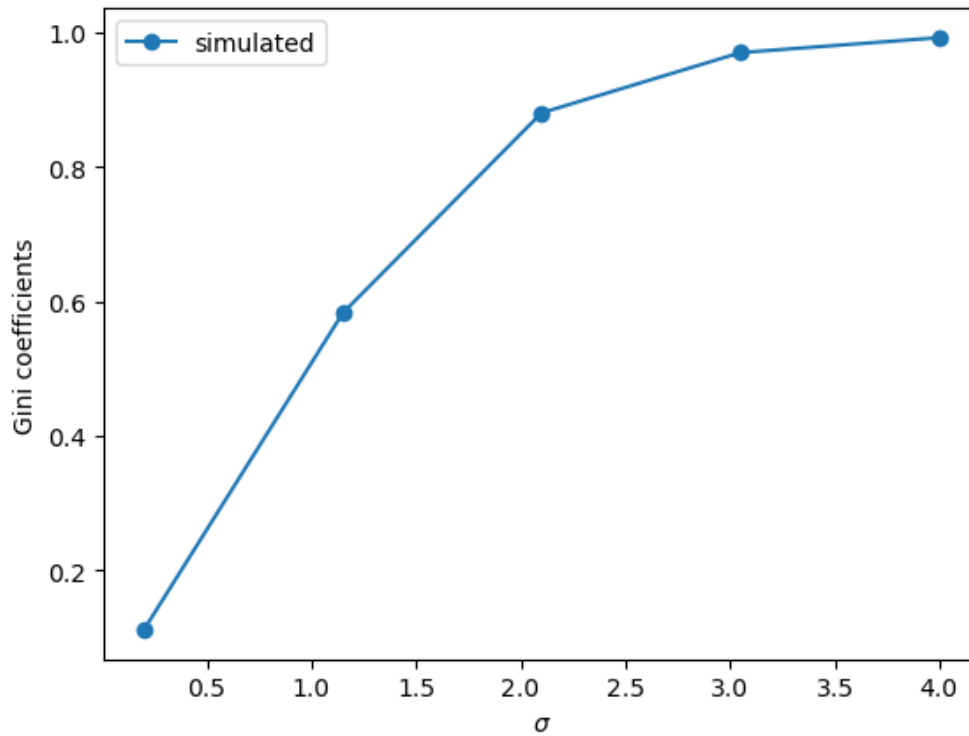

Fig. 6.5: Gini coefficients of simulated data

The plots show that inequality rises with σ, according to the Gini coefficient.

6.3.3 Gini coefficient for income (US data)

Let's look at the Gini coefficient for the distribution of income in the US.

We will get pre-computed Gini coefficients (based on income) from the World Bank using the wbgapi.

Let's use the `wbgapi` package we imported earlier to search the World Bank data for Gini to find the Series ID.

```
wb.search("gini")
```

```
========
Series: SI.POV.GINI


Developmentrelevance: ...growth of the bottom 40 per cent of the welfare␣
 ↪distribution in every country. Gini coefficients are important background␣
 ↪information for shared prosperity....
--------
IndicatorName: Gini index
--------
Limitationsandexceptions: ...Gini coefficients are not unique. It is possible for␣
 ↪two different Lorenz curves to...
--------
Longdefinition: ...Gini index measures the extent to which the distribution of␣
 ↪income (or, in some...
--------
Shortdefinition: ...The Gini index measures the extent to which the distribution␣
 ↪of income or consumption...
--------
Statisticalconceptandmethodology: ...The Gini index measures the area between the␣
 ↪Lorenz curve and a hypothetical line of...
```

We now know the series ID is `SI.POV.GINI`.

(Another way to find the series ID is to use the World Bank data portal and then use `wbgapi` to fetch the data.)

To get a quick overview, let's histogram Gini coefficients across all countries and all years in the World Bank dataset.

```python
# Fetch gini data for all countries
gini_all = wb.data.DataFrame("SI.POV.GINI")
# remove 'YR' in index and convert to integer
gini_all.columns = gini_all.columns.map(lambda x: int(x.replace('YR','')))

# Create a long series with a multi-index of the data to get global min and max values
gini_all = gini_all.unstack(level='economy').dropna()
```

```
# Build a histogram
ax = gini_all.plot(kind="hist", bins=20)
ax.set_xlabel("Gini coefficient")
ax.set_ylabel("frequency")
plt.show()
```
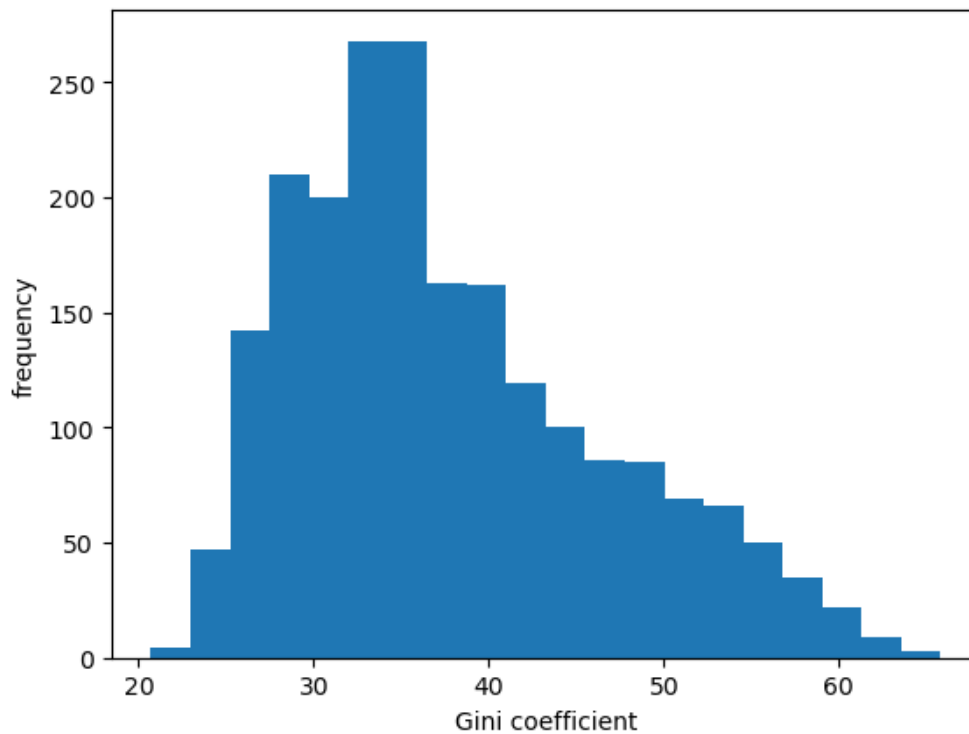

Fig. 6.6: Histogram of Gini coefficients across countries

We can see in Fig. 6.6 that across 50 years of data and all countries the measure varies between 20 and 65.

Let us fetch the data `DataFrame` for the USA.

```
data = wb.data.DataFrame("SI.POV.GINI", "USA")
data.head(n=5)
# remove 'YR' in index and convert to integer
data.columns = data.columns.map(lambda x: int(x.replace('YR','')))
```

(This package often returns data with year information contained in the columns. This is not always convenient for simple plotting with pandas so it can be useful to transpose the results before plotting.)

```
data = data.T              # Obtain years as rows
data_usa = data['USA']   # pd.Series of US data
```

Let us take a look at the data for the US.

```
fig, ax = plt.subplots()
ax = data_usa.plot(ax=ax)
ax.set_ylim(data_usa.min()-1, data_usa.max()+1)
ax.set_ylabel("Gini coefficient (income)")
ax.set_xlabel("year")
plt.show()
```
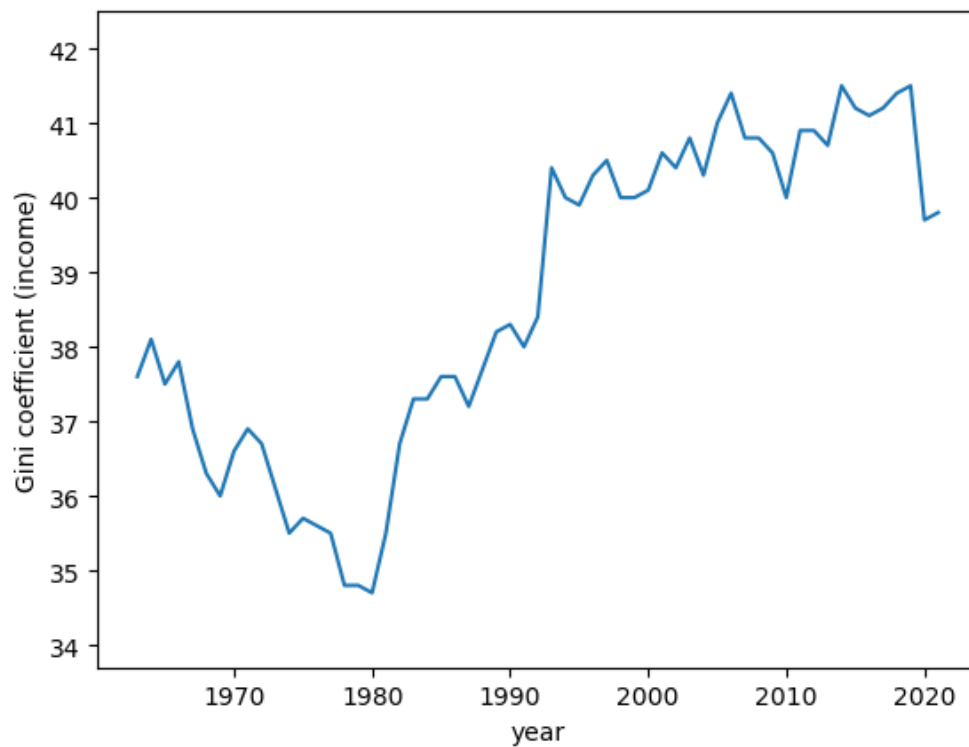

Fig. 6.7: Gini coefficients for income distribution (USA)

As can be seen in Fig. 6.7, the income Gini trended upward from 1980 to 2020 and then dropped following at the start of the COVID pandemic.

6.3.4 Gini coefficient for wealth

In the previous section we looked at the Gini coefficient for income, focusing on using US data.

Now let's look at the Gini coefficient for the distribution of wealth.

We will use US data from the *Survey of Consumer Finances*

```
df_income_wealth.year.describe()
```

```
count    509455.000000
mean       1982.122062
std          22.607350
min        1950.000000
25%        1959.000000
50%        1983.000000
75%        2004.000000
max        2016.000000
Name: year, dtype: float64
```

This notebook can be used to compute this information over the full dataset.

```
data_url = 'https://github.com/QuantEcon/lecture-python-intro/raw/main/lectures/_
 ↪static/lecture_specific/inequality/usa-gini-nwealth-tincome-lincome.csv'
ginis = pd.read_csv(data_url, index_col='year')
ginis.head(n=5)
```

```
      n_wealth   t_income   l_income
year
1950  0.825733   0.442487   0.534295
1953  0.805949   0.426454   0.515898
1956  0.812179   0.444269   0.534929
1959  0.795207   0.437493   0.521399
1962  0.808695   0.443584   0.534513
```

Let's plot the Gini coefficients for net wealth.

```
fig, ax = plt.subplots()
ax.plot(years, ginis["n_wealth"], marker='o')
ax.set_xlabel("year")
ax.set_ylabel("Gini coefficient")
plt.show()
```
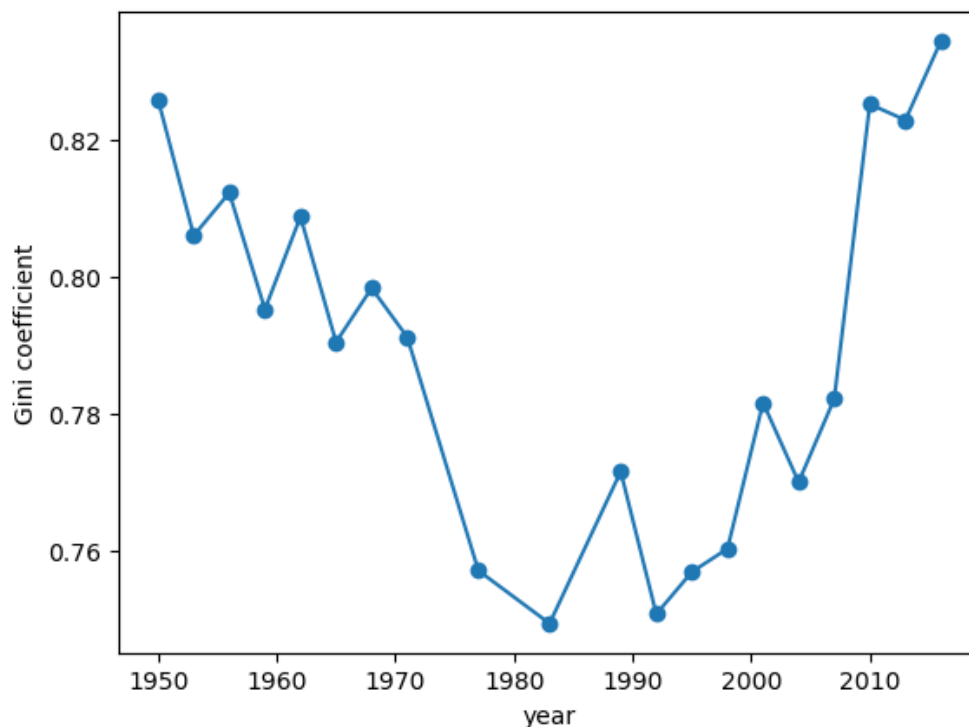
Fig. 6.8: Gini coefficients of US net wealth

The time series for the wealth Gini exhibits a U-shape, falling until the early 1980s and then increasing rapidly.

One possibility is that this change is mainly driven by technology.

However, we will see below that not all advanced economies experienced similar growth of inequality.

6.3.5 Cross-country comparisons of income inequality

Earlier in this lecture we used `wbgapi` to get Gini data across many countries and saved it in a variable called `gini_all`

In this section we will use this data to compare several advanced economies, and to look at the evolution in their respective income Ginis.

```
data = gini_all.unstack()
data.columns
```

```
Index(['USA', 'GBR', 'FRA', 'CAN', 'SWE', 'IND', 'ITA', 'ISR', 'NOR', 'PAN',
       ...
       'NRU', 'SRB', 'ARE', 'SYC', 'LCA', 'MMR', 'TUR', 'GRD', 'MHL', 'SUR'],
      dtype='object', name='economy', length=168)
```

There are 167 countries represented in this dataset.

Let us compare three advanced economies: the US, the UK, and Norway

```
ax = data[['USA','GBR', 'NOR']].plot()
ax.set_xlabel('year')
ax.set_ylabel('Gini coefficient')
ax.legend(title="")
plt.show()
```
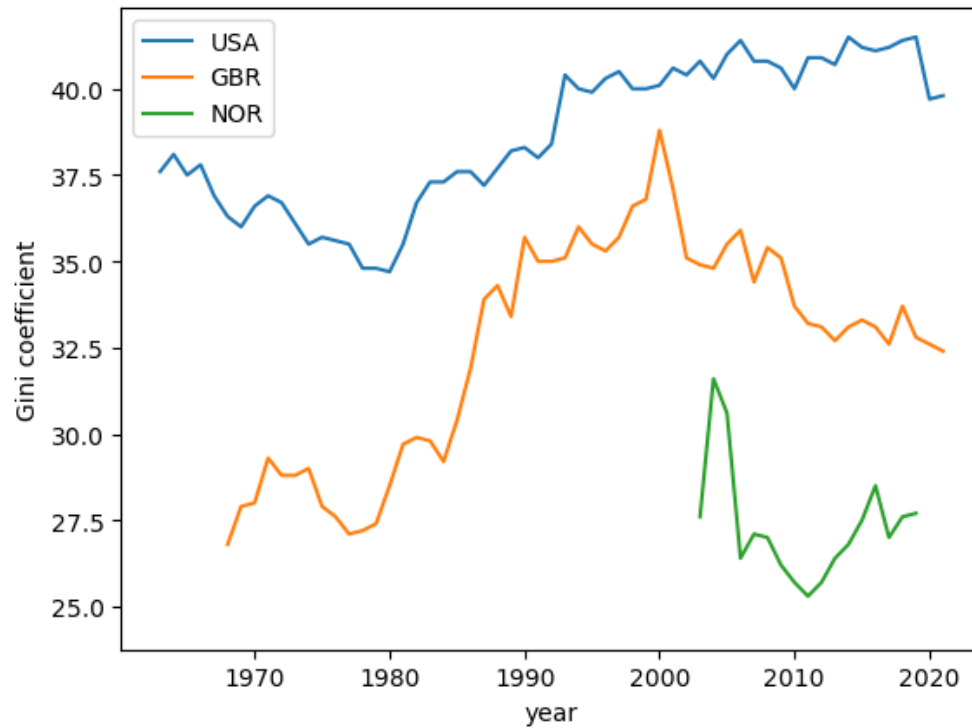

Fig. 6.9: Gini coefficients for income (USA, United Kingdom, and Norway)

We see that Norway has a shorter time series.

Let us take a closer look at the underlying data and see if we can rectify this.

```
data[['NOR']].dropna().head(n=5)
```

```
economy    NOR
1979      26.9
1986      24.6
1991      25.2
1995      26.0
2000      27.4
```

The data for Norway in this dataset goes back to 1979 but there are gaps in the time series and matplotlib is not showing those data points.

We can use the `.ffill()` method to copy and bring forward the last known value in a series to fill in these gaps

```python
data['NOR'] = data['NOR'].ffill()
ax = data[['USA','GBR', 'NOR']].plot()
ax.set_xlabel('year')
ax.set_ylabel('Gini coefficient')
ax.legend(title="")
plt.show()
```
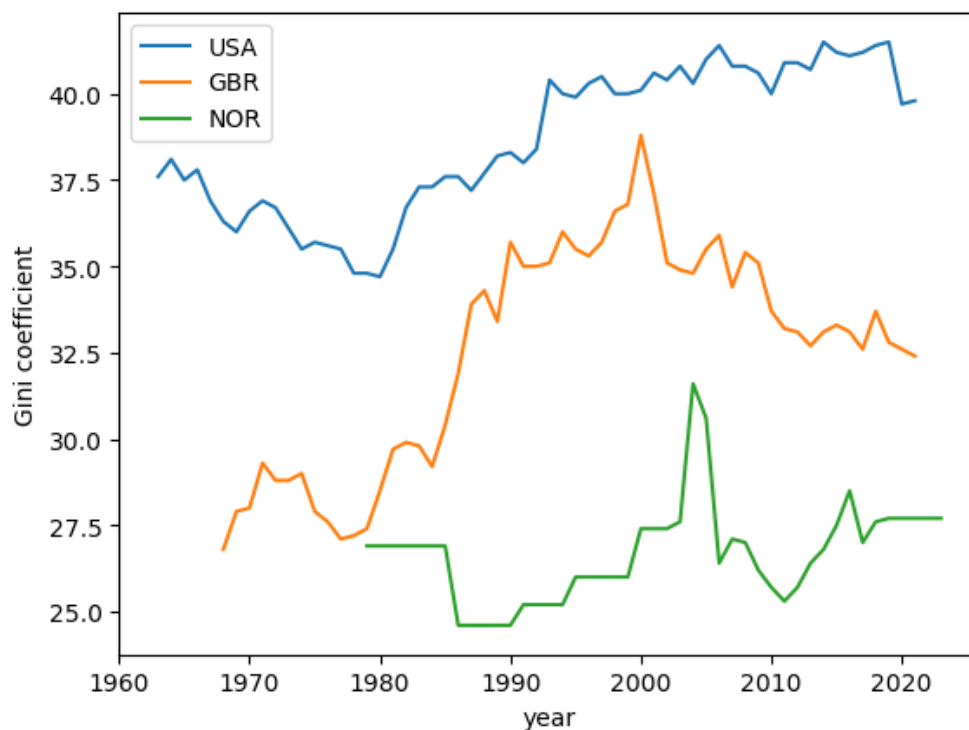

Fig. 6.10: Gini coefficients for income (USA, United Kingdom, and Norway)

From this plot we can observe that the US has a higher Gini coefficient (i.e. higher income inequality) when compared to the UK and Norway.

Norway has the lowest Gini coefficient over the three economies and, moreover, the Gini coefficient shows no upward trend.

6.3.6 Gini Coefficient and GDP per capita (over time)

We can also look at how the Gini coefficient compares with GDP per capita (over time).

Let's take another look at the US, Norway, and the UK.

```python
countries = ['USA', 'NOR', 'GBR']
gdppc = wb.data.DataFrame("NY.GDP.PCAP.KD", countries)
# remove 'YR' in index and convert to integer
gdppc.columns = gdppc.columns.map(lambda x: int(x.replace('YR','')))
gdppc = gdppc.T
```

We can rearrange the data so that we can plot GDP per capita and the Gini coefficient across years

```python
plot_data = pd.DataFrame(data[countries].unstack())
plot_data.index.names = ['country', 'year']
plot_data.columns = ['gini']
```

Now we can get the GDP per capita data into a shape that can be merged with `plot_data`

```python
pgdppc = pd.DataFrame(gdppc.unstack())
pgdppc.index.names = ['country', 'year']
pgdppc.columns = ['gdppc']
plot_data = plot_data.merge(pgdppc, left_index=True, right_index=True)
plot_data.reset_index(inplace=True)
```

Now we use Plotly to build a plot with GDP per capita on the y-axis and the Gini coefficient on the x-axis.

```python
min_year = plot_data.year.min()
max_year = plot_data.year.max()
```

The time series for all three countries start and stop in different years.

We will add a year mask to the data to improve clarity in the chart including the different end years associated with each country's time series.

```python
labels = [1979, 1986, 1991, 1995, 2000, 2020, 2021, 2022] + \
        list(range(min_year,max_year,5))
plot_data.year = plot_data.year.map(lambda x: x if x in labels else None)
```

```python
fig = px.line(plot_data,
            x = "gini",
            y = "gdppc",
```

(continues on next page)

```
                color = "country",
                text = "year",
                height = 800,
                labels = {"gini" : "Gini coefficient", "gdppc" : "GDP per capita"}
            )
fig.update_traces(textposition="bottom right")
fig.show()
```

This figure is built using `plotly` and is *available on the website*

This plot shows that all three Western economies'GDP per capita has grown over time with some fluctuations in the Gini coefficient.

From the early 80's the United Kingdom and the US economies both saw increases in income inequality.

Interestingly, since the year 2000, the United Kingdom saw a decline in income inequality while the US exhibits persistent but stable levels around a Gini coefficient of 40.

6.4 Top shares

Another popular measure of inequality is the top shares.

In this section we show how to compute top shares.

6.4.1 Definition

As before, suppose that the sample $w_1, \dots, w_n$ has been sorted from smallest to largest.

Given the Lorenz curve $y = L(x)$ defined above, the top $100 \times p\%$ share is defined as

---

Definition 6.4.1

$$T(p) = 1 - L(1-p) \approx \frac{\sum_{j \geq i} w_j}{\sum_{j \leq n} w_j}, \quad i = \lfloor n(1-p) \rfloor \tag{6.1}$$

Here $\lfloor \cdot \rfloor$ is the floor function, which rounds any number down to the integer less than or equal to that number.

The following code uses the data from dataframe `df_income_wealth` to generate another dataframe `df_topshares`.

`df_topshares` stores the top 10 percent shares for the total income, the labor income and net wealth from 1950 to 2016 in US.

Then let's plot the top shares.

```
fig, ax = plt.subplots()
ax.plot(years, df_topshares["topshare_l_income"],
        marker='o', label="labor income")
ax.plot(years, df_topshares["topshare_n_wealth"],
        marker='o', label="net wealth")
ax.plot(years, df_topshares["topshare_t_income"],
        marker='o', label="total income")
ax.set_xlabel("year")
ax.set_ylabel("top $10\%$ share")
ax.legend()
plt.show()
```


Fig. 6.11: US top shares

6.5 Exercises

Exercise 6.5.1

Using simulation, compute the top 10 percent shares for the collection of lognormal distributions associated with the random variables $w_\sigma = \exp(\mu + \sigma Z)$, where $Z \sim N(0, 1)$ and σ varies over a finite grid between 0.2 and 4.

As σ increases, so does the variance of w_σ.

To focus on volatility, adjust μ at each step to maintain the equality $\mu = -\sigma^2/2$.

For each σ, generate 2,000 independent draws of w_σ and calculate the Lorenz curve and Gini coefficient.

Confirm that higher variance generates more dispersion in the sample, and hence greater inequality.

Solution to Exercise 6.5.1

Here is one solution:

```python
def calculate_top_share(s, p=0.1):

    s = np.sort(s)
    n = len(s)
    index = int(n * (1 - p))
    return s[index:].sum() / s.sum()
```

```python
k = 5
σ_vals = np.linspace(0.2, 4, k)
n = 2_000

topshares = []
ginis = []
f_vals = []
l_vals = []

for σ in σ_vals:
    μ = -σ ** 2 / 2
    y = np.exp(μ + σ * np.random.randn(n))
    f_val, l_val = lorenz_curve(y)
    f_vals.append(f_val)
    l_vals.append(l_val)
    ginis.append(gini_coefficient(y))
```

```
    topshares.append(calculate_top_share(y))
```

```
fig, ax = plot_inequality_measures(σ_vals,
                                   topshares,
                                   "simulated data",
                                   "$\sigma$",
                                   "top $10\%$ share")
plt.show()
```
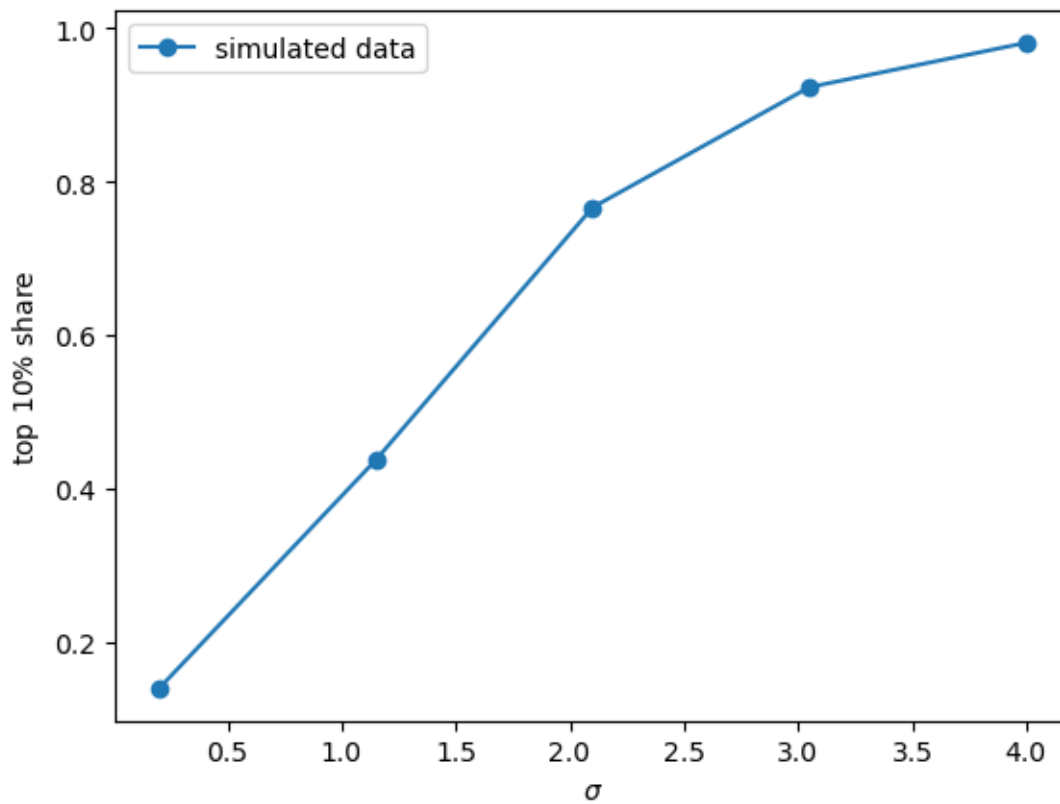

Fig. 6.12: Top shares of simulated data

```
fig, ax = plot_inequality_measures(σ_vals,
                                   ginis,
                                   "simulated data",
                                   "$\sigma$",
                                   "gini coefficient")
plt.show()
```
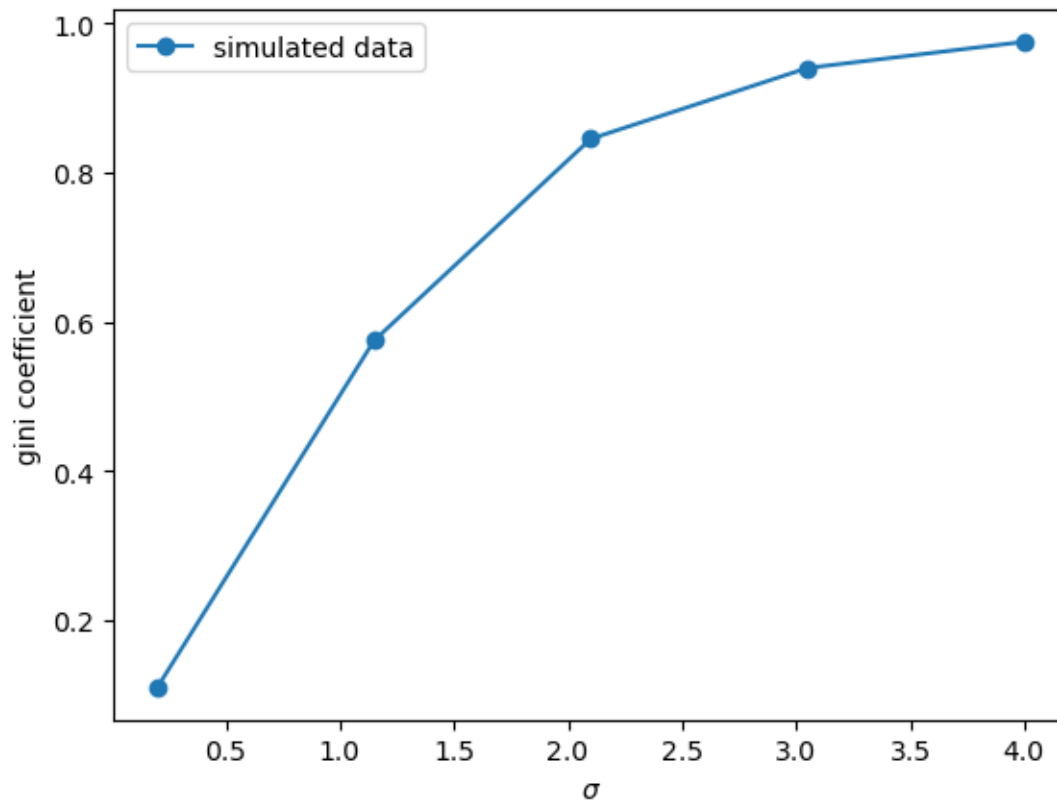
Fig. 6.13: Gini coefficients of simulated data

```
fig, ax = plt.subplots()
ax.plot([0,1],[0,1], label=f"equality")
for i in range(len(f_vals)):
    ax.plot(f_vals[i], l_vals[i], label=f"$\sigma$ = {σ_vals[i]}")
plt.legend()
plt.show()
```
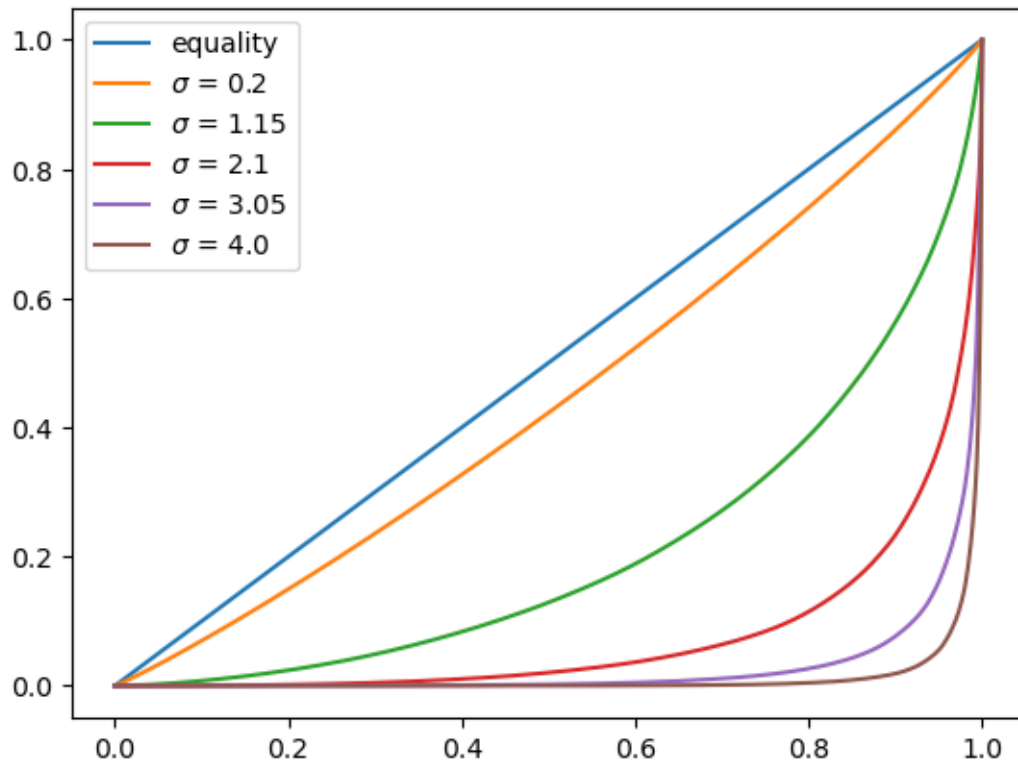

Fig. 6.14: Lorenz curves for simulated data

Exercise 6.5.2

According to the definition of the top shares (6.1) we can also calculate the top percentile shares using the Lorenz curve.

Compute the top shares of US net wealth using the corresponding Lorenz curves data: `f_vals_nw, l_vals_nw` and linear interpolation.

Plot the top shares generated from Lorenz curve and the top shares approximated from data together.

Solution to Exercise 6.5.2

Here is one solution:

```python
def lorenz2top(f_val, l_val, p=0.1):
    t = lambda x: np.interp(x, f_val, l_val)
    return 1- t(1 - p)
```

```python
top_shares_nw = []
for f_val, l_val in zip(f_vals_nw, l_vals_nw):
    top_shares_nw.append(lorenz2top(f_val, l_val))
```

```python
fig, ax = plt.subplots()

ax.plot(years, df_topshares["topshare_n_wealth"], marker='o',\
    label="net wealth-approx")
ax.plot(years, top_shares_nw, marker='o', label="net wealth-lorenz")

ax.set_xlabel("year")
ax.set_ylabel("top $10\%$ share")
ax.legend()
plt.show()
```
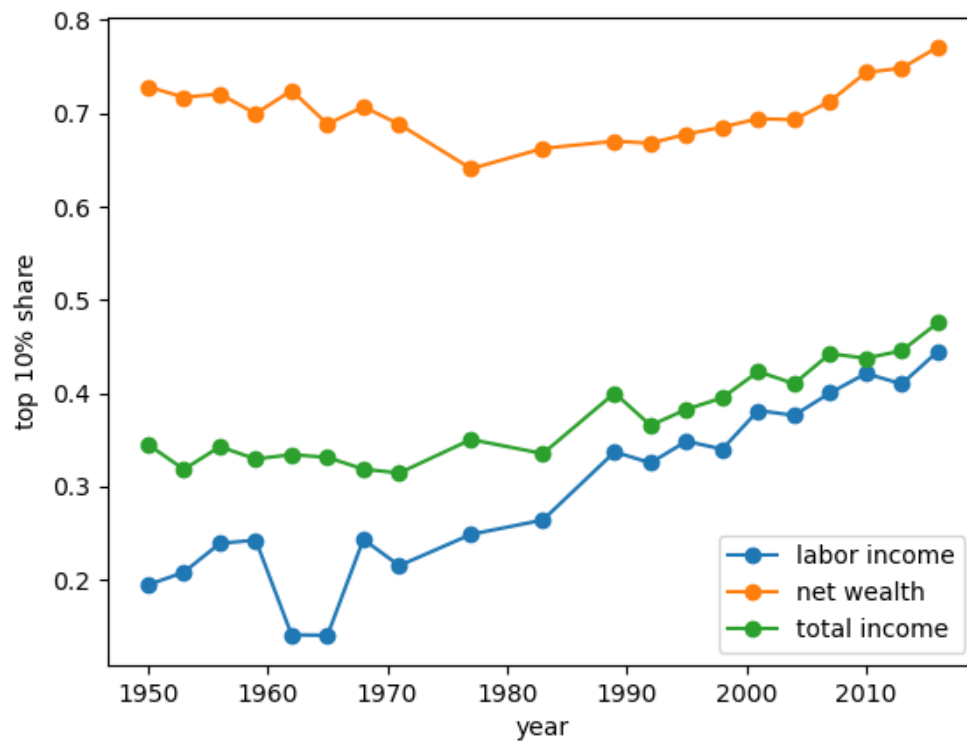
Exercise 6.5.3

The *code to compute the Gini coefficient is listed in the lecture above*.

This code uses loops to calculate the coefficient based on income or wealth data.

This function can be re-written using vectorization which will greatly improve the computational efficiency when using `python`.

Re-write the function `gini_coefficient` using `numpy` and vectorized code.

You can compare the output of this new function with the one above, and note the speed differences.

Solution to Exercise 6.5.3

Let's take a look at some raw data for the US that is stored in `df_income_wealth`
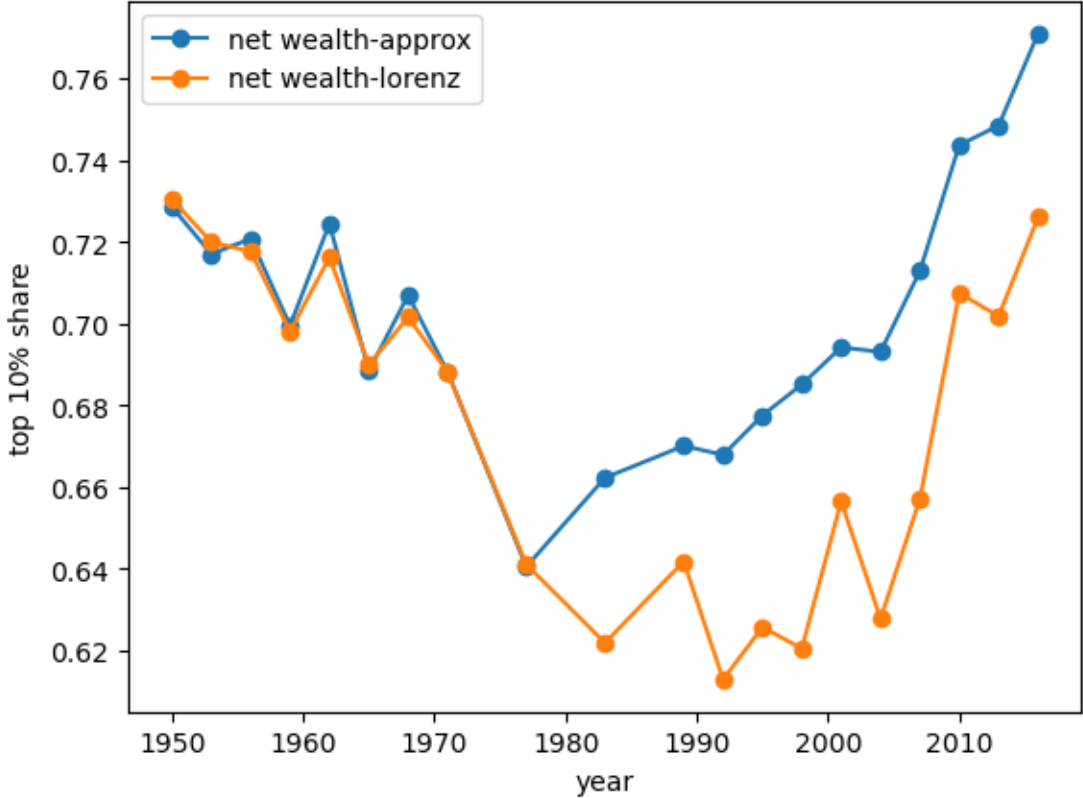
Fig. 6.15: US top shares: approximation vs Lorenz

```
df_income_wealth.describe()
```

```
                year       n_wealth       t_income       l_income       weights
count   509455.000000   5.094550e+05   5.094550e+05   5.094550e+05   509455.000000
mean      1982.122062   4.512145e+06   3.255242e+05   9.525005e+04        3.294007
std         22.607350   3.477071e+07   3.160138e+06   8.316296e+05        2.671516
min       1950.000000  -2.340803e+08  -8.001954e+07   0.000000e+00        0.000000
25%       1959.000000   1.357817e+04   2.614322e+04   0.000000e+00        1.207430
50%       1983.000000   8.484058e+04   4.812237e+04   3.247179e+04        2.380133
75%       2004.000000   3.622574e+05   9.077778e+04   6.582137e+04        5.017505
max       2016.000000   2.928346e+09   3.056805e+08   1.115575e+08       31.052229
```

```
df_income_wealth.head(n=4)
```

```
   year   n_wealth   t_income  l_income   weights nw_groups ti_groups
0  1950  266933.75  55483.027       0.0  0.998732    50-90%    50-90%
1  1950   87434.46  55483.027       0.0  0.998732    50-90%    50-90%
2  1950  795034.94  55483.027       0.0  0.998732   Top 10%    50-90%
3  1950   94531.78  55483.027       0.0  0.998732    50-90%    50-90%
```

We will focus on wealth variable `n_wealth` to compute a Gini coefficient for the year 2016.

```
data = df_income_wealth[df_income_wealth.year == 2016].sample(3000, random_state=1)
```

```
data.head(n=2)
```

```
        year   n_wealth   t_income  l_income   weights nw_groups ti_groups
479748  2016        0.0   9214.991       0.0  4.546196     0-50%     0-50%
495754  2016  6007000.0  36454.914       0.0  2.925190   Top 10%     0-50%
```

We can first compute the Gini coefficient using the function defined in the lecture above.

```
gini_coefficient(data.n_wealth.values)
```

```
0.9300769449032591
```

Now we can write a vectorized version using `numpy`

```python
def gini(y):
    n = len(y)
```

(continues on next page)

```
    y_1 = np.reshape(y, (n, 1))
    y_2 = np.reshape(y, (1, n))
    g_sum = np.sum(np.abs(y_1 - y_2))
    return g_sum / (2 * n * np.sum(y))
```

```
gini(data.n_wealth.values)
```

```
0.9300769449032564
```

Let's simulate five populations by drawing from a lognormal distribution as before

```
k = 5
σ_vals = np.linspace(0.2, 4, k)
n = 2_000
σ_vals = σ_vals.reshape((k,1))
μ_vals = -σ_vals**2/2
y_vals = np.exp(μ_vals + σ_vals*np.random.randn(n))
```

We can compute the Gini coefficient for these five populations using the vectorized function, the computation time is shown below:

```
%%time
gini_coefficients =[]
for i in range(k):
    gini_coefficients.append(gini(y_vals[i]))
```

```
CPU times: user 24 ms, sys: 25.1 ms, total: 49.1 ms
Wall time: 49.1 ms
```

This shows the vectorized function is much faster. This gives us the Gini coefficients for these five households.

```
gini_coefficients
```

```
[0.11225417473811854,
 0.5808615811182404,
 0.8582617168672961,
 0.9632316769785193,
 0.9899997967704439]
```

Part III

Foundations

SEVEN

INTRODUCTION TO SUPPLY AND DEMAND

7.1 Overview

This lecture is about some models of equilibrium prices and quantities, one of the core topics of elementary microeconomics.

Throughout the lecture, we focus on models with one good and one price.

See also:

In a *subsequent lecture* we will investigate settings with many goods.

7.1.1 Why does this model matter?

In the 15th, 16th, 17th and 18th centuries, mercantilist ideas held sway among most rulers of European countries.

Exports were regarded as good because they brought in bullion (gold flowed into the country).

Imports were regarded as bad because bullion was required to pay for them (gold flowed out).

This zero-sum view of economics was eventually overturned by the work of the classical economists such as Adam Smith and David Ricardo, who showed how freeing domestic and international trade can enhance welfare.

There are many different expressions of this idea in economics.

This lecture discusses one of the simplest: how free adjustment of prices can maximize a measure of social welfare in the market for a single good.

7.1.2 Topics and infrastructure

Key infrastructure concepts that we will encounter in this lecture are:

- inverse demand curves

- inverse supply curves

- consumer surplus

- producer surplus

- integration

- social welfare as the sum of consumer and producer surpluses

- the relationship between equilibrium quantity and social welfare optimum

In our exposition we will use the following Python imports.

```python
import numpy as np
import matplotlib.pyplot as plt
from collections import namedtuple
```

7.2 Consumer surplus

Before we look at the model of supply and demand, it will be helpful to have some background on (a) consumer and producer surpluses and (b) integration.

(If you are comfortable with both topics you can jump to the *next section*.)

7.2.1 A discrete example

Example 7.2.1

Regarding consumer surplus, suppose that we have a single good and 10 consumers.

These 10 consumers have different preferences; in particular, the amount they would be willing to pay for one unit of the good differs.

Suppose that the willingness to pay for each of the 10 consumers is as follows:

consumer	1	2	3	4	5	6	7	8	9	10
willing to pay	98	72	41	38	29	21	17	12	11	10

(We have ordered consumers by willingness to pay, in descending order.)

If p is the price of the good and w_i is the amount that consumer i is willing to pay, then i buys when $w_i \geq p$.

Note: If $p = w_i$ the consumer is indifferent between buying and not buying; we arbitrarily assume that they buy.

The **consumer surplus** of the i-th consumer is $\max\{w_i - p, 0\}$

- if $w_i \geq p$, then the consumer buys and gets surplus $w_i - p$

- if $w_i < p$, then the consumer does not buy and gets surplus 0

For example, if the price is $p = 40$, then consumer 1 gets surplus $98 - 40 = 58$.

The bar graph below shows the surplus of each consumer when $p = 25$.

The total height of each bar i is willingness to pay by consumer i.

The orange portion of some of the bars shows consumer surplus.

```
fig, ax = plt.subplots()
consumers = range(1, 11) # consumers 1,..., 10
# willingness to pay for each consumer
wtp = (98, 72, 41, 38, 29, 21, 17, 12, 11, 10)
price = 25
ax.bar(consumers, wtp, label="consumer surplus", color="darkorange", alpha=0.8)
ax.plot((0, 12), (price, price), lw=2, label="price $p$")
ax.bar(consumers, [min(w, price) for w in wtp], color="black", alpha=0.6)
ax.set_xlim(0, 12)
ax.set_xticks(consumers)
ax.set_ylabel("willingness to pay, price")
ax.set_xlabel("consumer, quantity")
ax.legend()
plt.show()
```

The total consumer surplus in this market is

$$\sum_{i=1}^{10} \max\{w_i - p, 0\} = \sum_{w_i \geq p} (w_i - p)$$

Since consumer surplus $\max\{w_i - p, 0\}$ of consumer i is a measure of her gains from trade (i.e., extent to which the good is valued over and above the amount the consumer had to pay), it is reasonable to consider total consumer surplus as a measurement of consumer welfare.
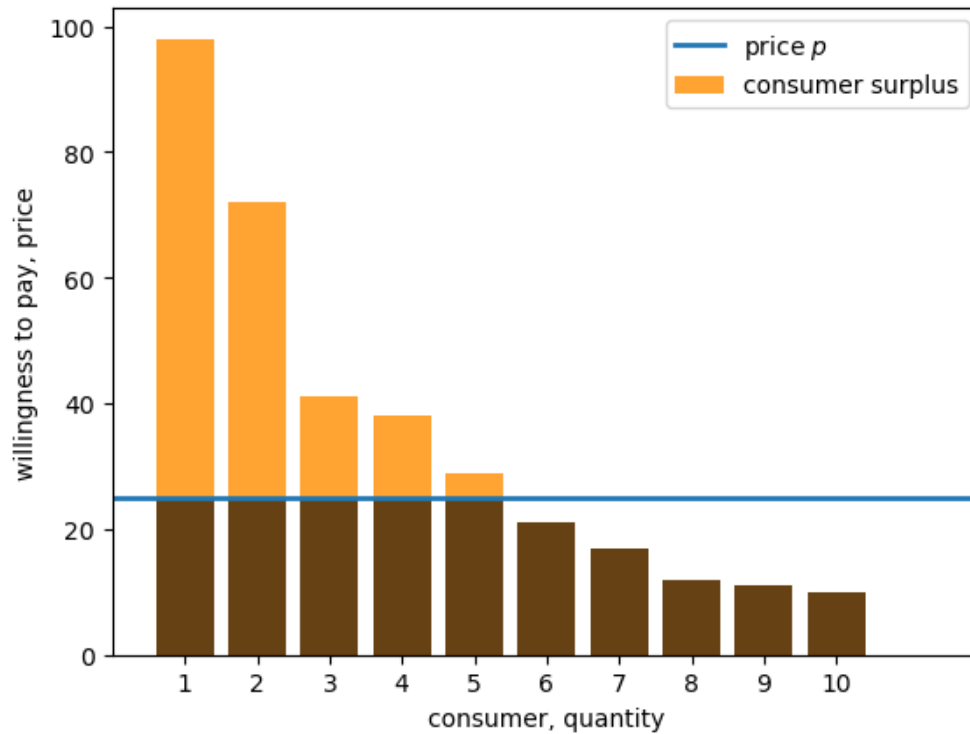
Fig. 7.1: Willingness to pay (discrete)

Later we will pursue this idea further, considering how different prices lead to different welfare outcomes for consumers and producers.

7.2.2 A comment on quantity.

Notice that in the figure, the horizontal axis is labeled "consumer, quantity".

We have added "quantity" here because we can read the number of units sold from this axis, assuming for now that there are sellers who are willing to sell as many units as the consumers demand, given the current market price p.

In this example, consumers 1 to 5 buy, and the quantity sold is 5.

Below we drop the assumption that sellers will provide any amount at a given price and study how this changes outcomes.

7.2.3 A continuous approximation

It is often convenient to assume that there is a "very large number" of consumers, so that willingness to pay becomes a continuous curve.

As before, the vertical axis measures willingness to pay, while the horizontal axis measures quantity.

This kind of curve is called an **inverse demand curve**

An example is provided below, showing both an inverse demand curve and a set price.

The inverse demand curve is given by

$$p = 100e^{-q}$$

```python
def inverse_demand(q):
    return 100 * np.exp(- q)

# build a grid to evaluate the function at different values of q
q_min, q_max = 0, 5
q_grid = np.linspace(q_min, q_max, 1000)

# plot the inverse demand curve
fig, ax = plt.subplots()
ax.plot((q_min, q_max), (price, price), lw=2, label="price")
ax.plot(q_grid, inverse_demand(q_grid),
        color="orange", label="inverse demand curve")
ax.set_ylabel("willingness to pay, price")
ax.set_xlabel("quantity")
ax.set_xlim(q_min, q_max)
ax.set_ylim(0, 110)
ax.legend()
plt.show()
```

Reasoning by analogy with the discrete case, the area under the demand curve and above the price is called the **consumer surplus**, and is a measure of total gains from trade on the part of consumers.

The consumer surplus is shaded in the figure below.

```python
# solve for the value of q where demand meets price
q_star = np.log(100) - np.log(price)

fig, ax = plt.subplots()
ax.plot((q_min, q_max), (price, price), lw=2, label="price")
```
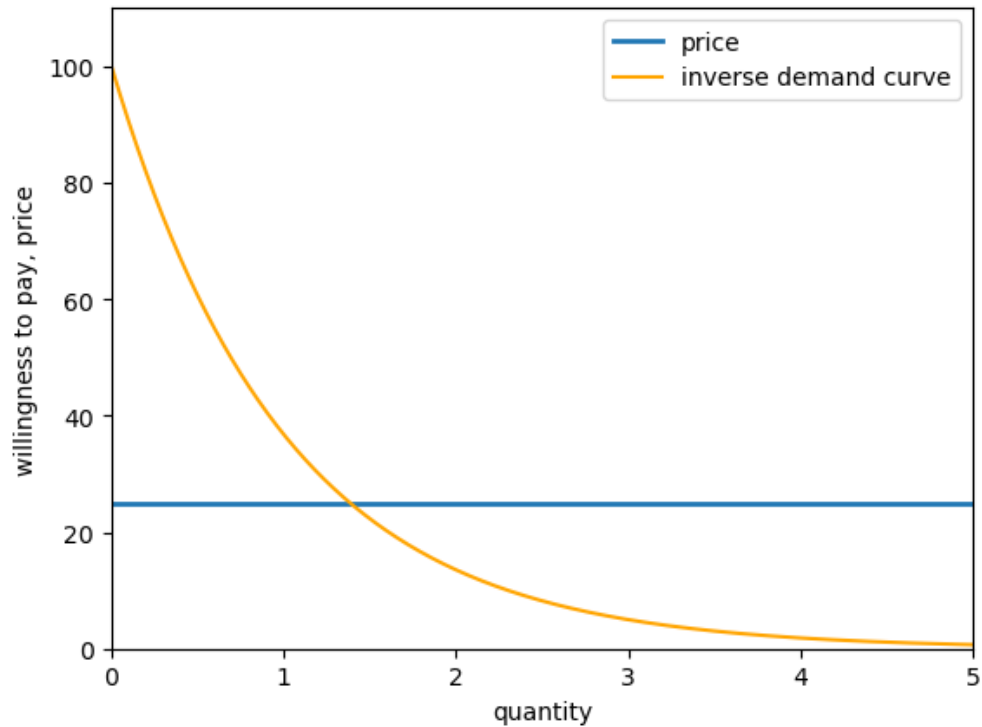
Fig. 7.2: Willingness to pay (continuous)

```python
ax.plot(q_grid, inverse_demand(q_grid),
        color="orange", label="inverse demand curve")
small_grid = np.linspace(0, q_star, 500)
ax.fill_between(small_grid, np.full(len(small_grid), price),
                inverse_demand(small_grid), color="orange",
                alpha=0.5, label="consumer surplus")
ax.vlines(q_star, 0, price, ls="--")
ax.set_ylabel("willingness to pay, price")
ax.set_xlabel("quantity")
ax.set_xlim(q_min, q_max)
ax.set_ylim(0, 110)
ax.text(q_star, -10, "$q^*$")
ax.legend()
plt.show()
```

The value q^* is where the inverse demand curve meets price.
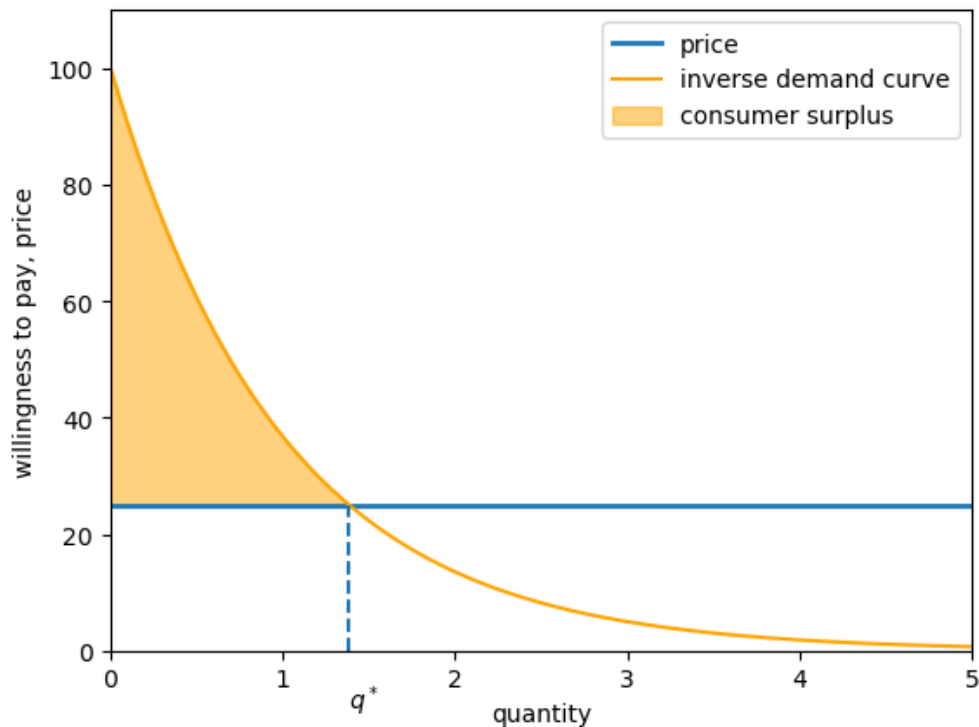
Fig. 7.3: Willingness to pay (continuous) with consumer surplus

7.3 Producer surplus

Having discussed demand, let's now switch over to the supply side of the market.

7.3.1 The discrete case

The figure below shows the price at which a collection of producers, also numbered 1 to 10, are willing to sell one unit of the good in question

```
fig, ax = plt.subplots()
producers = range(1, 11) # producers 1,..., 10
# willingness to sell for each producer
wts = (5, 8, 17, 22, 35, 39, 46, 57, 88, 91)
price = 25
ax.bar(producers, wts, label="willingness to sell", color="green", alpha=0.5)
ax.set_xlim(0, 12)
ax.set_xticks(producers)
ax.set_ylabel("willingness to sell")
ax.set_xlabel("producer")
```

```
ax.legend()
plt.show()
```
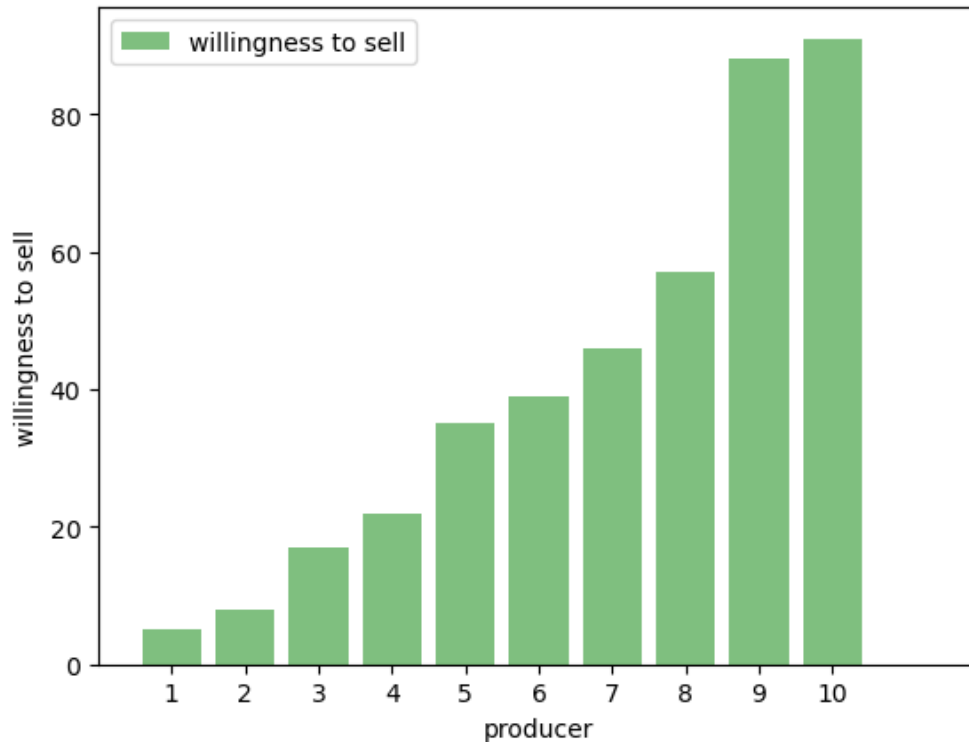

Fig. 7.4: Willingness to sell (discrete)

Let v_i be the price at which producer i is willing to sell the good.

When the price is p, producer surplus for producer i is $\max\{p - v_i, 0\}$.

Example 7.3.1

For example, a producer willing to sell at \$10 and selling at price \$20 makes a surplus of \$10.

Total producer surplus is given by

$$\sum_{i=1}^{10} \max\{p - v_i, 0\} = \sum_{p \geq v_i} (p - v_i)$$

As for the consumer case, it can be helpful for analysis if we approximate producer willingness to sell into a continuous curve.

This curve is called the **inverse supply curve**

We show an example below where the inverse supply curve is

$$p = 2q^2$$

The shaded area is the total producer surplus in this continuous model.

```python
def inverse_supply(q):
    return 2 * q**2

# solve for the value of q where supply meets price
q_star = (price / 2)**(1/2)

# plot the inverse supply curve
fig, ax = plt.subplots()
ax.plot((q_min, q_max), (price, price), lw=2, label="price")
ax.plot(q_grid, inverse_supply(q_grid),
        color="green", label="inverse supply curve")
small_grid = np.linspace(0, q_star, 500)
ax.fill_between(small_grid, inverse_supply(small_grid),
                np.full(len(small_grid), price),
                color="green",
                alpha=0.5, label="producer surplus")
ax.vlines(q_star, 0, price, ls="--")
ax.set_ylabel("willingness to sell, price")
ax.set_xlabel("quantity")
ax.set_xlim(q_min, q_max)
ax.set_ylim(0, 60)
ax.text(q_star, -10, "$q^*$")
ax.legend()
plt.show()
```

7.4 Integration

How can we calculate the consumer and producer surplus in the continuous case?

The short answer is: by using integration.

Some readers will already be familiar with the basics of integration.

For those who are not, here is a quick introduction.

In general, for a function f, the **integral** of f over the interval $[a, b]$ is the area under the curve f between a and b.
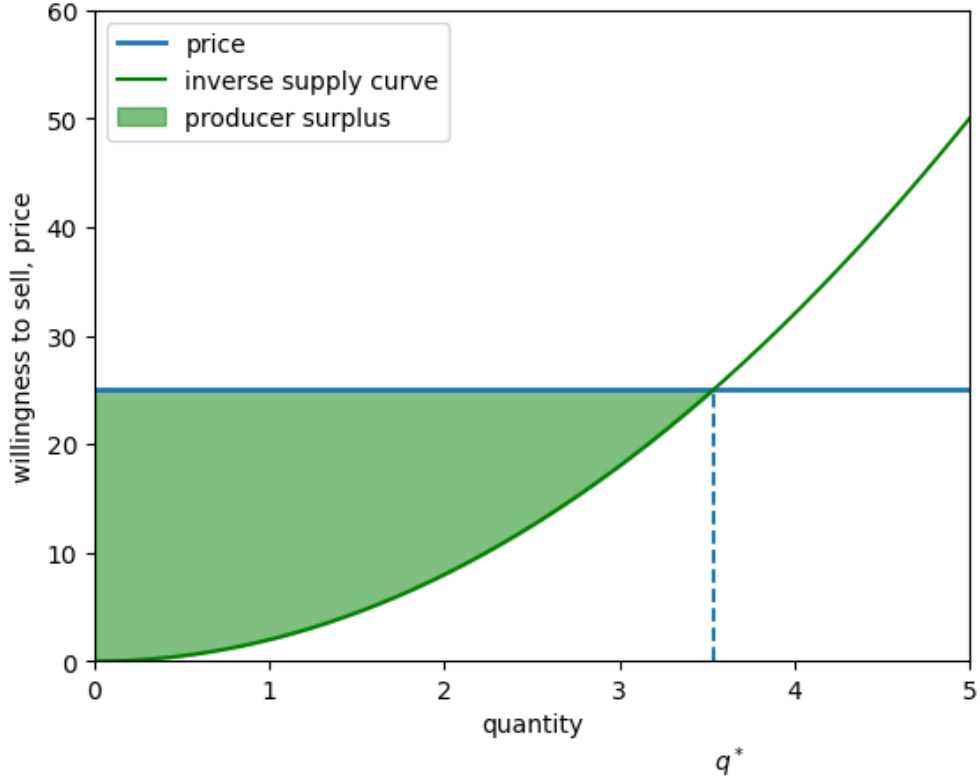
Fig. 7.5: Willingness to sell (continuous) with producer surplus

This value is written as $\int_a^b f(x)\mathrm{d}x$ and illustrated in the figure below when $f(x) = \cos(x/2) + 1$.

```python
def f(x):
    return np.cos(x/2) + 1

xmin, xmax = 0, 5
a, b = 1, 3
x_grid = np.linspace(xmin, xmax, 1000)
ab_grid = np.linspace(a, b, 400)

fig, ax = plt.subplots()
ax.plot(x_grid, f(x_grid), label="$f$", color="k")
ax.fill_between(ab_grid, [0] * len(ab_grid), f(ab_grid),
                label="$\int_a^b f(x) dx$")
ax.legend()
plt.show()
```
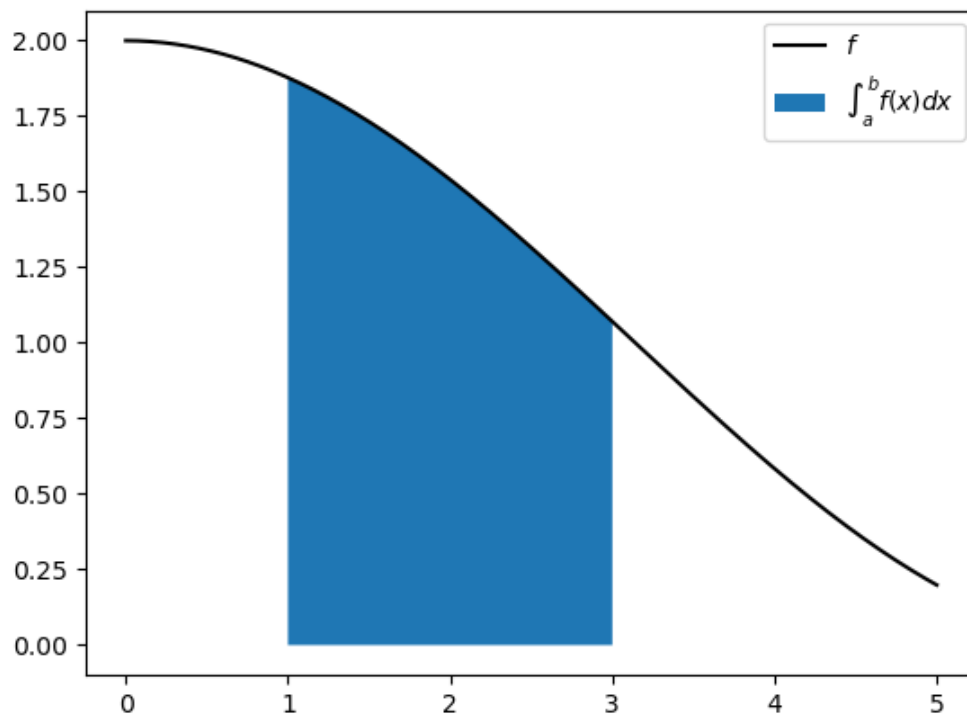

Fig. 7.6: Area under the curve

There are many rules for calculating integrals, with different rules applying to different choices of f.

Many of these rules relate to one of the most beautiful and powerful results in all of mathematics: the fundamental theorem of calculus.

We will not try to cover these ideas here, partly because the subject is too big, and partly because you only need to know one rule for this lecture, stated below.

If $f(x) = c + dx$, then

$$\int_a^b f(x)\mathrm{d}x = c(b-a) + \frac{d}{2}(b^2 - a^2)$$

In fact this rule is so simple that it can be calculated from elementary geometry –you might like to try by graphing f and calculating the area under the curve between a and b.

We use this rule repeatedly in what follows.

7.5 Supply and demand

Let's now put supply and demand together.

This leads us to the all important notion of market equilibrium, and from there onto a discussion of equilibria and welfare.

For most of this discussion, we'll assume that inverse demand and supply curves are **affine** functions of quantity.

Note: "Affine"means "linear plus a constant"and here is a nice discussion about it.

We'll also assume affine inverse supply and demand functions when we study models with multiple consumption goods in our *subsequent lecture*.

We do this in order to simplify the exposition and enable us to use just a few tools from linear algebra, namely, matrix multiplication and matrix inversion.

We study a market for a single good in which buyers and sellers exchange a quantity q for a price p.

Quantity q and price p are both scalars.

We assume that inverse demand and supply curves for the good are:

$$p = d_0 - d_1 q, \quad d_0, d_1 > 0$$

$$p = s_0 + s_1 q, \quad s_0, s_1 > 0$$

We call them inverse demand and supply curves because price is on the left side of the equation rather than on the right side as it would be in a direct demand or supply function.

We can use a namedtuple to store the parameters for our single good market.

```
Market = namedtuple('Market', ['d_0', # demand intercept
                               'd_1', # demand slope
                               's_0', # supply intercept
                               's_1']) # supply slope
              )
```

The function below creates an instance of a Market namedtuple with default values.

```
def create_market(d_0=1.0, d_1=0.6, s_0=0.1, s_1=0.4):
    return Market(d_0=d_0, d_1=d_1, s_0=s_0, s_1=s_1)
```

This `market` can then be used by our `inverse_demand` and `inverse_supply` functions.

```
def inverse_demand(q, model):
    return model.d_0 - model.d_1 * q

def inverse_supply(q, model):
    return model.s_0 + model.s_1 * q
```

Here is a plot of these two functions using `market`.

```
market = create_market()

grid_min, grid_max, grid_size = 0, 1.5, 200
q_grid = np.linspace(grid_min, grid_max, grid_size)
supply_curve = inverse_supply(q_grid, market)
demand_curve = inverse_demand(q_grid, market)

fig, ax = plt.subplots()
ax.plot(q_grid, supply_curve, label='supply', color='green')
ax.plot(q_grid, demand_curve, label='demand', color='orange')
ax.legend(loc='upper center', frameon=False)
ax.set_ylim(0, 1.2)
ax.set_xticks((0, 1))
ax.set_yticks((0, 1))
ax.set_xlabel('quantity')
ax.set_ylabel('price')
plt.show()
```

In the above graph, an **equilibrium** price-quantity pair occurs at the intersection of the supply and demand curves.
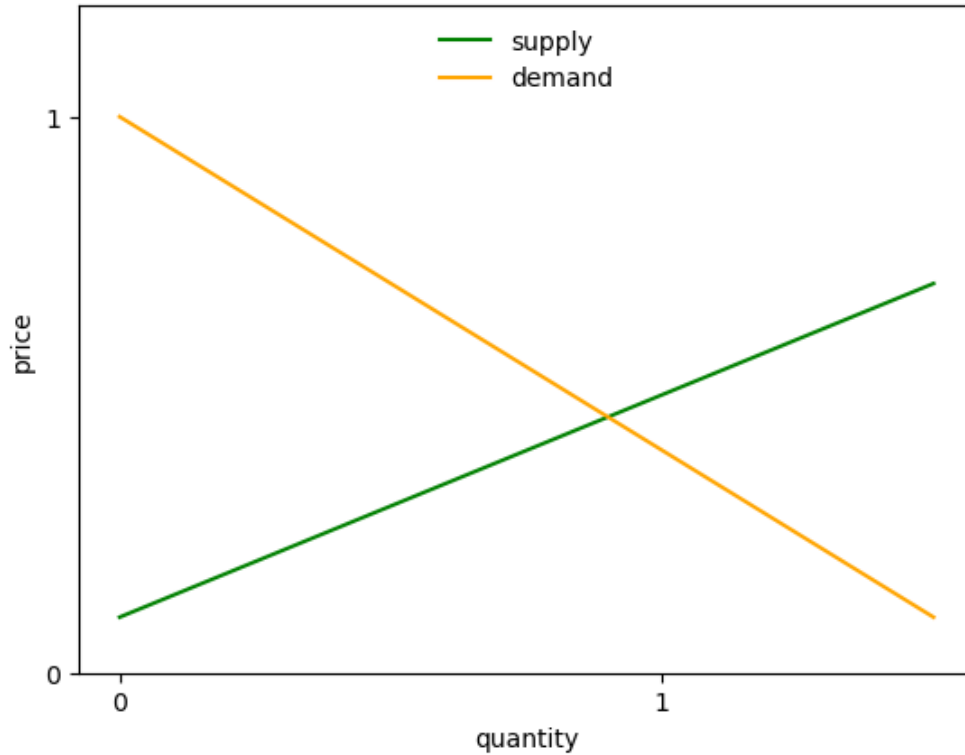
Fig. 7.7: Supply and demand

7.5.1 Consumer surplus

Let a quantity q be given and let $p := d_0 - d_1 q$ be the corresponding price on the inverse demand curve.

We define **consumer surplus** $S_c(q)$ as the area under an inverse demand curve minus pq:

$$S_c(q) := \int_0^q (d_0 - d_1 x)\mathrm{d}x - pq \tag{7.1}$$

The next figure illustrates

Consumer surplus provides a measure of total consumer welfare at quantity q.

The idea is that the inverse demand curve $d_0 - d_1 q$ shows a consumer's willingness to pay for an additional increment of the good at a given quantity q.

The difference between willingness to pay and the actual price is consumer surplus.

The value $S_c(q)$ is the "sum"(i.e., integral) of these surpluses when the total quantity purchased is q and the purchase price is p.

Evaluating the integral in the definition of consumer surplus (7.1) gives

$$S_c(q) = d_0 q - \frac{1}{2} d_1 q^2 - pq$$
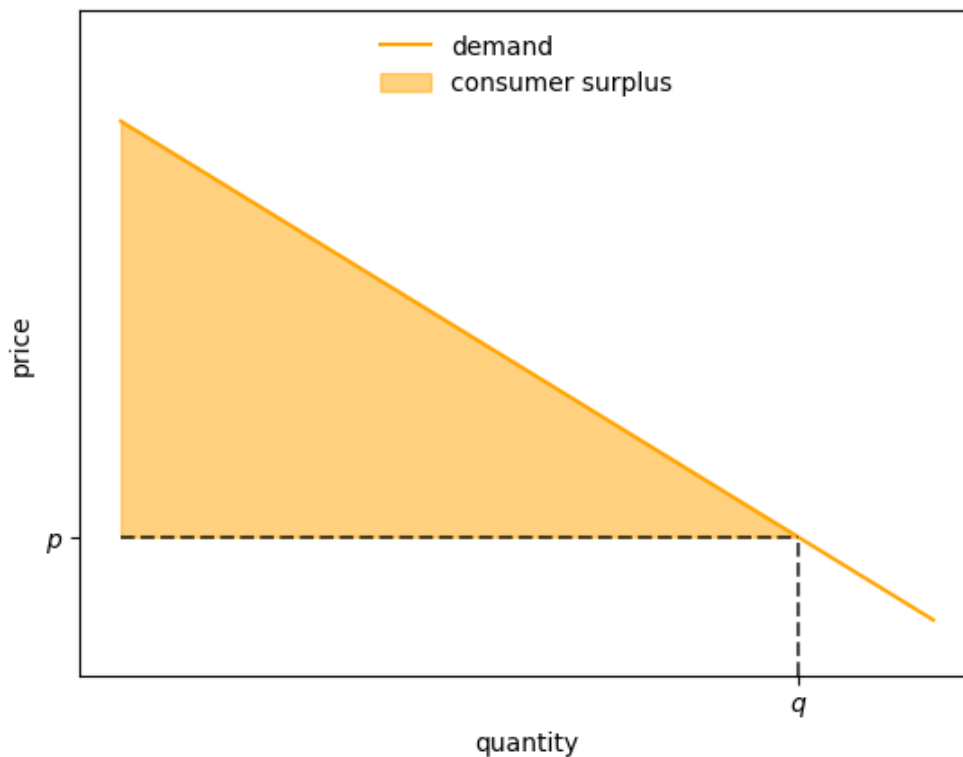
Fig. 7.8: Supply and demand (consumer surplus)

7.5.2 Producer surplus

Let a quantity q be given and let $p := s_0 + s_1 q$ be the corresponding price on the inverse supply curve.

We define **producer surplus** as pq minus the area under an inverse supply curve

$$S_p(q) := pq - \int_0^q (s_0 + s_1 x) \mathrm{d}x \tag{7.2}$$

The next figure illustrates

Producer surplus measures total producer welfare at quantity q

The idea is similar to that of consumer surplus.

The inverse supply curve $s_0 + s_1 q$ shows the price at which producers are prepared to sell, given quantity q.

The difference between willingness to sell and the actual price is producer surplus.

The value $S_p(q)$ is the integral of these surpluses.

Evaluating the integral in the definition of producer surplus (7.2) gives

$$S_p(q) = pq - s_0 q - \frac{1}{2} s_1 q^2$$
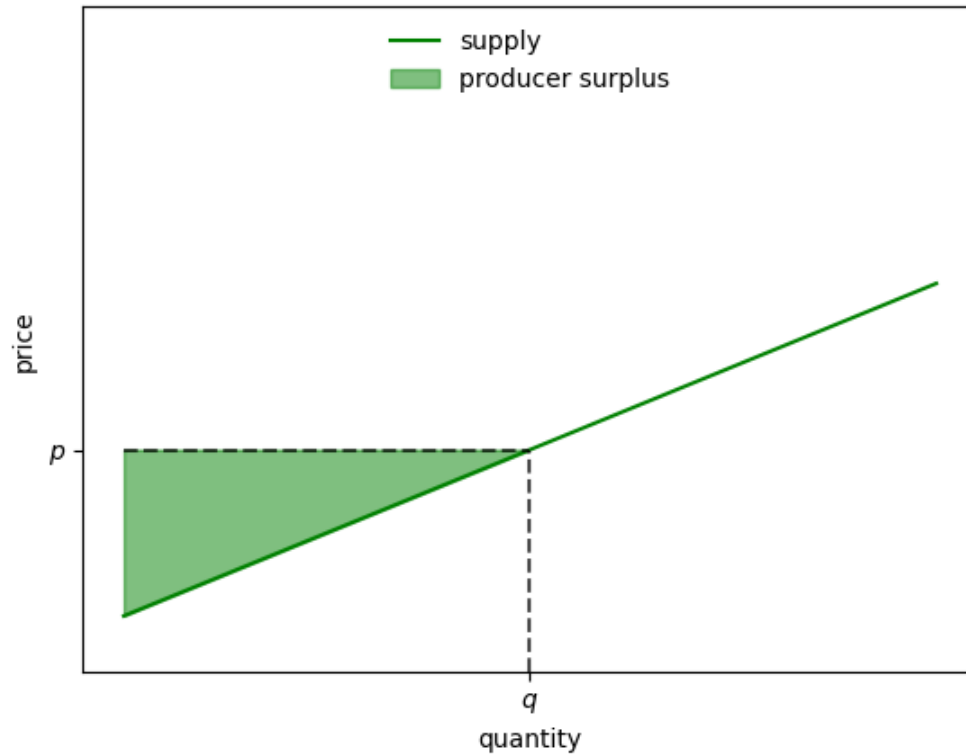
Fig. 7.9: Supply and demand (producer surplus)

7.5.3 Social welfare

Sometimes economists measure social welfare by a **welfare criterion** that equals consumer surplus plus producer surplus, assuming that consumers and producers pay the same price:

$$W(q) = \int_0^q (d_0 - d_1 x) dx - \int_0^q (s_0 + s_1 x) \mathrm{d}x$$

Evaluating the integrals gives

$$W(q) = (d_0 - s_0)q - \frac{1}{2}(d_1 + s_1)q^2$$

Here is a Python function that evaluates this social welfare at a given quantity q and a fixed set of parameters.

```python
def W(q, market):
    # Compute and return welfare
    return (market.d_0 - market.s_0) * q - 0.5 * (market.d_1 + market.s_1) * q**2
```

The next figure plots welfare as a function of q.

Let's now give a social planner the task of maximizing social welfare.
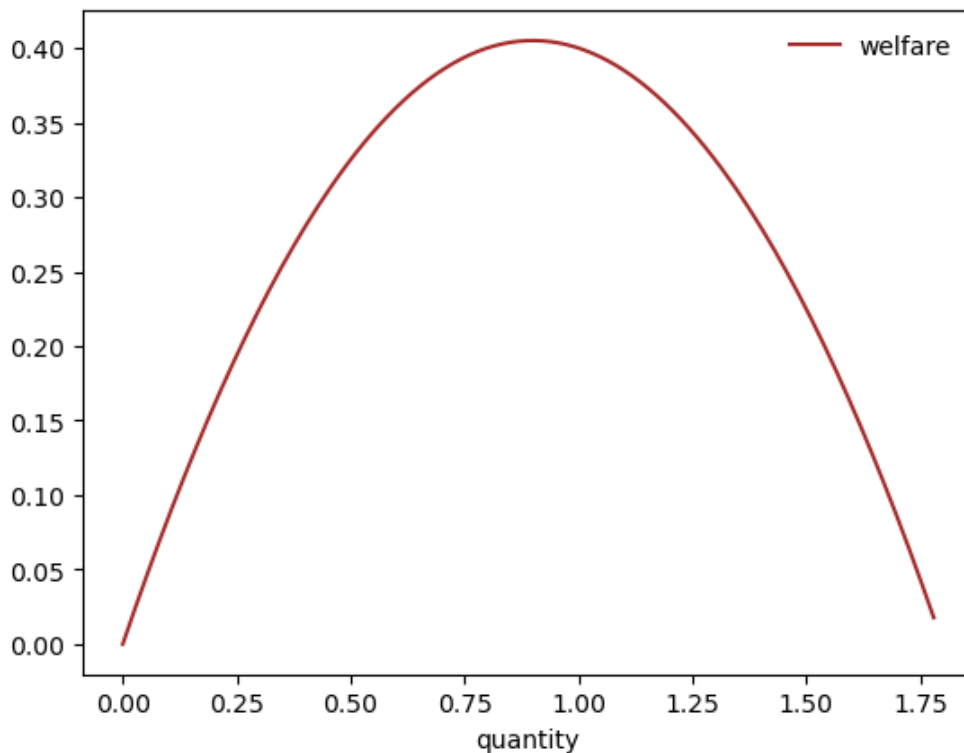
Fig. 7.10: Welfare

To compute a quantity that maximizes the welfare criterion, we differentiate W with respect to q and then set the derivative to zero.

$$\frac{\mathrm{d}W(q)}{\mathrm{d}q} = d_0 - s_0 - (d_1 + s_1)q = 0$$

Solving for q yields

$$q = \frac{d_0 - s_0}{s_1 + d_1} \tag{7.3}$$

Let's remember the quantity q given by equation (7.3) that a social planner would choose to maximize consumer surplus plus producer surplus.

We'll compare it to the quantity that emerges in a competitive equilibrium that equates supply to demand.

7.5.4 Competitive equilibrium

Instead of equating quantities supplied and demanded, we can accomplish the same thing by equating demand price to supply price:

$$p = d_0 - d_1 q = s_0 + s_1 q$$

If we solve the equation defined by the second equality in the above line for q, we obtain

$$q = \frac{d_0 - s_0}{s_1 + d_1} \tag{7.4}$$

This is the competitive equilibrium quantity.

Observe that the equilibrium quantity equals the same q given by equation (7.3).

The outcome that the quantity determined by equation (7.3) equates supply to demand brings us a *key finding*:

- a competitive equilibrium quantity maximizes our welfare criterion

This is a version of the first fundamental welfare theorem,

It also brings a useful **competitive equilibrium computation strategy:**

- after solving the welfare problem for an optimal quantity, we can read a competitive equilibrium price from either supply price or demand price at the competitive equilibrium quantity

7.6 Generalizations

In a *later lecture*, we'll derive generalizations of the above demand and supply curves from other objects.

Our generalizations will extend the preceding analysis of a market for a single good to the analysis of n simultaneous markets in n goods.

In addition

- we'll derive *demand curves* from a consumer problem that maximizes a *utility function* subject to a *budget constraint*.

- we'll derive *supply curves* from the problem of a producer who is price taker and maximizes his profits minus total costs that are described by a *cost function*.

7.7 Exercises

Suppose now that the inverse demand and supply curves are modified to take the form

$$p = i_d(q) := d_0 - d_1 q^{0.6}$$

$$p = i_s(q) := s_0 + s_1 q^{1.8}$$

All parameters are positive, as before.

Exercise 7.7.1

Use the same `Market` namedtuple that holds the parameter values as before but make new `inverse_demand` and `inverse_supply` functions to match these new definitions.

Then plot the inverse demand and supply curves i_d and i_s.

Solution to Exercise 7.7.1

Let's update the `inverse_demand` and `inverse_supply` functions, as defined above.

```python
def inverse_demand(q, model):
    return model.d_0 - model.d_1 * q**0.6


def inverse_supply(q, model):
    return model.s_0 + model.s_1 * q**1.8
```

Here is a plot of inverse supply and demand.

```python
grid_min, grid_max, grid_size = 0, 1.5, 200
q_grid = np.linspace(grid_min, grid_max, grid_size)
market = create_market()
supply_curve = inverse_supply(q_grid, market)
demand_curve = inverse_demand(q_grid, market)

fig, ax = plt.subplots()
ax.plot(q_grid, supply_curve, label='supply', color='green')
ax.plot(q_grid, demand_curve, label='demand', color='orange')
ax.legend(loc='upper center', frameon=False)
ax.set_ylim(0, 1.2)
ax.set_xticks((0, 1))
ax.set_yticks((0, 1))
ax.set_xlabel('quantity')
ax.set_ylabel('price')
plt.show()
```
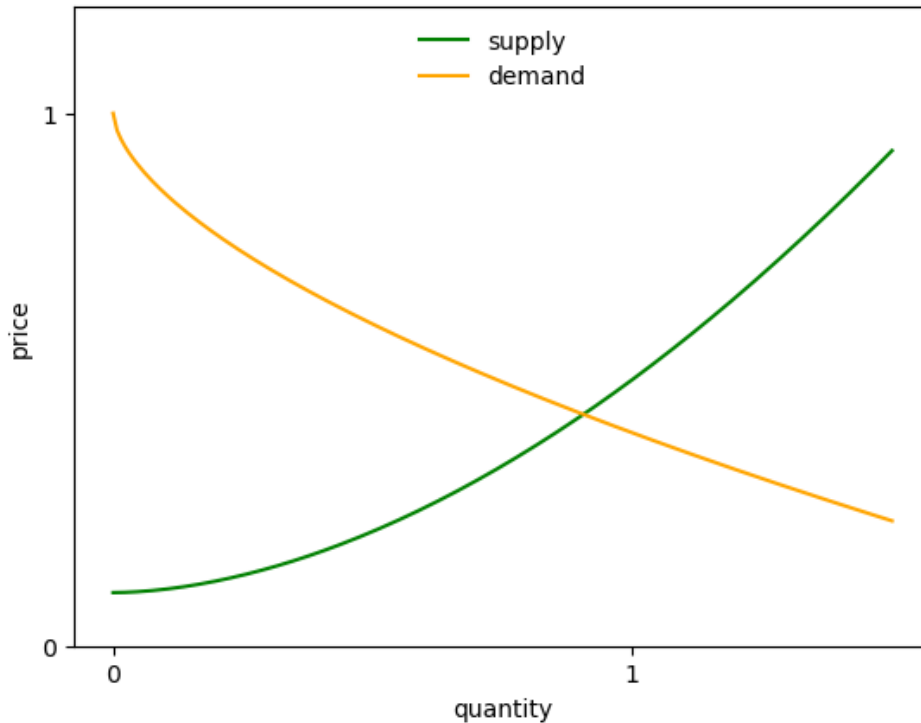
Exercise 7.7.2

As before, consumer surplus at q is the area under the demand curve minus price times quantity:

$$S_c(q) = \int_0^q i_d(x)dx - pq$$

Here p is set to $i_d(q)$

Producer surplus is price times quantity minus the area under the inverse supply curve:

$$S_p(q) = pq - \int_0^q i_s(x)\mathrm{d}x$$

Here p is set to $i_s(q)$.

Social welfare is the sum of consumer and producer surplus under the assumption that the price is the same for buyers and sellers:

$$W(q) = \int_0^q i_d(x)dx - \int_0^q i_s(x)\mathrm{d}x$$

Solve the integrals and write a function to compute this quantity numerically at given q.

Plot welfare as a function of q.

Solution to Exercise 7.7.2

Solving the integrals gives

$$W(q) = d_0 q - \frac{d_1 q^{1.6}}{1.6} - \left(s_0 q + \frac{s_1 q^{2.8}}{2.8} \right)$$

Here's a Python function that computes this value:

```python
def W(q, market):
    # Compute and return welfare
    S_c = market.d_0 * q - market.d_1 * q**1.6 / 1.6
    S_p = market.s_0 * q + market.s_1 * q**2.8 / 2.8
    return S_c - S_p
```

The next figure plots welfare as a function of q.

```python
fig, ax = plt.subplots()
ax.plot(q_vals, W(q_vals, market), label='welfare', color='brown')
ax.legend(frameon=False)
ax.set_xlabel('quantity')
plt.show()
```
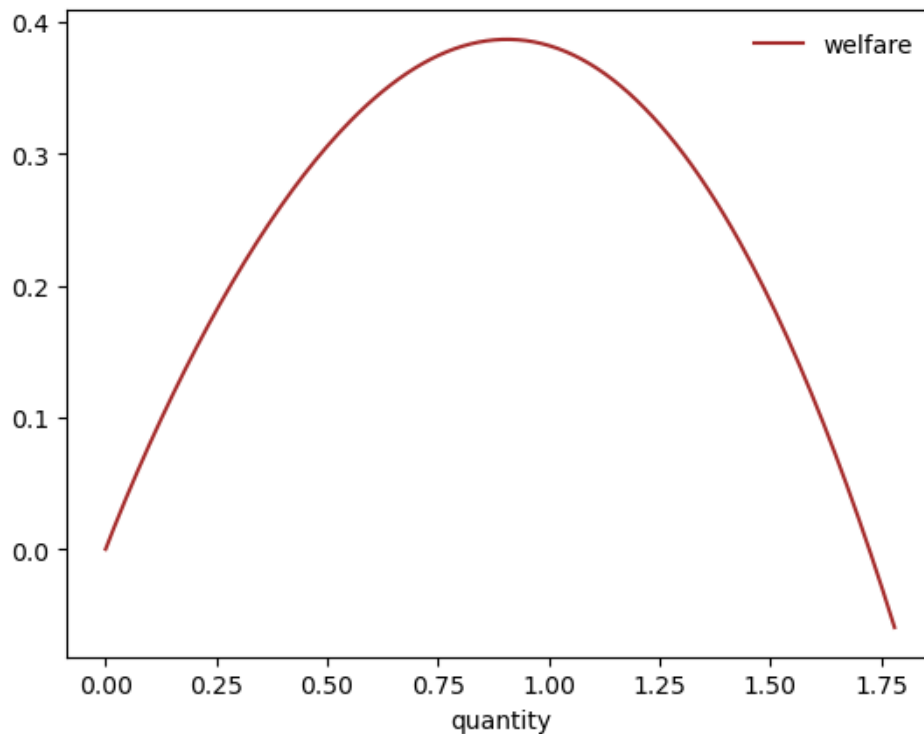

Exercise 7.7.3

Due to non-linearities, the new welfare function is not easy to maximize with pencil and paper.

Maximize it using `scipy.optimize.minimize_scalar` instead.

See also:

Our SciPy lecture has a section on Optimization is a useful resource to find out more.

Solution to Exercise 7.7.3

```python
from scipy.optimize import minimize_scalar


def objective(q):
    return -W(q, market)


result = minimize_scalar(objective, bounds=(0, 10))
print(result.message)
```

```
Solution found.
```

```python
maximizing_q = result.x
print(f"{maximizing_q: .5f}")
```

```
0.90564
```

Exercise 7.7.4

Now compute the equilibrium quantity by finding the price that equates supply and demand.

You can do this numerically by finding the root of the excess demand function

$$e_d(q) := i_d(q) - i_s(q)$$

You can use `scipy.optimize.newton` to compute the root.

See also:

Our SciPy lecture has a section on Roots and Fixed Points is a useful resource to find out more.

Initialize `newton` with a starting guess somewhere close to 1.0.

(Similar initial conditions will give the same result.)

You should find that the equilibrium price agrees with the welfare maximizing price, in line with the first fundamental welfare theorem.

Solution to Exercise 7.7.4

```python
from scipy.optimize import newton

def excess_demand(q):
    return inverse_demand(q, market) - inverse_supply(q, market)

equilibrium_q = newton(excess_demand, 0.99)
print(f"{equilibrium_q: .5f}")
```

```
0.90564
```

LINEAR EQUATIONS AND MATRIX ALGEBRA

8.1 Overview

Many problems in economics and finance require solving linear equations.

In this lecture we discuss linear equations and their applications.

To illustrate the importance of linear equations, we begin with a two good model of supply and demand.

The two good case is so simple that solutions can be calculated by hand.

But often we need to consider markets containing many goods.

In the multiple goods case we face large systems of linear equations, with many equations and unknowns.

To handle such systems we need two things:

- matrix algebra (and the knowledge of how to use it) plus

- computer code to apply matrix algebra to the problems of interest.

This lecture covers these steps.

We will use the following packages:

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

8.2 A two good example

In this section we discuss a simple two good example and solve it by

1. pencil and paper

2. matrix algebra

The second method is more general, as we will see.

8.2.1 Pencil and paper methods

Suppose that we have two related goods, such as

- propane and ethanol, and

- rice and wheat, etc.

To keep things simple, we label them as good 0 and good 1.

The demand for each good depends on the price of both goods:

$$
\begin{aligned}
q_0^d &= 100 - 10p_0 - 5p_1 \\
q_1^d &= 50 - p_0 - 10p_1
\end{aligned}
\tag{8.1}
$$

(We are assuming demand decreases when the price of either good goes up, but other cases are also possible.)

Let's suppose that supply is given by

$$
\begin{aligned}
q_0^s &= 10p_0 + 5p_1 \\
q_1^s &= 5p_0 + 10p_1
\end{aligned}
\tag{8.2}
$$

Equilibrium holds when supply equals demand ($q_0^s = q_0^d$ and $q_1^s = q_1^d$).

This yields the linear system

$$
\begin{aligned}
100 - 10p_0 - 5p_1 &= 10p_0 + 5p_1 \\
50 - p_0 - 10p_1 &= 5p_0 + 10p_1
\end{aligned}
\tag{8.3}
$$

We can solve this with pencil and paper to get

$$
p_0 = 4.41 \quad \text{and} \quad p_1 = 1.18.
$$

Inserting these results into either (8.1) or (8.2) yields the equilibrium quantities

$$
q_0 = 50 \quad \text{and} \quad q_1 = 33.82.
$$

8.2.2 Looking forward

Pencil and paper methods are easy in the two good case.

But what if there are many goods?

For such problems we need matrix algebra.

Before solving problems with matrix algebra, let's first recall the basics of vectors and matrices, in both theory and computation.

8.3 Vectors

A **vector** of length n is just a sequence (or array, or tuple) of n numbers, which we write as $x = (x_1, \dots, x_n)$ or $x = \begin{bmatrix} x_1, \dots, x_n \end{bmatrix}$.

We can write these sequences either horizontally or vertically.

But when we use matrix operations, our default assumption is that vectors are column vectors.

The set of all n-vectors is denoted by $\mathbb{R}^n$.

Example 8.3.1

- $\mathbb{R}^2$ is the plane —the set of pairs (x_1, x_2).

- $\mathbb{R}^3$ is 3 dimensional space —the set of vectors (x_1, x_2, x_3).

Often vectors are represented visually as arrows from the origin to the point.

Here's a visualization.
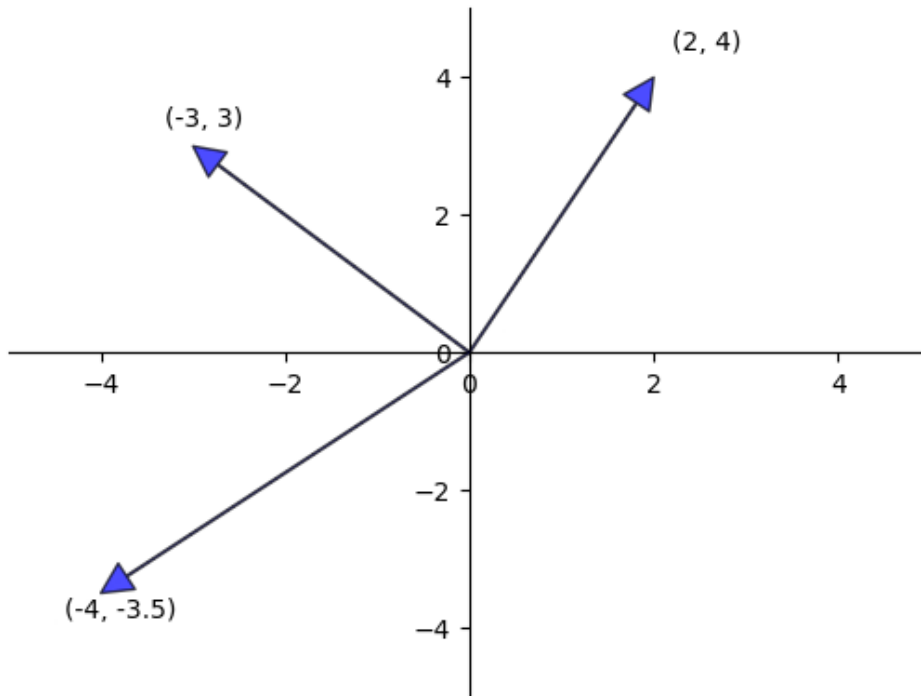
8.3.1 Vector operations

Sometimes we want to modify vectors.

The two most common operators on vectors are addition and scalar multiplication, which we now describe.

When we add two vectors, we add them element-by-element.

Example 8.3.2

$$\begin{bmatrix} 4 \\ -2 \end{bmatrix} + \begin{bmatrix} 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 4 & + & 3 \\ -2 & + & 3 \end{bmatrix} = \begin{bmatrix} 7 \\ 1 \end{bmatrix}.$$

In general,

$$x + y = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} := \begin{bmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{bmatrix}.$$

We can visualise vector addition in $\mathbb{R}^2$ as follows.
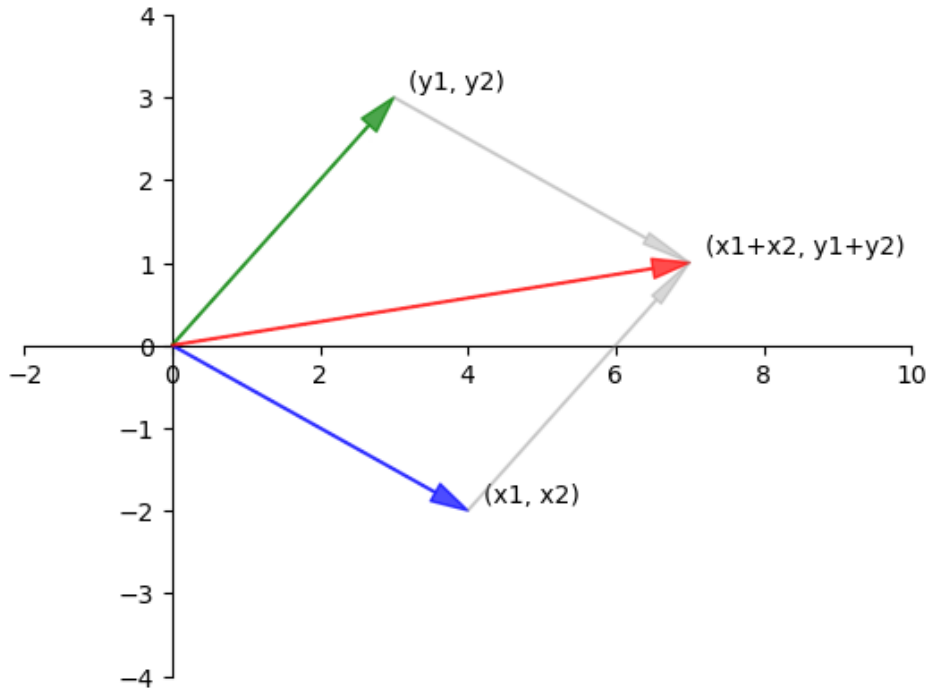
Scalar multiplication is an operation that multiplies a vector x with a scalar elementwise.

Example 8.3.3

$$-2 \begin{bmatrix} 3 \\ -7 \end{bmatrix} = \begin{bmatrix} -2 & \times & 3 \\ -2 & \times & -7 \end{bmatrix} = \begin{bmatrix} -6 \\ 14 \end{bmatrix}.$$

More generally, it takes a number γ and a vector x and produces

$$\gamma x := \begin{bmatrix} \gamma x_1 \\ \gamma x_2 \\ \vdots \\ \gamma x_n \end{bmatrix}.$$

Scalar multiplication is illustrated in the next figure.
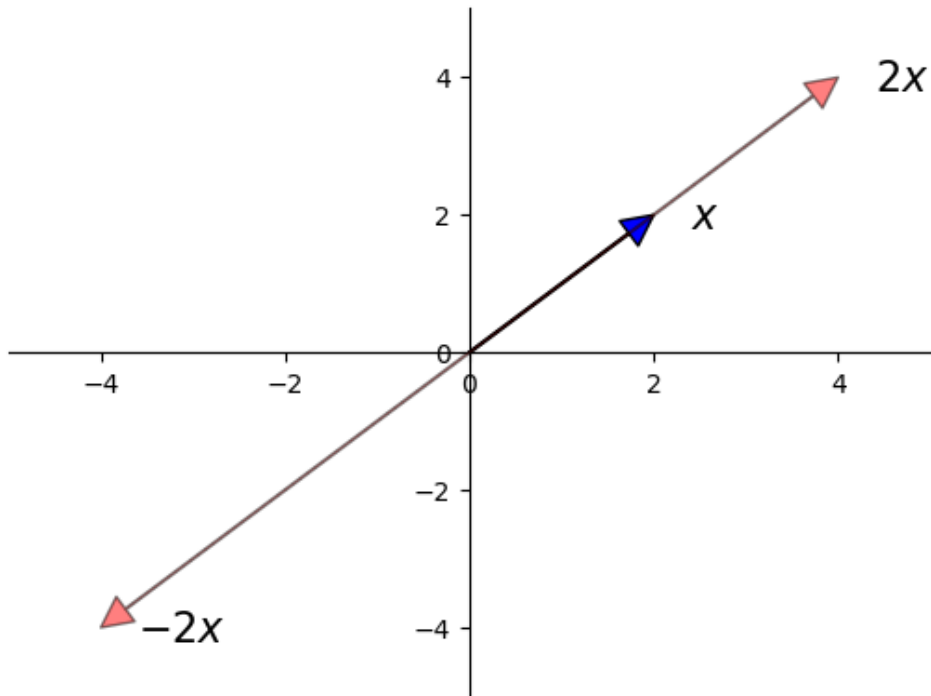
In Python, a vector can be represented as a list or tuple, such as `x = [2, 4, 6]` or `x = (2, 4, 6)`.

However, it is more common to represent vectors with NumPy arrays.

One advantage of NumPy arrays is that scalar multiplication and addition have very natural syntax.

```
x = np.ones(3)              # Vector of three ones
y = np.array((2, 4, 6))     # Converts tuple (2, 4, 6) into a NumPy array
x + y                       # Add (element-by-element)
```

```
array([3., 5., 7.])
```

```
4 * x                       # Scalar multiply
```

```
array([4., 4., 4.])
```

8.3.2 Inner product and norm

The **inner product** of vectors $x, y \in \mathbb{R}^n$ is defined as

$$
x^\top y = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n := \sum_{i=1}^{n} x_i y_i.
$$

The **norm** of a vector x represents its "length"(i.e., its distance from the zero vector) and is defined as

$$
\|x\| := \sqrt{x^\top x} := \left(\sum_{i=1}^{n} x_i^2 \right)^{1/2}.
$$

The expression $\|x - y\|$ can be thought of as the "distance"between x and y.

The inner product and norm can be computed as follows

```python
np.sum(x*y)          # Inner product of x and y
```

```
12.0
```

```python
x @ y                # Another way to compute the inner product
```

```
12.0
```

```python
np.sqrt(np.sum(x**2))   # Norm of x, method one
```

```
1.7320508075688772
```

```python
np.linalg.norm(x)       # Norm of x, method two
```

```
1.7320508075688772
```

8.4 Matrix operations

When we discussed linear price systems, we mentioned using matrix algebra.

Matrix algebra is similar to algebra for numbers.

Let's review some details.

8.4.1 Addition and scalar multiplication

Just as was the case for vectors, we can add, subtract and scalar multiply matrices.

Scalar multiplication and addition are generalizations of the vector case:

Example 8.4.1

$$3 \begin{bmatrix} 2 & -13 \\ 0 & 5 \end{bmatrix} = \begin{bmatrix} 6 & -39 \\ 0 & 15 \end{bmatrix}.$$

In general for a number γ and any matrix A,

$$\gamma A = \gamma \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} := \begin{bmatrix} \gamma a_{11} & \cdots & \gamma a_{1k} \\ \vdots & \vdots & \vdots \\ \gamma a_{n1} & \cdots & \gamma a_{nk} \end{bmatrix}.$$

Example 8.4.2

Consider this example of matrix addition,

$$\begin{bmatrix} 1 & 5 \\ 7 & 3 \end{bmatrix} + \begin{bmatrix} 12 & -1 \\ 0 & 9 \end{bmatrix} = \begin{bmatrix} 13 & 4 \\ 7 & 12 \end{bmatrix}.$$

In general,

$$A + B = \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{bmatrix} + \begin{bmatrix} b_{11} & \cdots & b_{1k} \\ \vdots & \vdots & \vdots \\ b_{n1} & \cdots & b_{nk} \end{bmatrix} := \begin{bmatrix} a_{11} + b_{11} & \cdots & a_{1k} + b_{1k} \\ \vdots & \vdots & \vdots \\ a_{n1} + b_{n1} & \cdots & a_{nk} + b_{nk} \end{bmatrix}.$$

In the latter case, the matrices must have the same shape in order for the definition to make sense.

8.4.2 Matrix multiplication

We also have a convention for *multiplying* two matrices.

The rule for matrix multiplication generalizes the idea of inner products discussed above.

If A and B are two matrices, then their product AB is formed by taking as its i, j-th element the inner product of the i-th row of A and the j-th column of B.

If A is $n \times k$ and B is $j \times m$, then to multiply A and B we require $k = j$, and the resulting matrix AB is $n \times m$.

Example 8.4.3

Here's an example of a 2×2 matrix multiplied by a 2×1 vector.

$$Ax = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 \\ a_{21}x_1 + a_{22}x_2 \end{bmatrix}$$

As an important special case, consider multiplying $n \times k$ matrix A and $k \times 1$ column vector x.

According to the preceding rule, this gives us an $n \times 1$ column vector.

$$Ax = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ \vdots & \vdots & & \vdots \\ a_{i1} & a_{i2} & \cdots & a_ik \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{bmatrix}_{n \times k} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_k \end{bmatrix}_{k \times 1} := \begin{bmatrix} a_{11}x_1 + a_{22}x_2 + \cdots + a_{1k}x_k \\ \vdots \\ a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{ik}x_k \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nk}x_k \end{bmatrix}_{n \times 1} \tag{8.4}$$

Here is a simple illustration of multiplication of two matrices.

$$AB = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} := \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

There are many tutorials to help you further visualize this operation, such as

- this one, or

- the discussion on the Wikipedia page.

Note: Unlike number products, AB and BA are not generally the same thing.

One important special case is the identity matrix, which has ones on the principal diagonal and zero elsewhere:

$$I = \begin{bmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{bmatrix}$$

It is a useful exercise to check the following:

- if A is $n \times k$ and I is the $k \times k$ identity matrix, then $AI = A$, and

- if I is the $n \times n$ identity matrix, then $IA = A$.

8.4.3 Matrices in NumPy

NumPy arrays are also used as matrices, and have fast, efficient functions and methods for all the standard matrix operations.

You can create them manually from tuples of tuples (or lists of lists) as follows

```
A = ((1, 2),
     (3, 4))

type(A)
```

```
tuple
```

```
A = np.array(A)

type(A)
```

```
numpy.ndarray
```

```
A.shape
```

```
(2, 2)
```

The `shape` attribute is a tuple giving the number of rows and columns —see here for more discussion.

To get the transpose of `A`, use `A.transpose()` or, more simply, `A.T`.

There are many convenient functions for creating common matrices (matrices of zeros, ones, etc.) —see here.

Since operations are performed elementwise by default, scalar multiplication and addition have very natural syntax.

```
A = np.identity(3)    # 3 x 3 identity matrix
B = np.ones((3, 3))   # 3 x 3 matrix of ones
2 * A
```

```
array([[2., 0., 0.],
       [0., 2., 0.],
       [0., 0., 2.]])
```

```
A + B
```

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

To multiply matrices we use the @ symbol.

Note: In particular, `A @ B` is matrix multiplication, whereas `A * B` is element-by-element multiplication.

8.4.4 Two good model in matrix form

We can now revisit the two good model and solve (8.3) numerically via matrix algebra.

This involves some extra steps but the method is widely applicable —as we will see when we include more goods.

First we rewrite (8.1) as

$$q^d = Dp + h \quad \text{where} \quad q^d = \begin{bmatrix} q_0^d \\ q_1^d \end{bmatrix} \quad D = \begin{bmatrix} -10 & -5 \\ -1 & -10 \end{bmatrix} \quad \text{and} \quad h = \begin{bmatrix} 100 \\ 50 \end{bmatrix}. \tag{8.5}$$

Recall that $p \in \mathbb{R}^2$ is the price of two goods.

(Please check that $q^d = Dp + h$ represents the same equations as (8.1).)

We rewrite (8.2) as

$$q^s = Cp \quad \text{where} \quad q^s = \begin{bmatrix} q_0^s \\ q_1^s \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} 10 & 5 \\ 5 & 10 \end{bmatrix}. \tag{8.6}$$

Now equality of supply and demand can be expressed as $q^s = q^d$, or

$$Cp = Dp + h.$$

We can rearrange the terms to get

$$(C - D)p = h.$$

If all of the terms were numbers, we could solve for prices as $p = h/(C - D)$.

Matrix algebra allows us to do something similar: we can solve for equilibrium prices using the inverse of $C - D$:

$$p = (C - D)^{-1}h. \tag{8.7}$$

Before we implement the solution let us consider a more general setting.

8.4. Matrix operations

8.4.5 More goods

It is natural to think about demand systems with more goods.

For example, even within energy commodities there are many different goods, including crude oil, gasoline, coal, natural gas, ethanol, and uranium.

The prices of these goods are related, so it makes sense to study them together.

Pencil and paper methods become very time consuming with large systems.

But fortunately the matrix methods described above are essentially unchanged.

In general, we can write the demand equation as $q^d = Dp + h$, where

- q^d is an $n \times 1$ vector of demand quantities for n different goods.

- D is an $n \times n$ "coefficient"matrix.

- h is an $n \times 1$ vector of constant values.

Similarly, we can write the supply equation as $q^s = Cp + e$, where

- q^s is an $n \times 1$ vector of supply quantities for the same goods.

- C is an $n \times n$ "coefficient"matrix.

- e is an $n \times 1$ vector of constant values.

To find an equilibrium, we solve $Dp + h = Cp + e$, or

$$(D - C)p = e - h. \tag{8.8}$$

Then the price vector of the n different goods is

$$p = (D - C)^{-1}(e - h).$$

8.4.6 General linear systems

A more general version of the problem described above looks as follows.

$$
\begin{array}{ccccccccc}
a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & a_{1n}x_n & = & b_1 \\
\vdots & & \vdots & & & & \vdots & & \vdots \\
a_{n1}x_1 & + & a_{n2}x_2 & + & \cdots & + & a_{nn}x_n & = & b_n
\end{array}
\tag{8.9}
$$

The objective here is to solve for the "unknowns"$x_1, \dots, x_n$.

We take as given the coefficients $a_{11}, \dots, a_{nn}$ and constants $b_1, \dots, b_n$.

Notice that we are treating a setting where the number of unknowns equals the number of equations.

This is the case where we are most likely to find a well-defined solution.

(The other cases are referred to as overdetermined and underdetermined systems of equations —we defer discussion of these cases until later lectures.)

In matrix form, the system (8.9) becomes

$$Ax = b \quad \text{where} \quad A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}. \tag{8.10}$$

Example 8.4.4

For example, (8.8) has this form with

$$A = D - C, \quad b = e - h \quad \text{and} \quad x = p.$$

When considering problems such as (8.10), we need to ask at least some of the following questions

- Does a solution actually exist?

- If a solution exists, how should we compute it?

8.5 Solving systems of equations

Recall again the system of equations (8.9), which we write here again as

$$Ax = b. \tag{8.11}$$

The problem we face is to find a vector $x \in \mathbb{R}^n$ that solves (8.11), taking b and A as given.

We may not always find a unique vector x that solves (8.11).

We illustrate two such cases below.

8.5.1 No solution

Consider the system of equations given by,

$$x + 3y = 3$$
$$2x + 6y = -8.$$

It can be verified manually that this system has no possible solution.

To illustrate why this situation arises let's plot the two lines.

```
fig, ax = plt.subplots()
x = np.linspace(-10, 10)
plt.plot(x, (3-x)/3, label=f'$x + 3y = 3$')
plt.plot(x, (-8-2*x)/6, label=f'$2x + 6y = -8$')
plt.legend()
plt.show()
```
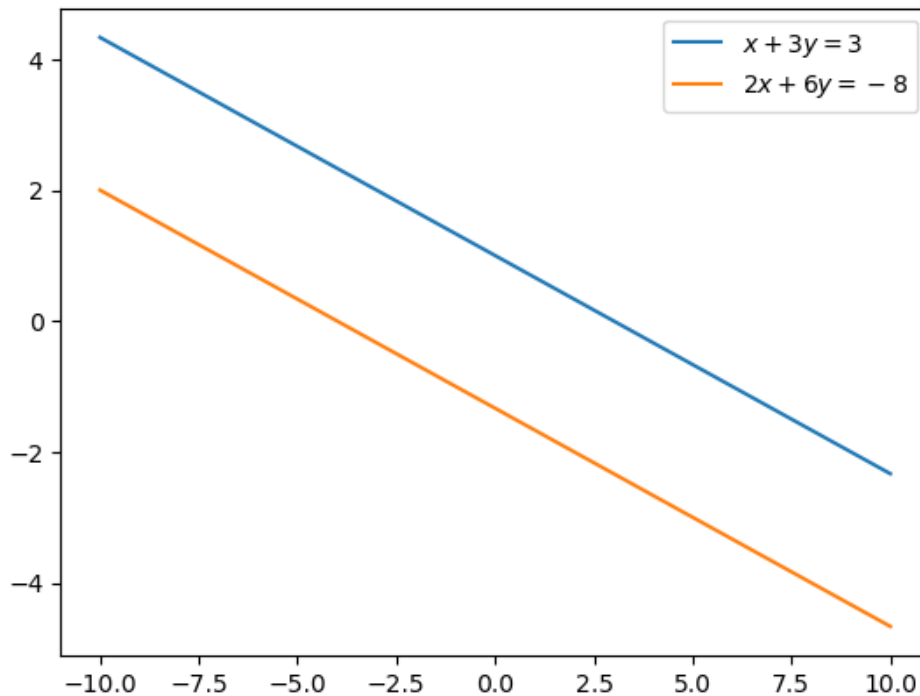

Clearly, these are parallel lines and hence we will never find a point $x \in \mathbb{R}^2$ such that these lines intersect.

Thus, this system has no possible solution.

We can rewrite this system in matrix form as

$$Ax = b \quad \text{where} \quad A = \begin{bmatrix} 1 & 3 \\ 2 & 6 \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} 3 \\ -8 \end{bmatrix}. \tag{8.12}$$

It can be noted that the 2nd row of matrix $A = (2, 6)$ is just a scalar multiple of the 1st row of matrix $A = (1, 3)$.

The rows of matrix A in this case are called **linearly dependent.**

Note: Advanced readers can find a detailed explanation of linear dependence and independence here.

But these details are not needed in what follows.

8.5.2 Many solutions

Now consider,

$$x - 2y = -4$$

$$-2x + 4y = 8.$$

Any vector $v = (x, y)$ such that $x = 2y - 4$ will solve the above system.

Since we can find infinite such vectors this system has infinitely many solutions.

This is because the rows of the corresponding matrix

$$A = \begin{bmatrix} 1 & -2 \\ -2 & 4 \end{bmatrix}. \tag{8.13}$$

are linearly dependent —can you see why?

We now impose conditions on A in (8.11) that rule out these problems.

8.5.3 Nonsingular matrices

To every square matrix we can assign a unique number called the determinant.

For 2×2 matrices, the determinant is given by,

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc.$$

If the determinant of A is not zero, then we say that A is *nonsingular*.

A square matrix A is nonsingular if and only if the rows and columns of A are linearly independent.

A more detailed explanation of matrix inverse can be found here.

You can check yourself that the in (8.12) and (8.13) with linearly dependent rows are singular matrices.

This gives us a useful one-number summary of whether or not a square matrix can be inverted.

In particular, a square matrix A has a nonzero determinant, if and only if it possesses an *inverse matrix* A^{-1}, with the property that $AA^{-1} = A^{-1}A = I$.

As a consequence, if we pre-multiply both sides of $Ax = b$ by A^{-1}, we get

$$x = A^{-1}b. \tag{8.14}$$

This is the solution to $Ax = b$ —the solution we are looking for.

8.5.4 Linear equations with NumPy

In the two good example we obtained the matrix equation,

$$p = (C - D)^{-1}h.$$

where C, D and h are given by (8.5) and (8.6).

This equation is analogous to (8.14) with $A = (C - D)^{-1}$, $b = h$, and $x = p$.

We can now solve for equilibrium prices with NumPy's `linalg` submodule.

All of these routines are Python front ends to time-tested and highly optimized FORTRAN code.

```
C = ((10, 5),       # Matrix C
     (5, 10))
```

Now we change this to a NumPy array.

```
C = np.array(C)
```

```
D = ((-10, -5),     # Matrix D
     (-1, -10))
D = np.array(D)
```

```
h = np.array((100, 50))   # Vector h
h.shape = 2,1             # Transforming h to a column vector
```

```
from numpy.linalg import det, inv
A = C - D
# Check that A is nonsingular (non-zero determinant), and hence invertible
det(A)
```

```
340.0000000000001
```

```
A_inv = inv(A)   # compute the inverse
A_inv
```

```
array([[ 0.05882353, -0.02941176],
       [-0.01764706,  0.05882353]])
```

```
p = A_inv @ h   # equilibrium prices
p
```

```
array([[4.41176471],
       [1.17647059]])
```

```
q = C @ p   # equilibrium quantities
q
```

```
array([[50.        ],
       [33.82352941]])
```

Notice that we get the same solutions as the pencil and paper case.

We can also solve for p using `solve(A, h)` as follows.

```
from numpy.linalg import solve
p = solve(A, h)   # equilibrium prices
p
```

```
array([[4.41176471],
       [1.17647059]])
```

```
q = C @ p   # equilibrium quantities
q
```

```
array([[50.        ],
       [33.82352941]])
```

Observe how we can solve for $x = A^{-1}y$ by either via `inv(A) @ y`, or using `solve(A, y)`.

The latter method uses a different algorithm that is numerically more stable and hence should be the default option.

8.6 Exercises

Exercise 8.6.1

Let's consider a market with 3 commodities - good 0, good 1 and good 2.

The demand for each good depends on the price of the other two goods and is given by:

$$q_0^d = 90 - 15p_0 + 5p_1 + 5p_2$$
$$q_1^d = 60 + 5p_0 - 10p_1 + 10p_2$$
$$q_2^d = 50 + 5p_0 + 5p_1 - 5p_2$$

(Here demand decreases when own price increases but increases when prices of other goods increase.)

The supply of each good is given by:

$$q_0^s = -10 + 20p_0$$
$$q_1^s = -15 + 15p_1$$
$$q_2^s = -5 + 10p_2$$

Equilibrium holds when supply equals demand, i.e, $q_0^d = q_0^s$, $q_1^d = q_1^s$ and $q_2^d = q_2^s$.

1. Set up the market as a system of linear equations.

2. Use matrix algebra to solve for equilibrium prices. Do this using both the `numpy.linalg.solve` and `inv(A)` methods. Compare the solutions.

Solution to Exercise 8.6.1

The generated system would be:

$$35p_0 - 5p_1 - 5p_2 = 100$$
$$-5p_0 + 25p_1 - 10p_2 = 75$$
$$-5p_0 - 5p_1 + 15p_2 = 55$$

In matrix form we will write this as:

$$Ap = b \quad \text{where} \quad A = \begin{bmatrix} 35 & -5 & -5 \\ -5 & 25 & -10 \\ -5 & -5 & 15 \end{bmatrix}, \quad p = \begin{bmatrix} p_0 \\ p_1 \\ p_2 \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} 100 \\ 75 \\ 55 \end{bmatrix}$$

```python
import numpy as np
from numpy.linalg import det
```

```
A = np.array([[35, -5, -5],   # matrix A
              [-5, 25, -10],
              [-5, -5, 15]])

b = np.array((100, 75, 55))   # column vector b
b.shape = (3, 1)

det(A)   # check if A is nonsingular
```

```
9999.99999999999
```

```
# Using inverse
from numpy.linalg import det

A_inv = inv(A)

p = A_inv @ b
p
```

```
array([[4.9625],
       [7.0625],
       [7.675 ]])
```

```
# Using numpy.linalg.solve
from numpy.linalg import solve
p = solve(A, b)
p
```

```
array([[4.9625],
       [7.0625],
       [7.675 ]])
```

The solution is given by: $p_0 = 4.6925,\ p_1 = 7.0625\ $ and $\ p_2 = 7.675$

Exercise 8.6.2

Earlier in the lecture we discussed cases where the system of equations given by $Ax = b$ has no solution.

In this case $Ax = b$ is called an *inconsistent* system of equations.

When faced with an inconsistent system we try to find the best "approximate" solution.

There are various methods to do this, one such method is the **method of least squares.**

Suppose we have an inconsistent system

$$Ax = b \qquad (8.15)$$

where A is an $m \times n$ matrix and b is an $m \times 1$ column vector.

A **least squares solution** to (8.15) is an $n \times 1$ column vector $\hat{x}$ such that, for all other vectors $x \in \mathbb{R}^n$, the distance from $A\hat{x}$ to b is less than the distance from Ax to b.

That is,

$$\|A\hat{x} - b\| \le \|Ax - b\|$$

It can be shown that, for the system of equations $Ax = b$, the least squares solution $\hat{x}$ is

$$\hat{x} = (A^T A)^{-1} A^T b \qquad (8.16)$$

Now consider the general equation of a linear demand curve of a good given by:

$$p = m - nq$$

where p is the price of the good and q is the quantity demanded.

Suppose we are trying to *estimate* the values of m and n.

We do this by repeatedly observing the price and quantity (for example, each month) and then choosing m and n to fit the relationship between p and q.

We have the following observations:

Price	Quantity Demanded
1	9
3	7
8	3

Requiring the demand curve $p = m - nq$ to pass through all these points leads to the following three equations:

$$1 = m - 9n$$
$$3 = m - 7n$$
$$8 = m - 3n$$

Thus we obtain a system of equations $Ax = b$ where $A = \begin{bmatrix} 1 & -9 \\ 1 & -7 \\ 1 & -3 \end{bmatrix}$, $x = \begin{bmatrix} m \\ n \end{bmatrix}$ and $b = \begin{bmatrix} 1 \\ 3 \\ 8 \end{bmatrix}$.

It can be verified that this system has no solutions.

(The problem is that we have three equations and only two unknowns.)

We will thus try to find the best approximate solution for x.

1. Use (8.16) and matrix algebra to find the least squares solution $\hat{x}$.

2. Find the least squares solution using `numpy.linalg.lstsq` and compare the results.

Solution to Exercise 8.6.2

```python
import numpy as np
from numpy.linalg import inv
```

```python
# Using matrix algebra
A = np.array([[1, -9],   # matrix A
              [1, -7],
              [1, -3]])

A_T = np.transpose(A)   # transpose of matrix A

b = np.array((1, 3, 8))   # column vector b
b.shape = (3, 1)

x = inv(A_T @ A) @ A_T @ b
x
```

```
array([[11.46428571],
       [ 1.17857143]])
```

```python
# Using numpy.linalg.lstsq
x, res, _, _ = np.linalg.lstsq(A, b, rcond=None)
```

```
x̂ = [[11.46428571]
 [ 1.17857143]]
‖Ax̂ − b‖² = 0.07142857142857129
```

Here is a visualization of how the least squares method approximates the equation of a line connecting a set of points.

We can also describe this as "fitting"a line between a set of points.

```
fig, ax = plt.subplots()
p = np.array((1, 3, 8))
q = np.array((9, 7, 3))

a, b = x

ax.plot(q, p, 'o', label='observations', markersize=5)
ax.plot(q, a - b*q, 'r', label='Fitted line')
plt.xlabel('quantity demanded')
plt.ylabel('price')
plt.legend()
plt.show()
```
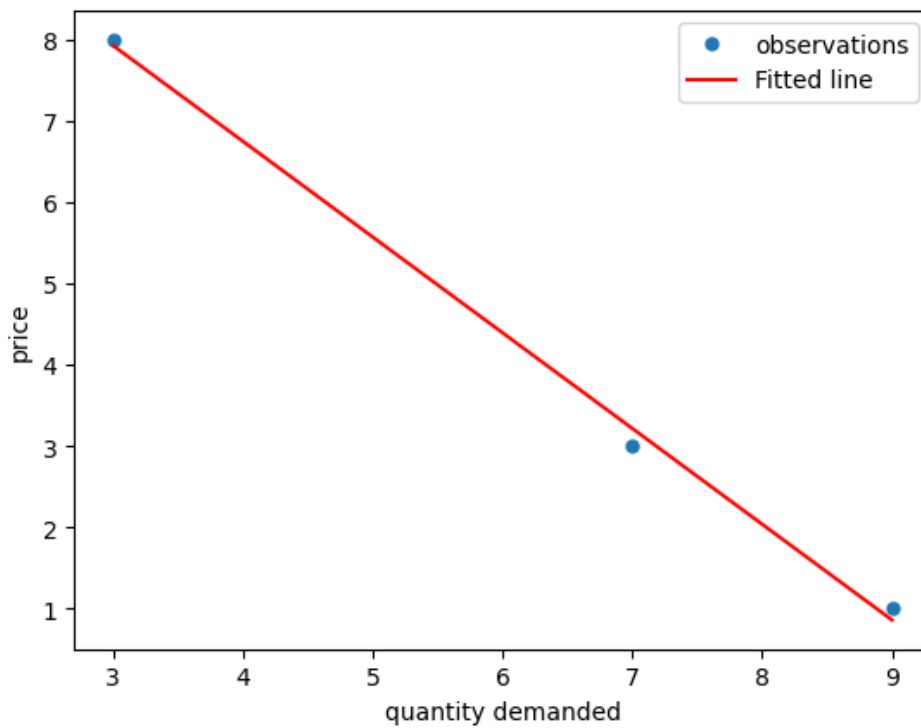

8.6.1 Further reading

The documentation of the `numpy.linalg` submodule can be found here.

More advanced topics in linear algebra can be found here.

COMPLEX NUMBERS AND TRIGONOMETRY

9.1 Overview

This lecture introduces some elementary mathematics and trigonometry.

Useful and interesting in its own right, these concepts reap substantial rewards when studying dynamics generated by linear difference equations or linear differential equations.

For example, these tools are keys to understanding outcomes attained by Paul Samuelson (1939) [Samuelson, 1939] in his classic paper on interactions between the investment accelerator and the Keynesian consumption function, our topic in the lecture Samuelson Multiplier Accelerator.

In addition to providing foundations for Samuelson's work and extensions of it, this lecture can be read as a stand-alone quick reminder of key results from elementary high school trigonometry.

So let's dive in.

9.1.1 Complex Numbers

A complex number has a **real part** x and a purely **imaginary part** y.

The Euclidean, polar, and trigonometric forms of a complex number z are:

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

The second equality above is known as **Euler's formula**

- Euler contributed many other formulas too!

The complex conjugate $\bar{z}$ of z is defined as

$$\bar{z} = x - iy = re^{-i\theta} = r(\cos\theta - i\sin\theta)$$

The value x is the **real** part of z and y is the **imaginary** part of z.

The symbol $|z| = \sqrt{\bar{z} \cdot z} = r$ represents the **modulus** of z.

The value r is the Euclidean distance of vector (x, y) from the origin:

$$r = |z| = \sqrt{x^2 + y^2}$$

The value θ is the angle of (x, y) with respect to the real axis.

Evidently, the tangent of θ is $\left(\frac{y}{x}\right)$.

Therefore,

$$\theta = \tan^{-1}\left(\frac{y}{x}\right)$$

Three elementary trigonometric functions are

$$\cos\theta = \frac{x}{r} = \frac{e^{i\theta} + e^{-i\theta}}{2}, \quad \sin\theta = \frac{y}{r} = \frac{e^{i\theta} - e^{-i\theta}}{2i}, \quad \tan\theta = \frac{y}{x}$$

We'll need the following imports:

```python
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
from sympy import (Symbol, symbols, Eq, nsolve, sqrt, cos, sin, simplify,
                   init_printing, integrate)
```

9.1.2 An Example

Example 9.1.1

Consider the complex number $z = 1 + \sqrt{3}i$.

For $z = 1 + \sqrt{3}i$, $x = 1$, $y = \sqrt{3}$.

It follows that $r = 2$ and $\theta = \tan^{-1}(\sqrt{3}) = \frac{\pi}{3} = 60^o$.

Let's use Python to plot the trigonometric form of the complex number $z = 1 + \sqrt{3}i$.

```python
# Abbreviate useful values and functions
π = np.pi


# Set parameters
r = 2
```

(continues on next page)

```
θ = π/3
x = r * np.cos(θ)
x_range = np.linspace(0, x, 1000)
θ_range = np.linspace(0, θ, 1000)

# Plot
fig = plt.figure(figsize=(8, 8))
ax = plt.subplot(111, projection='polar')

ax.plot((0, θ), (0, r), marker='o', color='b')          # Plot r
ax.plot(np.zeros(x_range.shape), x_range, color='b')      # Plot x
ax.plot(θ_range, x / np.cos(θ_range), color='b')         # Plot y
ax.plot(θ_range, np.full(θ_range.shape, 0.1), color='r')  # Plot ϑ

ax.margins(0) # Let the plot starts at origin

ax.set_title("Trigonometry of complex numbers", va='bottom',
    fontsize='x-large')

ax.set_rmax(2)
ax.set_rticks((0.5, 1, 1.5, 2))   # Less radial ticks
ax.set_rlabel_position(-88.5)     # Get radial labels away from plotted line

ax.text(θ, r+0.01 , r'$z = x + iy = 1 + \sqrt{3}\, i$')   # Label z
ax.text(θ+0.2, 1 , '$r = 2$')                             # Label r
ax.text(0-0.2, 0.5, '$x = 1$')                            # Label x
ax.text(0.5, 1.2, r'$y = \sqrt{3}$')                      # Label y
ax.text(0.25, 0.15, r'$\theta = 60^o$')                   # Label ϑ

ax.grid(True)
plt.show()
```

Trigonometry of complex numbers
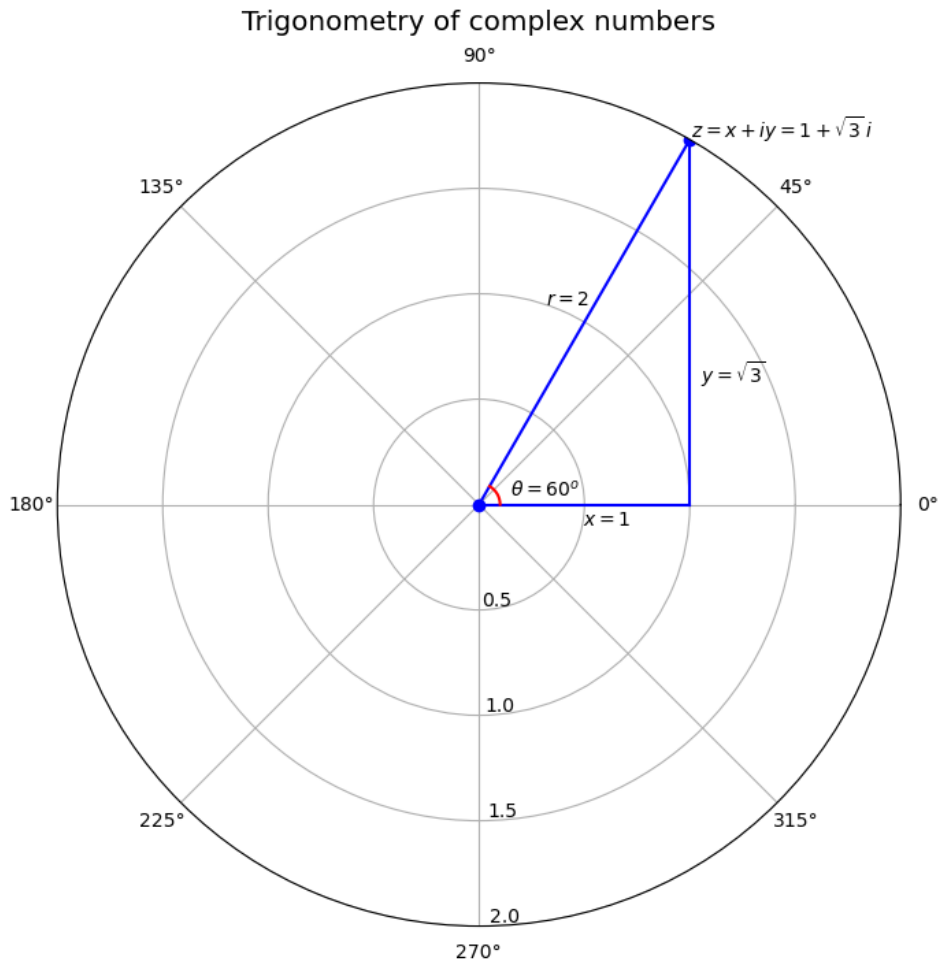
9.2 De Moivre's Theorem

de Moivre's theorem states that:

$$(r(\cos\theta + i\sin\theta))^n = r^n e^{in\theta} = r^n(\cos n\theta + i\sin n\theta)$$

To prove de Moivre's theorem, note that

$$(r(\cos\theta + i\sin\theta))^n = \left(re^{i\theta}\right)^n$$

and compute.

9.3 Applications of de Moivre's Theorem

9.3.1 Example 1

We can use de Moivre's theorem to show that $r = \sqrt{x^2 + y^2}$.

We have

$$1 = e^{i\theta}e^{-i\theta}$$
$$= (\cos\theta + i\sin\theta)(\cos(-\theta) + i\sin(-\theta))$$
$$= (\cos\theta + i\sin\theta)(\cos\theta - i\sin\theta)$$
$$= \cos^2\theta + \sin^2\theta$$
$$= \frac{x^2}{r^2} + \frac{y^2}{r^2}$$

and thus

$$x^2 + y^2 = r^2$$

We recognize this as a theorem of **Pythagoras**.

9.3.2 Example 2

Let $z = re^{i\theta}$ and $\bar{z} = re^{-i\theta}$ so that $\bar{z}$ is the **complex conjugate** of z.

$(z, \bar{z})$ form a **complex conjugate pair** of complex numbers.

Let $a = pe^{i\omega}$ and $\bar{a} = pe^{-i\omega}$ be another complex conjugate pair.

For each element of a sequence of integers $n = 0, 1, 2, \dots,$.

To do so, we can apply de Moivre's formula.

Thus,

$$x_n = az^n + \bar{a}\bar{z}^n$$
$$= pe^{i\omega}(re^{i\theta})^n + pe^{-i\omega}(re^{-i\theta})^n$$
$$= pr^n e^{i(\omega + n\theta)} + pr^n e^{-i(\omega + n\theta)}$$
$$= pr^n[\cos(\omega + n\theta) + i\sin(\omega + n\theta) + \cos(\omega + n\theta) - i\sin(\omega + n\theta)]$$
$$= 2pr^n \cos(\omega + n\theta)$$

9.3.3 Example 3

This example provides machinery that is at the heard of Samuelson's analysis of his multiplier-accelerator model [Samuelson, 1939].

Thus, consider a **second-order linear difference equation**

$$x_{n+2} = c_1 x_{n+1} + c_2 x_n$$

whose **characteristic polynomial** is

$$z^2 - c_1 z - c_2 = 0$$

or

$$(z^2 - c_1 z - c_2) = (z - z_1)(z - z_2) = 0$$

has roots z_1, z_1.

A **solution** is a sequence $\{x_n\}_{n=0}^{\infty}$ that satisfies the difference equation.

Under the following circumstances, we can apply our example 2 formula to solve the difference equation

- the roots z_1, z_2 of the characteristic polynomial of the difference equation form a complex conjugate pair

- the values x_0, x_1 are given initial conditions

To solve the difference equation, recall from example 2 that

$$x_n = 2pr^n \cos(\omega + n\theta)$$

where ω, p are coefficients to be determined from information encoded in the initial conditions x_1, x_0.

Since $x_0 = 2p \cos \omega$ and $x_1 = 2pr \cos(\omega + \theta)$ the ratio of x_1 to x_0 is

$$\frac{x_1}{x_0} = \frac{r \cos(\omega + \theta)}{\cos \omega}$$

We can solve this equation for ω then solve for p using $x_0 = 2pr^0 \cos(\omega + n\theta)$.

With the `sympy` package in Python, we are able to solve and plot the dynamics of x_n given different values of n.

In this example, we set the initial values: - $r = 0.9$ - $\theta = \frac{1}{4}\pi$ - $x_0 = 4$ - $x_1 = r \cdot 2\sqrt{2} = 1.8\sqrt{2}$.

We first numerically solve for ω and p using `nsolve` in the `sympy` package based on the above initial condition:

```
# Set parameters
r = 0.9
θ = π/4
x0 = 4
```

```python
x1 = 2 * r * sqrt(2)

# Define symbols to be calculated
ω, p = symbols('ω p', real=True)

# Solve for ω
## Note: we choose the solution near 0
eq1 = Eq(x1/x0 - r * cos(ω+θ) / cos(ω), 0)
ω = nsolve(eq1, ω, 0)
ω = float(ω)
print(f'ω = {ω:1.3f}')

# Solve for p
eq2 = Eq(x0 - 2 * p * cos(ω), 0)
p = nsolve(eq2, p, 0)
p = float(p)
print(f'p = {p:1.3f}')
```

```
ω = 0.000
p = 2.000
```

Using the code above, we compute that $\omega = 0$ and $p = 2$.

Then we plug in the values we solve for ω and p and plot the dynamic.

```python
# Define range of n
max_n = 30
n = np.arange(0, max_n+1, 0.01)

# Define x_n
x = lambda n: 2 * p * r**n * np.cos(ω + n * θ)

# Plot
fig, ax = plt.subplots(figsize=(12, 8))

ax.plot(n, x(n))
ax.set(xlim=(0, max_n), ylim=(-5, 5), xlabel='$n$', ylabel='$x_n$')

# Set x-axis in the middle of the plot
ax.spines['bottom'].set_position('center')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
```

```
ax.yaxis.set_ticks_position('left')

ticklab = ax.xaxis.get_ticklabels()[0]  # Set x-label position
trans = ticklab.get_transform()
ax.xaxis.set_label_coords(31, 0, transform=trans)

ticklab = ax.yaxis.get_ticklabels()[0]  # Set y-label position
trans = ticklab.get_transform()
ax.yaxis.set_label_coords(0, 5, transform=trans)

ax.grid()
plt.show()
```
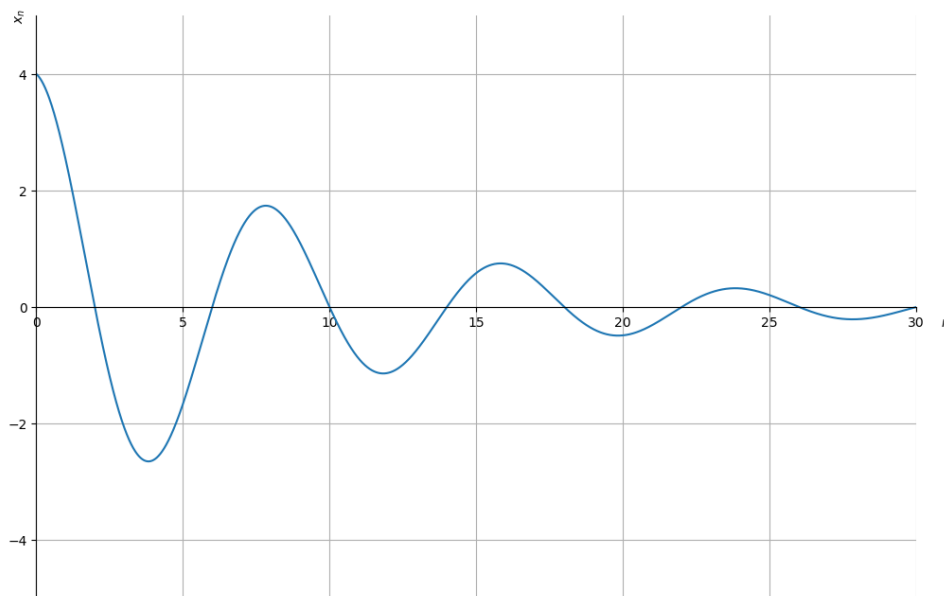

9.3.4 Trigonometric Identities

We can obtain a complete suite of trigonometric identities by appropriately manipulating polar forms of complex numbers.

We'll get many of them by deducing implications of the equality

$$e^{i(\omega+\theta)} = e^{i\omega}e^{i\theta}$$

For example, we'll calculate identities for

$\cos(\omega + \theta)$ and $\sin(\omega + \theta)$.

Using the sine and cosine formulas presented at the beginning of this lecture, we have:

$$\cos(\omega + \theta) = \frac{e^{i(\omega+\theta)} + e^{-i(\omega+\theta)}}{2}$$

$$\sin(\omega + \theta) = \frac{e^{i(\omega+\theta)} - e^{-i(\omega+\theta)}}{2i}$$

We can also obtain the trigonometric identities as follows:

$$\cos(\omega + \theta) + i\sin(\omega + \theta) = e^{i(\omega+\theta)}$$
$$= e^{i\omega}e^{i\theta}$$
$$= (\cos\omega + i\sin\omega)(\cos\theta + i\sin\theta)$$
$$= (\cos\omega\cos\theta - \sin\omega\sin\theta) + i(\cos\omega\sin\theta + \sin\omega\cos\theta)$$

Since both real and imaginary parts of the above formula should be equal, we get:

$$\cos(\omega + \theta) = \cos\omega\cos\theta - \sin\omega\sin\theta$$

$$\sin(\omega + \theta) = \cos\omega\sin\theta + \sin\omega\cos\theta$$

The equations above are also known as the **angle sum identities**. We can verify the equations using the `simplify` function in the `sympy` package:

```
# Define symbols
ω, θ = symbols('ω θ', real=True)

# Verify
print("cos(ω)cos(θ) - sin(ω)sin(θ) =",
    simplify(cos(ω)*cos(θ) - sin(ω) * sin(θ)))
print("cos(ω)sin(θ) + sin(ω)cos(θ) =",
    simplify(cos(ω)*sin(θ) + sin(ω) * cos(θ)))
```

```
cos(ω)cos(θ) - sin(ω)sin(θ) = cos(θ + ω)
cos(ω)sin(θ) + sin(ω)cos(θ) = sin(θ + ω)
```

9.3.5 Trigonometric Integrals

We can also compute the trigonometric integrals using polar forms of complex numbers.

For example, we want to solve the following integral:

$$\int_{-\pi}^{\pi} \cos(\omega)\sin(\omega)\,d\omega$$

Using Euler's formula, we have:

$$
\begin{aligned}
\int \cos(\omega)\sin(\omega)\,d\omega &= \int \frac{(e^{i\omega}+e^{-i\omega})}{2}\frac{(e^{i\omega}-e^{-i\omega})}{2i}\,d\omega \\
&= \frac{1}{4i}\int e^{2i\omega}-e^{-2i\omega}\,d\omega \\
&= \frac{1}{4i}\left(\frac{-i}{2}e^{2i\omega}-\frac{i}{2}e^{-2i\omega}+C_1\right) \\
&= -\frac{1}{8}\left[\left(e^{i\omega}\right)^2+\left(e^{-i\omega}\right)^2-2\right]+C_2 \\
&= -\frac{1}{8}(e^{i\omega}-e^{-i\omega})^2+C_2 \\
&= \frac{1}{2}\left(\frac{e^{i\omega}-e^{-i\omega}}{2i}\right)^2+C_2 \\
&= \frac{1}{2}\sin^2(\omega)+C_2
\end{aligned}
$$

and thus:

$$
\int_{-\pi}^{\pi}\cos(\omega)\sin(\omega)\,d\omega = \frac{1}{2}\sin^2(\pi)-\frac{1}{2}\sin^2(-\pi) = 0
$$

We can verify the analytical as well as numerical results using `integrate` in the `sympy` package:

```python
# Set initial printing
init_printing(use_latex="mathjax")


ω = Symbol('ω')
print('The analytical solution for integral of cos(ω)sin(ω) is:')
integrate(cos(ω) * sin(ω), ω)
```

    The analytical solution for integral of cos(ω)sin(ω) is:

$$
\frac{\sin^2(\omega)}{2}
$$

```python
print('The numerical solution for the integral of cos(ω)sin(ω) \
from -π to π is:')
integrate(cos(ω) * sin(ω), (ω, -π, π))
```

    The numerical solution for the integral of cos(ω)sin(ω) from -π to π is:

$$0$$

9.3.6 Exercises

Exercise 9.3.1

We invite the reader to verify analytically and with the `sympy` package the following two equalities:

$$\int_{-\pi}^{\pi} \cos(\omega)^2 \, d\omega = \pi$$

$$\int_{-\pi}^{\pi} \sin(\omega)^2 \, d\omega = \pi$$

Solution to Exercise 9.3.1

Let's import symbolic π from `sympy`

```python
# Import symbolic π from sympy
from sympy import pi
```

```python
print('The analytical solution for the integral of cos(ω)**2 \
from -π to π is:')

integrate(cos(ω)**2, (ω, -pi, pi))
```

    The analytical solution for the integral of cos(ω)**2 from -π to π is:

$$\pi$$

```python
print('The analytical solution for the integral of sin(ω)**2 \
from -π to π is:')

integrate(sin(ω)**2, (ω, -pi, pi))
```

    The analytical solution for the integral of sin(ω)**2 from -π to π is:

$$\pi$$

GEOMETRIC SERIES FOR ELEMENTARY ECONOMICS

10.1 Overview

The lecture describes important ideas in economics that use the mathematics of geometric series.

Among these are

- the Keynesian **multiplier**

- the money **multiplier** that prevails in fractional reserve banking systems

- interest rates and present values of streams of payouts from assets

(As we shall see below, the term **multiplier** comes down to meaning **sum of a convergent geometric series**)

These and other applications prove the truth of the wise crack that

"In economics, a little knowledge of geometric series goes a long way."

Below we'll use the following imports:

```python
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
import numpy as np
import sympy as sym
from sympy import init_printing
from matplotlib import cm
```

10.2 Key formulas

To start, let c be a real number that lies strictly between -1 and 1.

- We often write this as $c \in (-1, 1)$.

- Here $(-1, 1)$ denotes the collection of all real numbers that are strictly less than 1 and strictly greater than -1.

- The symbol $\in$ means *in* or *belongs to the set after the symbol*.

We want to evaluate geometric series of two types –infinite and finite.

10.2.1 Infinite geometric series

The first type of geometric that interests us is the infinite series

$$1 + c + c^2 + c^3 + \cdots$$

Where $\cdots$ means that the series continues without end.

The key formula is

$$1 + c + c^2 + c^3 + \cdots = \frac{1}{1 - c} \tag{10.1}$$

To prove key formula (10.1), multiply both sides by $(1 - c)$ and verify that if $c \in (-1, 1)$, then the outcome is the equation $1 = 1$.

10.2.2 Finite geometric series

The second series that interests us is the finite geometric series

$$1 + c + c^2 + c^3 + \cdots + c^T$$

where T is a positive integer.

The key formula here is

$$1 + c + c^2 + c^3 + \cdots + c^T = \frac{1 - c^{T+1}}{1 - c}$$

Remark: The above formula works for any value of the scalar c. We don't have to restrict c to be in the set $(-1, 1)$.

We now move on to describe some famous economic applications of geometric series.

10.3 Example: The Money Multiplier in Fractional Reserve Banking

In a fractional reserve banking system, banks hold only a fraction $r \in (0, 1)$ of cash behind each **deposit receipt** that they issue

- In recent times

    - cash consists of pieces of paper issued by the government and called dollars or pounds or ...

    - a *deposit* is a balance in a checking or savings account that entitles the owner to ask the bank for immediate payment in cash

- When the UK and France and the US were on either a gold or silver standard (before 1914, for example)

    - cash was a gold or silver coin

    - a *deposit receipt* was a *bank note* that the bank promised to convert into gold or silver on demand; (sometimes it was also a checking or savings account balance)

Economists and financiers often define the **supply of money** as an economy-wide sum of **cash** plus **deposits**.

In a **fractional reserve banking system** (one in which the reserve ratio r satisfies $0 < r < 1$), **banks create money** by issuing deposits *backed* by fractional reserves plus loans that they make to their customers.

A geometric series is a key tool for understanding how banks create money (i.e., deposits) in a fractional reserve system.

The geometric series formula (10.1) is at the heart of the classic model of the money creation process –one that leads us to the celebrated **money multiplier**.

10.3.1 A simple model

There is a set of banks named $i = 0, 1, 2, ...$

Bank i's loans L_i, deposits D_i, and reserves R_i must satisfy the balance sheet equation (because **balance sheets balance**):

$$L_i + R_i = D_i \tag{10.2}$$

The left side of the above equation is the sum of the bank's **assets**, namely, the loans L_i it has outstanding plus its reserves of cash R_i.

The right side records bank i's liabilities, namely, the deposits D_i held by its depositors; these are IOU's from the bank to its depositors in the form of either checking accounts or savings accounts (or before 1914, bank notes issued by a bank stating promises to redeem notes for gold or silver on demand).

Each bank i sets its reserves to satisfy the equation

$$R_i = rD_i \tag{10.3}$$

where $r \in (0, 1)$ is its **reserve-deposit ratio** or **reserve ratio** for short

- the reserve ratio is either set by a government or chosen by banks for precautionary reasons

Next we add a theory stating that bank $i + 1$'s deposits depend entirely on loans made by bank i, namely

$$D_{i+1} = L_i \tag{10.4}$$

Thus, we can think of the banks as being arranged along a line with loans from bank i being immediately deposited in $i + 1$

- in this way, the debtors to bank i become creditors of bank $i + 1$

Finally, we add an *initial condition* about an exogenous level of bank 0's deposits

$$D_0 \text{ is given exogenously}$$

We can think of D_0 as being the amount of cash that a first depositor put into the first bank in the system, bank number $i = 0$.

Now we do a little algebra.

Combining equations (10.2) and (10.3) tells us that

$$L_i = (1 - r)D_i \tag{10.5}$$

This states that bank i loans a fraction $(1 - r)$ of its deposits and keeps a fraction r as cash reserves.

Combining equation (10.5) with equation (10.4) tells us that

$$D_{i+1} = (1 - r)D_i \text{ for } i \geq 0$$

which implies that

$$D_i = (1 - r)^i D_0 \text{ for } i \geq 0 \tag{10.6}$$

Equation (10.6) expresses D_i as the i th term in the product of D_0 and the geometric series

$$1, (1 - r), (1 - r)^2, \cdots$$

Therefore, the sum of all deposits in our banking system $i = 0, 1, 2, \ldots$ is

$$\sum_{i=0}^{\infty} (1 - r)^i D_0 = \frac{D_0}{1 - (1 - r)} = \frac{D_0}{r} \tag{10.7}$$

10.3.2 Money multiplier

The **money multiplier** is a number that tells the multiplicative factor by which an exogenous injection of cash into bank 0 leads to an increase in the total deposits in the banking system.

Equation (10.7) asserts that the **money multiplier** is $\frac{1}{r}$

- An initial deposit of cash of D_0 in bank 0 leads the banking system to create total deposits of $\frac{D_0}{r}$.

- The initial deposit D_0 is held as reserves, distributed throughout the banking system according to $D_0 = \sum_{i=0}^{\infty} R_i$.

10.4 Example: The Keynesian Multiplier

The famous economist John Maynard Keynes and his followers created a simple model intended to determine national income y in circumstances in which

- there are substantial unemployed resources, in particular **excess supply** of labor and capital

- prices and interest rates fail to adjust to make aggregate **supply equal demand** (e.g., prices and interest rates are frozen)

- national income is entirely determined by aggregate demand

10.4.1 Static version

An elementary Keynesian model of national income determination consists of three equations that describe aggregate demand for y and its components.

The first equation is a national income identity asserting that consumption c plus investment i equals national income y:

$$c + i = y$$

The second equation is a Keynesian consumption function asserting that people consume a fraction $b \in (0, 1)$ of their income:

$$c = by$$

The fraction $b \in (0, 1)$ is called the **marginal propensity to consume**.

The fraction $1 - b \in (0, 1)$ is called the **marginal propensity to save**.

The third equation simply states that investment is exogenous at level i.

- *exogenous* means *determined outside this model*.

Substituting the second equation into the first gives $(1 - b)y = i$.

Solving this equation for y gives

$$y = \frac{1}{1 - b} i$$

The quantity $\frac{1}{1-b}$ is called the **investment multiplier** or simply the **multiplier**.

Applying the formula for the sum of an infinite geometric series, we can write the above equation as

$$y = i \sum_{t=0}^{\infty} b^t$$

where t is a nonnegative integer.

So we arrive at the following equivalent expressions for the multiplier:

$$\frac{1}{1-b} = \sum_{t=0}^{\infty} b^t$$

The expression $\sum_{t=0}^{\infty} b^t$ motivates an interpretation of the multiplier as the outcome of a dynamic process that we describe next.

10.4.2 Dynamic version

We arrive at a dynamic version by interpreting the nonnegative integer t as indexing time and changing our specification of the consumption function to take time into account

- we add a one-period lag in how income affects consumption

We let c_t be consumption at time t and i_t be investment at time t.

We modify our consumption function to assume the form

$$c_t = by_{t-1}$$

so that b is the marginal propensity to consume (now) out of last period's income.

We begin with an initial condition stating that

$$y_{-1} = 0$$

We also assume that

$$i_t = i \ \text{ for all } t \geq 0$$

so that investment is constant over time.

It follows that

$$y_0 = i + c_0 = i + by_{-1} = i$$

and

$$y_1 = c_1 + i = by_0 + i = (1 + b)i$$

and

$$y_2 = c_2 + i = by_1 + i = (1 + b + b^2)i$$

and more generally

$$y_t = by_{t-1} + i = (1 + b + b^2 + \cdots + b^t)i$$

or

$$y_t = \frac{1 - b^{t+1}}{1 - b} i$$

Evidently, as $t \to +\infty$,

$$y_t \to \frac{1}{1 - b} i$$

Remark 1: The above formula is often applied to assert that an exogenous increase in investment of Δi at time 0 ignites a dynamic process of increases in national income by successive amounts

$$\Delta i, (1 + b)\Delta i, (1 + b + b^2)\Delta i, \cdots$$

at times $0, 1, 2, ...$.

Remark 2 Let g_t be an exogenous sequence of government expenditures.

If we generalize the model so that the national income identity becomes

$$c_t + i_t + g_t = y_t$$

then a version of the preceding argument shows that the **government expenditures multiplier** is also $\frac{1}{1-b}$, so that a permanent increase in government expenditures ultimately leads to an increase in national income equal to the multiplier times the increase in government expenditures.

10.5 Example: Interest Rates and Present Values

We can apply our formula for geometric series to study how interest rates affect values of streams of dollar payments that extend over time.

We work in discrete time and assume that $t = 0, 1, 2, ...$ indexes time.

We let $r \in (0, 1)$ be a one-period **net nominal interest rate**

- if the nominal interest rate is 5 percent, then $r = .05$

A one-period **gross nominal interest rate** R is defined as

$$R = 1 + r \in (1, 2)$$

- if $r = .05$, then $R = 1.05$

Remark: The gross nominal interest rate R is an **exchange rate** or **relative price** of dollars at between times t and $t+1$. The units of R are dollars at time $t + 1$ per dollar at time t.

When people borrow and lend, they trade dollars now for dollars later or dollars later for dollars now.

The price at which these exchanges occur is the gross nominal interest rate.

- If I sell x dollars to you today, you pay me Rx dollars tomorrow.

- This means that you borrowed x dollars for me at a gross interest rate R and a net interest rate r.

We assume that the net nominal interest rate r is fixed over time, so that R is the gross nominal interest rate at times $t = 0, 1, 2, ...$

Two important geometric sequences are

$$1, R, R^2, \cdots \tag{10.8}$$

and

$$1, R^{-1}, R^{-2}, \cdots \tag{10.9}$$

Sequence (10.8) tells us how dollar values of an investment **accumulate** through time.

Sequence (10.9) tells us how to **discount** future dollars to get their values in terms of today's dollars.

10.5.1 Accumulation

Geometric sequence (10.8) tells us how one dollar invested and re-invested in a project with gross one period nominal rate of return accumulates

- here we assume that net interest payments are reinvested in the project

- thus, 1 dollar invested at time 0 pays interest r dollars after one period, so we have $r + 1 = R$ dollars at time1

- at time 1 we reinvest $1 + r = R$ dollars and receive interest of rR dollars at time 2 plus the *principal* R dollars, so we receive $rR + R = (1 + r)R = R^2$ dollars at the end of period 2

- and so on

Evidently, if we invest x dollars at time 0 and reinvest the proceeds, then the sequence

$$x, xR, xR^2, \cdots$$

tells how our account accumulates at dates $t = 0, 1, 2, ...$

10.5.2 Discounting

Geometric sequence (10.9) tells us how much future dollars are worth in terms of today's dollars.

Remember that the units of R are dollars at $t + 1$ per dollar at t.

It follows that

- the units of R^{-1} are dollars at t per dollar at $t + 1$

- the units of R^{-2} are dollars at t per dollar at $t+2$

- and so on; the units of R^{-j} are dollars at t per dollar at $t+j$

So if someone has a claim on x dollars at time $t+j$, it is worth xR^{-j} dollars at time t (e.g., today).

10.5.3 Application to asset pricing

A **lease** requires a payments stream of x_t dollars at times $t = 0, 1, 2, ...$ where

$$x_t = G^t x_0$$

where $G = (1+g)$ and $g \in (0, 1)$.

Thus, lease payments increase at g percent per period.

For a reason soon to be revealed, we assume that $G < R$.

The **present value** of the lease is

$$\begin{aligned} p_0 &= x_0 + x_1/R + x_2/(R^2) + \cdots \\ &= x_0(1 + GR^{-1} + G^2R^{-2} + \cdots) \\ &= x_0 \frac{1}{1 - GR^{-1}} \end{aligned}$$

where the last line uses the formula for an infinite geometric series.

Recall that $R = 1 + r$ and $G = 1 + g$ and that $R > G$ and $r > g$ and that r and g are typically small numbers, e.g., .05 or .03.

Use the Taylor series of $\frac{1}{1+r}$ about $r = 0$, namely,

$$\frac{1}{1+r} = 1 - r + r^2 - r^3 + \cdots$$

and the fact that r is small to approximate $\frac{1}{1+r} \approx 1 - r$.

Use this approximation to write p_0 as

$$\begin{aligned} p_0 &= x_0 \frac{1}{1 - GR^{-1}} \\ &= x_0 \frac{1}{1 - (1+g)(1-r)} \\ &= x_0 \frac{1}{1 - (1 + g - r - rg)} \\ &\approx x_0 \frac{1}{r - g} \end{aligned}$$

where the last step uses the approximation $rg \approx 0$.

The approximation

$$p_0 = \frac{x_0}{r - g}$$

is known as the **Gordon formula** for the present value or current price of an infinite payment stream $x_0 G^t$ when the nominal one-period interest rate is r and when $r > g$.

We can also extend the asset pricing formula so that it applies to finite leases.

Let the payment stream on the lease now be x_t for $t = 1, 2, \ldots, T$, where again

$$x_t = G^t x_0$$

The present value of this lease is:

$$\begin{aligned} p_0 &= x_0 + x_1/R + \cdots + x_T/R^T \\ &= x_0(1 + GR^{-1} + \cdots + G^T R^{-T}) \\ &= \frac{x_0(1 - G^{T+1} R^{-(T+1)})}{1 - GR^{-1}} \end{aligned}$$

Applying the Taylor series to $R^{-(T+1)}$ about $r = 0$ we get:

$$\frac{1}{(1+r)^{T+1}} = 1 - r(T+1) + \frac{1}{2} r^2(T+1)(T+2) + \cdots \approx 1 - r(T+1)$$

Similarly, applying the Taylor series to G^{T+1} about $g = 0$:

$$(1+g)^{T+1} = 1 + (T+1)g + \frac{T(T+1)}{2!} g^2 + \frac{(T-1)T(T+1)}{3!} g^3 + \cdots \approx 1 + (T+1)g$$

Thus, we get the following approximation:

$$p_0 = \frac{x_0(1 - (1 + (T+1)g)(1 - r(T+1)))}{1 - (1-r)(1+g)}$$

Expanding:

$$\begin{aligned} p_0 &= \frac{x_0(1 - 1 + (T+1)^2 rg + r(T+1) - g(T+1))}{1 - 1 + r - g + rg} \\ &= \frac{x_0(T+1)((T+1)rg + r - g)}{r - g + rg} \\ &= \frac{x_0(T+1)(r - g)}{r - g + rg} + \frac{x_0 rg(T+1)^2}{r - g + rg} \\ &\approx \frac{x_0(T+1)(r - g)}{r - g} + \frac{x_0 rg(T+1)}{r - g} \\ &= x_0(T+1) + \frac{x_0 rg(T+1)}{r - g} \end{aligned}$$

We could have also approximated by removing the second term $rgx_0(T+1)$ when T is relatively small compared to $1/(rg)$ to get $x_0(T+1)$ as in the finite stream approximation.

We will plot the true finite stream present-value and the two approximations, under different values of T, and g and r in Python.

First we plot the true finite stream present-value after computing it below

```python
# True present value of a finite lease
def finite_lease_pv_true(T, g, r, x_0):
    G = (1 + g)
    R = (1 + r)
    return (x_0 * (1 - G**(T + 1) * R**(-T - 1))) / (1 - G * R**(-1))
# First approximation for our finite lease


def finite_lease_pv_approx_1(T, g, r, x_0):
    p = x_0 * (T + 1) + x_0 * r * g * (T + 1) / (r - g)
    return p


# Second approximation for our finite lease
def finite_lease_pv_approx_2(T, g, r, x_0):
    return (x_0 * (T + 1))


# Infinite lease
def infinite_lease(g, r, x_0):
    G = (1 + g)
    R = (1 + r)
    return x_0 / (1 - G * R**(-1))
```

Now that we have defined our functions, we can plot some outcomes.

First we study the quality of our approximations

```python
def plot_function(axes, x_vals, func, args):
    axes.plot(x_vals, func(*args), label=func.__name__)


T_max = 50


T = np.arange(0, T_max+1)
g = 0.02
r = 0.03
x_0 = 1


our_args = (T, g, r, x_0)
funcs = [finite_lease_pv_true,
        finite_lease_pv_approx_1,
        finite_lease_pv_approx_2]
        # the three functions we want to compare


fig, ax = plt.subplots()
for f in funcs:
    plot_function(ax, T, f, our_args)
```
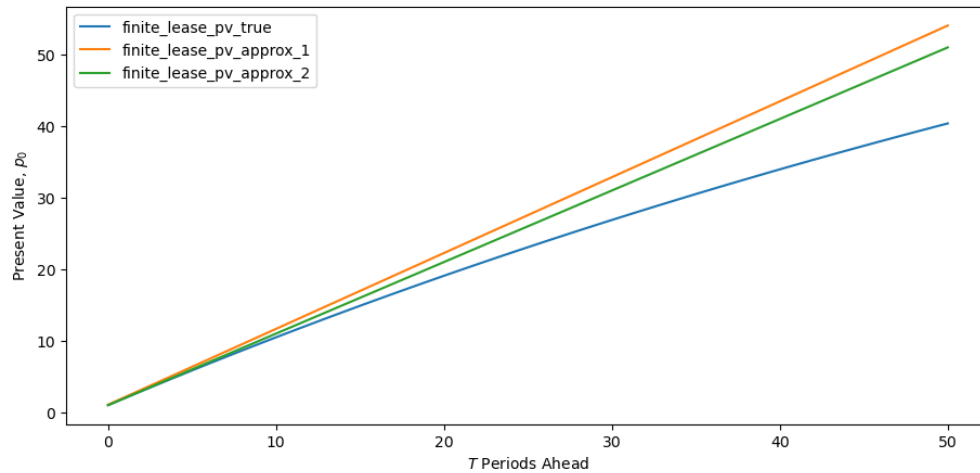
(continues on next page)

```
ax.legend()
ax.set_xlabel('$T$ Periods Ahead')
ax.set_ylabel('Present Value, $p_0$')
plt.show()
```


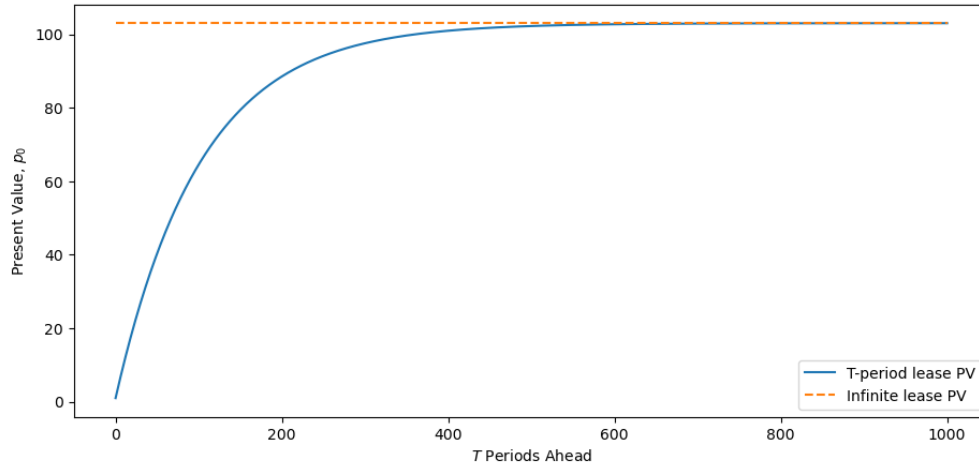Fig. 10.1: Finite lease present value T periods ahead

Evidently our approximations perform well for small values of T.

However, holding g and r fixed, our approximations deteriorate as T increases.

Next we compare the infinite and finite duration lease present values over different lease lengths T.

```
# Convergence of infinite and finite
T_max = 1000
T = np.arange(0, T_max+1)
fig, ax = plt.subplots()
f_1 = finite_lease_pv_true(T, g, r, x_0)
f_2 = np.full(T_max+1, infinite_lease(g, r, x_0))
ax.plot(T, f_1, label='T-period lease PV')
ax.plot(T, f_2, '--', label='Infinite lease PV')
ax.set_xlabel('$T$ Periods Ahead')
ax.set_ylabel('Present Value, $p_0$')
ax.legend()
plt.show()
```

The graph above shows how as duration $T \to +\infty$, the value of a lease of duration T approaches the value of a perpetual

Fig. 10.2: Infinite and finite lease present value T periods ahead

lease.

Now we consider two different views of what happens as r and g covary

```
# First view
# Changing r and g
fig, ax = plt.subplots()
ax.set_ylabel('Present Value, $p_0$')
ax.set_xlabel('$T$ periods ahead')
T_max = 10
T=np.arange(0, T_max+1)

rs, gs = (0.9, 0.5, 0.4001, 0.4), (0.4, 0.4, 0.4, 0.5),
comparisons = ('$\gg$', '$>$', r'$\approx$', '$<$')
for r, g, comp in zip(rs, gs, comparisons):
    ax.plot(finite_lease_pv_true(T, g, r, x_0), label=f'r(={r}) {comp} g(={g})')

ax.legend()
plt.show()
```
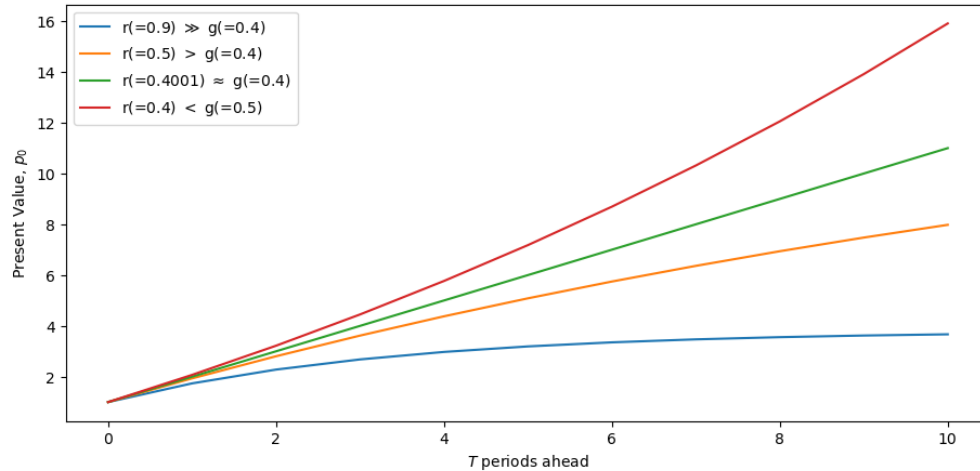
This graph gives a big hint for why the condition $r > g$ is necessary if a lease of length $T = +\infty$ is to have finite value.

For fans of 3-d graphs the same point comes through in the following graph.

If you aren't enamored of 3-d graphs, feel free to skip the next visualization!

```
# Second view
fig = plt.figure(figsize = [16, 5])
```

Fig. 10.3: Value of lease of length T

```python
T = 3
ax = plt.subplot(projection='3d')
r = np.arange(0.01, 0.99, 0.005)
g = np.arange(0.011, 0.991, 0.005)

rr, gg = np.meshgrid(r, g)
z = finite_lease_pv_true(T, gg, rr, x_0)

# Removes points where undefined
same = (rr == gg)
z[same] = np.nan
surf = ax.plot_surface(rr, gg, z, cmap=cm.coolwarm,
    antialiased=True, clim=(0, 15))
fig.colorbar(surf, shrink=0.5, aspect=5)
ax.set_xlabel('$r$')
ax.set_ylabel('$g$')
ax.set_zlabel('Present Value, $p_0$')
ax.view_init(20, 8)
plt.show()
```

We can use a little calculus to study how the present value p_0 of a lease varies with r and g.

We will use a library called SymPy.

SymPy enables us to do symbolic math calculations including computing derivatives of algebraic equations.

We will illustrate how it works by creating a symbolic expression that represents our present value formula for an infinite
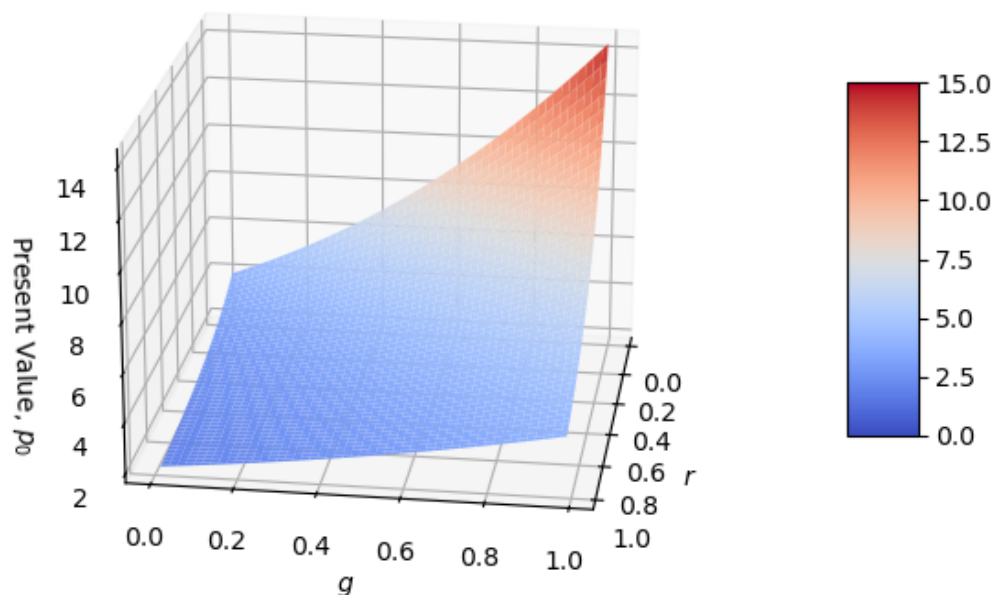
Fig. 10.4: Three period lease PV with varying g and r

lease.

After that, we'll use SymPy to compute derivatives

```
# Creates algebraic symbols that can be used in an algebraic expression
g, r, x0 = sym.symbols('g, r, x0')
G = (1 + g)
R = (1 + r)
p0 = x0 / (1 - G * R**(-1))
init_printing(use_latex='mathjax')
print('Our formula is:')
p0
```

```
Our formula is:
```

$$\frac{x_0}{-\frac{g+1}{r+1}+1}$$

```
print('dp0 / dg is:')
dp_dg = sym.diff(p0, g)
dp_dg
```

```
dp0 / dg is:
```

$$\frac{x_0}{(r+1)\left(-\frac{g+1}{r+1}+1\right)^2}$$

```
print('dp0 / dr is:')
dp_dr = sym.diff(p0, r)
dp_dr
```

```
dp0 / dr is:
```

$$-\frac{x_0\,(g+1)}{(r+1)^2\left(-\frac{g+1}{r+1}+1\right)^2}$$

We can see that for $\frac{\partial p_0}{\partial r} < 0$ as long as $r > g$, $r > 0$ and $g > 0$ and x_0 is positive, so $\frac{\partial p_0}{\partial r}$ will always be negative.

Similarly, $\frac{\partial p_0}{\partial g} > 0$ as long as $r > g$, $r > 0$ and $g > 0$ and x_0 is positive, so $\frac{\partial p_0}{\partial g}$ will always be positive.

10.6 Back to the Keynesian multiplier

We will now go back to the case of the Keynesian multiplier and plot the time path of y_t, given that consumption is a constant fraction of national income, and investment is fixed.

```
# Function that calculates a path of y
def calculate_y(i, b, g, T, y_init):
    y = np.zeros(T+1)
    y[0] = i + b * y_init + g
    for t in range(1, T+1):
        y[t] = b * y[t-1] + i + g
    return y


# Initial values
i_0 = 0.3
g_0 = 0.3
# 2/3 of income goes towards consumption
b = 2/3
y_init = 0
T = 100
```

```
fig, ax = plt.subplots()
ax.set_xlabel('$t$')
ax.set_ylabel('$y_t$')
ax.plot(np.arange(0, T+1), calculate_y(i_0, b, g_0, T, y_init))
# Output predicted by geometric series
ax.hlines(i_0 / (1 - b) + g_0 / (1 - b), xmin=-1, xmax=101, linestyles='--')
plt.show()
```
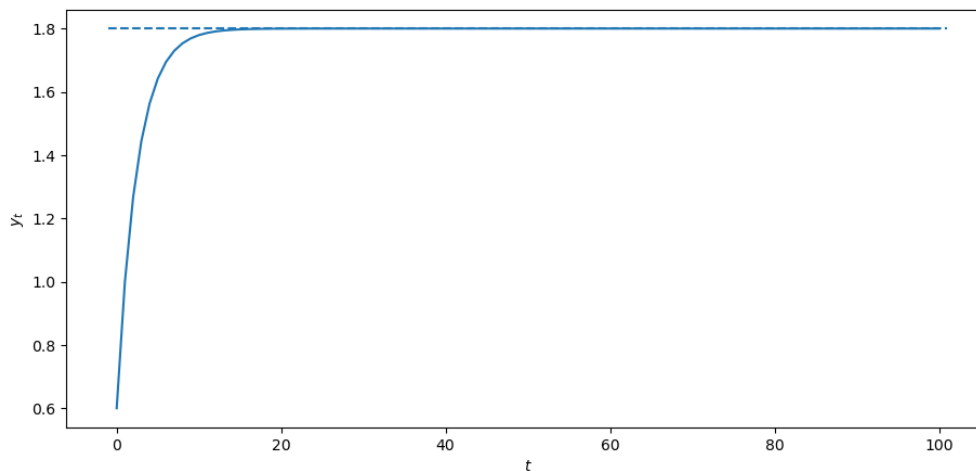

Fig. 10.5: Path of aggregate output tver time

In this model, income grows over time, until it gradually converges to the infinite geometric series sum of income.

We now examine what will happen if we vary the so-called **marginal propensity to consume**, i.e., the fraction of income that is consumed

```
bs = (1/3, 2/3, 5/6, 0.9)

fig,ax = plt.subplots()
ax.set_ylabel('$y_t$')
ax.set_xlabel('$t$')
x = np.arange(0, T+1)
for b in bs:
    y = calculate_y(i_0, b, g_0, T, y_init)
    ax.plot(x, y, label=r'$b=$'+f"{b:.2f}")
ax.legend()
plt.show()
```
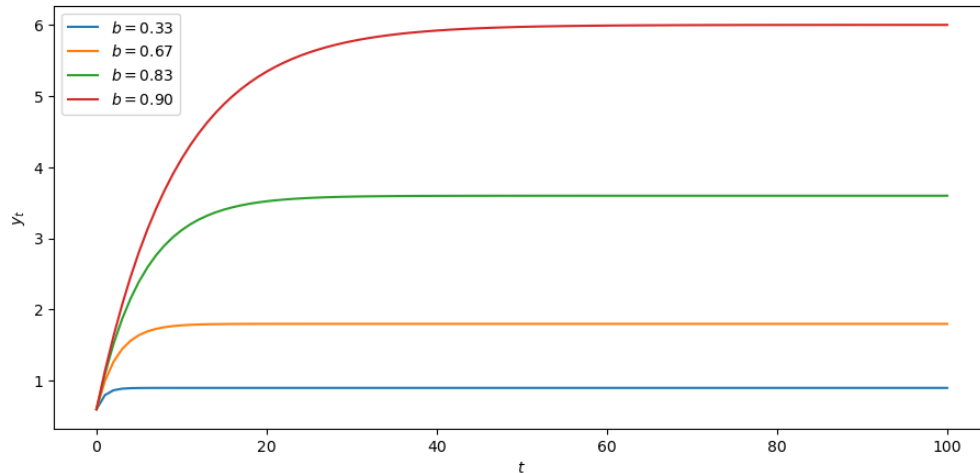
Fig. 10.6: Changing consumption as a fraction of income

Increasing the marginal propensity to consume b increases the path of output over time.

Now we will compare the effects on output of increases in investment and government spending.

```python
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(6, 10))
fig.subplots_adjust(hspace=0.3)

x = np.arange(0, T+1)
values = [0.3, 0.4]

for i in values:
    y = calculate_y(i, b, g_0, T, y_init)
    ax1.plot(x, y, label=f"i={i}")
for g in values:
    y = calculate_y(i_0, b, g, T, y_init)
    ax2.plot(x, y, label=f"g={g}")

axes = ax1, ax2
param_labels = "Investment", "Government Spending"
for ax, param in zip(axes, param_labels):
    ax.set_title(f'An Increase in {param} on Output')
    ax.legend(loc ="lower right")
    ax.set_ylabel('$y_t$')
    ax.set_xlabel('$t$')
plt.show()
```

Notice here, whether government spending increases from 0.3 to 0.4 or investment increases from 0.3 to 0.4, the shifts
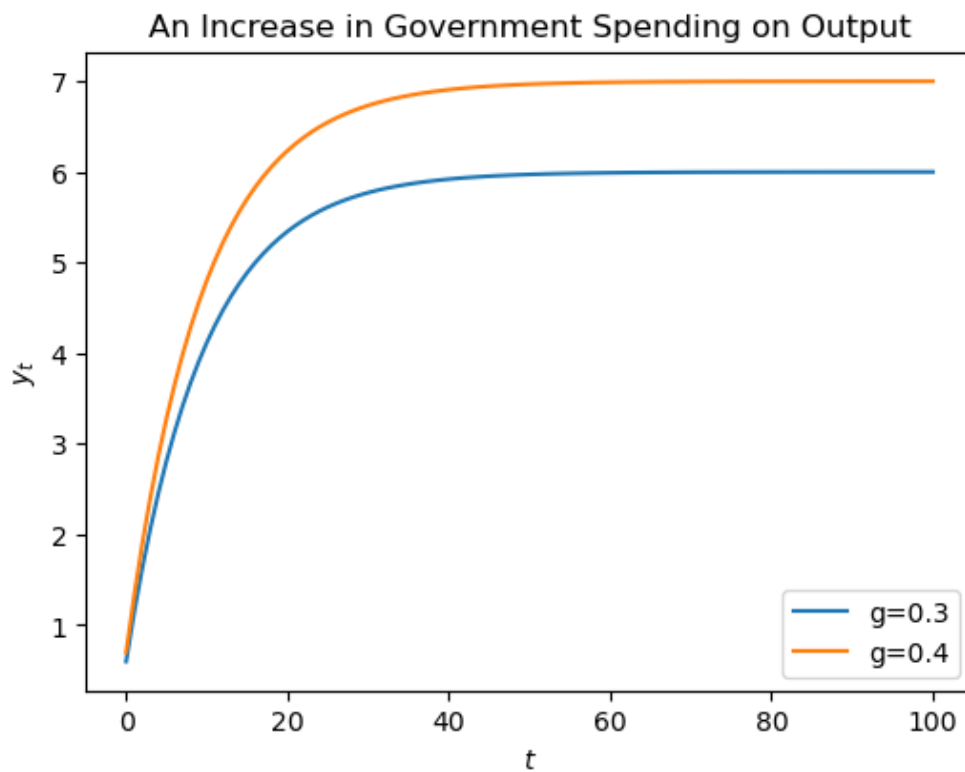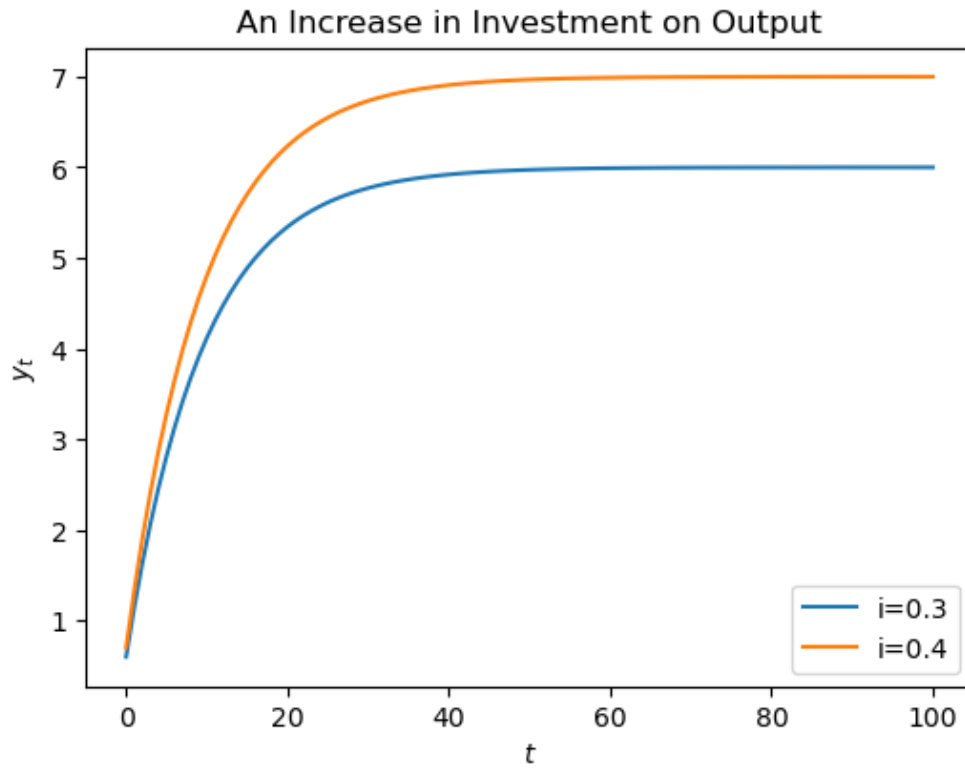
Fig. 10.7: Different increase on output

in the graphs are identical.

Part IV

Linear Dynamics: Finite Horizons

PRESENT VALUES

11.1 Overview

This lecture describes the **present value model** that is a starting point of much asset pricing theory.

Asset pricing theory is a component of theories about many economic decisions including

- consumption

- labor supply

- education choice

- demand for money

In asset pricing theory, and in economic dynamics more generally, a basic topic is the relationship among different **time series**.

A **time series** is a **sequence** indexed by time.

In this lecture, we'll represent a sequence as a vector.

So our analysis will typically boil down to studying relationships among vectors.

Our main tools in this lecture will be

- matrix multiplication, and

- matrix inversion.

We'll use the calculations described here in subsequent lectures, including *consumption smoothing*, *equalizing difference model*, and *monetarist theory of price levels*.

Let's dive in.

11.2 Analysis

Let

- $\{d_t\}_{t=0}^{T}$ be a sequence of dividends or "payouts"

- $\{p_t\}_{t=0}^{T}$ be a sequence of prices of a claim on the continuation of the asset's payout stream from date t on, namely, $\{d_s\}_{s=t}^{T}$

- $\delta \in (0, 1)$ be a one-period "discount factor"

- p_{T+1}^{*} be a terminal price of the asset at time $T + 1$

We assume that the dividend stream $\{d_t\}_{t=0}^{T}$ and the terminal price p_{T+1}^{*} are both exogenous.

This means that they are determined outside the model.

Assume the sequence of asset pricing equations

$$p_t = d_t + \delta p_{t+1}, \quad t = 0, 1, \dots, T \tag{11.1}$$

We say equation**s**, plural, because there are $T + 1$ equations, one for each $t = 0, 1, \dots, T$.

Equations (11.1) assert that price paid to purchase the asset at time t equals the payout d_t plus the price at time $t + 1$ multiplied by a time discount factor δ.

Discounting tomorrow's price by multiplying it by δ accounts for the "value of waiting one period".

We want to solve the system of $T + 1$ equations (11.1) for the asset price sequence $\{p_t\}_{t=0}^{T}$ as a function of the dividend sequence $\{d_t\}_{t=0}^{T}$ and the exogenous terminal price p_{T+1}^{*}.

A system of equations like (11.1) is an example of a linear **difference equation**.

There are powerful mathematical methods available for solving such systems and they are well worth studying in their own right, being the foundation for the analysis of many interesting economic models.

For an example, see Samuelson multiplier-accelerator

In this lecture, we'll solve system (11.1) using matrix multiplication and matrix inversion, basic tools from linear algebra introduced in *linear equations and matrix algebra*.

We will use the following imports

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

11.3 Representing sequences as vectors

The equations in system (11.1) can be arranged as follows:

$$p_0 = d_0 + \delta p_1$$
$$p_1 = d_1 + \delta p_2$$
$$\vdots \qquad\qquad\qquad (11.2)$$
$$p_{T-1} = d_{T-1} + \delta p_T$$
$$p_T = d_T + \delta p^*_{T+1}$$

Write the system (11.2) of $T + 1$ asset pricing equations as the single matrix equation

$$
\begin{bmatrix}
1 & -\delta & 0 & 0 & \cdots & 0 & 0 \\
0 & 1 & -\delta & 0 & \cdots & 0 & 0 \\
0 & 0 & 1 & -\delta & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & 0 & 0 \\
0 & 0 & 0 & 0 & \cdots & 1 & -\delta \\
0 & 0 & 0 & 0 & \cdots & 0 & 1
\end{bmatrix}
\begin{bmatrix}
p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{T-1} \\ p_T
\end{bmatrix}
=
\begin{bmatrix}
d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{T-1} \\ d_T
\end{bmatrix}
+
\begin{bmatrix}
0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ \delta p^*_{T+1}
\end{bmatrix}
\qquad (11.3)
$$

Exercise 11.3.1

Carry out the matrix multiplication in *(11.3.2)* by hand and confirm that you recover the equations in *(11.3.1)*.

In vector-matrix notation, we can write system (11.3) as

$$Ap = d + b \qquad (11.4)$$

Here A is the matrix on the left side of equation (11.3), while

$$
p = \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_T \end{bmatrix}, \quad
d = \begin{bmatrix} d_0 \\ d_1 \\ \vdots \\ d_T \end{bmatrix}, \quad \text{and} \quad
b = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ p^*_{T+1} \end{bmatrix}
$$

The solution for the vector of prices is

$$p = A^{-1}(d + b) \qquad (11.5)$$

For example, suppose that the dividend stream is

$$d_{t+1} = 1.05 d_t, \quad t = 0, 1, \dots, T - 1.$$

Let's write Python code to compute and plot the dividend stream.

```
T = 6
current_d = 1.0
d = []
for t in range(T+1):
    d.append(current_d)
    current_d = current_d * 1.05

fig, ax = plt.subplots()
ax.plot(d, 'o', label='dividends')
ax.legend()
ax.set_xlabel('time')
plt.show()
```
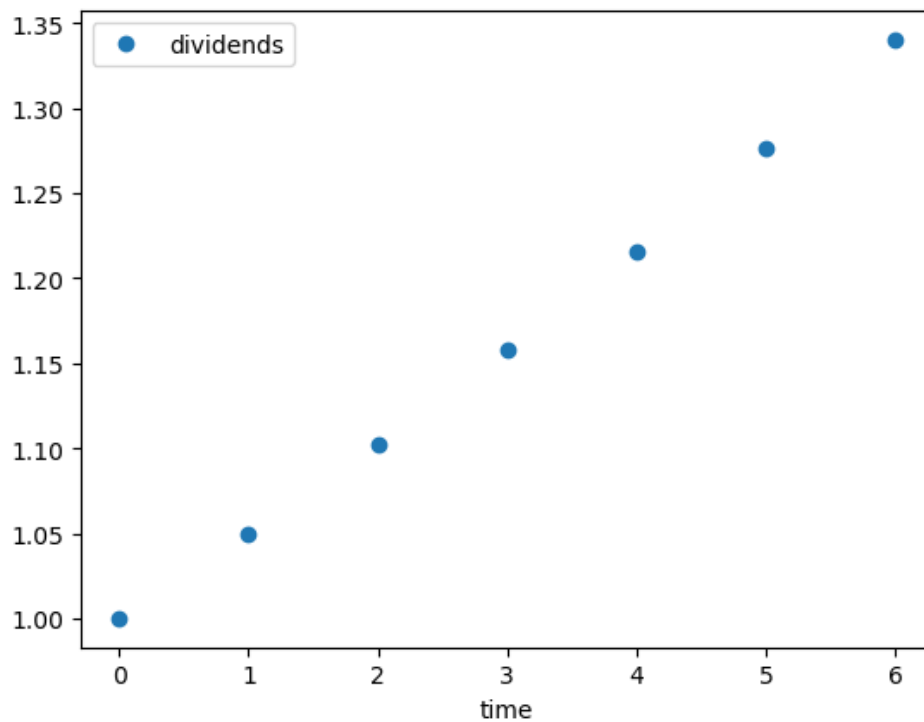

Now let's compute and plot the asset price.

We set δ and p^*_{T+1} to

```
δ = 0.99
p_star = 10.0
```

Let's build the matrix A

```
A = np.zeros((T+1, T+1))
for i in range(T+1):
```

```
    for j in range(T+1):
        if i == j:
            A[i, j] = 1
            if j < T:
                A[i, j+1] = -δ
```

Let's inspect A

```
A
```

```
array([[ 1.  , -0.99,  0.  ,  0.  ,  0.  ,  0.  ,  0.  ],
       [ 0.  ,  1.  , -0.99,  0.  ,  0.  ,  0.  ,  0.  ],
       [ 0.  ,  0.  ,  1.  , -0.99,  0.  ,  0.  ,  0.  ],
       [ 0.  ,  0.  ,  0.  ,  1.  , -0.99,  0.  ,  0.  ],
       [ 0.  ,  0.  ,  0.  ,  0.  ,  1.  , -0.99,  0.  ],
       [ 0.  ,  0.  ,  0.  ,  0.  ,  0.  ,  1.  , -0.99],
       [ 0.  ,  0.  ,  0.  ,  0.  ,  0.  ,  0.  ,  1.  ]])
```

Now let's solve for prices using (11.5).

```
b = np.zeros(T+1)
b[-1] = δ * p_star
p = np.linalg.solve(A, d + b)
fig, ax = plt.subplots()
ax.plot(p, 'o', label='asset price')
ax.legend()
ax.set_xlabel('time')
plt.show()
```
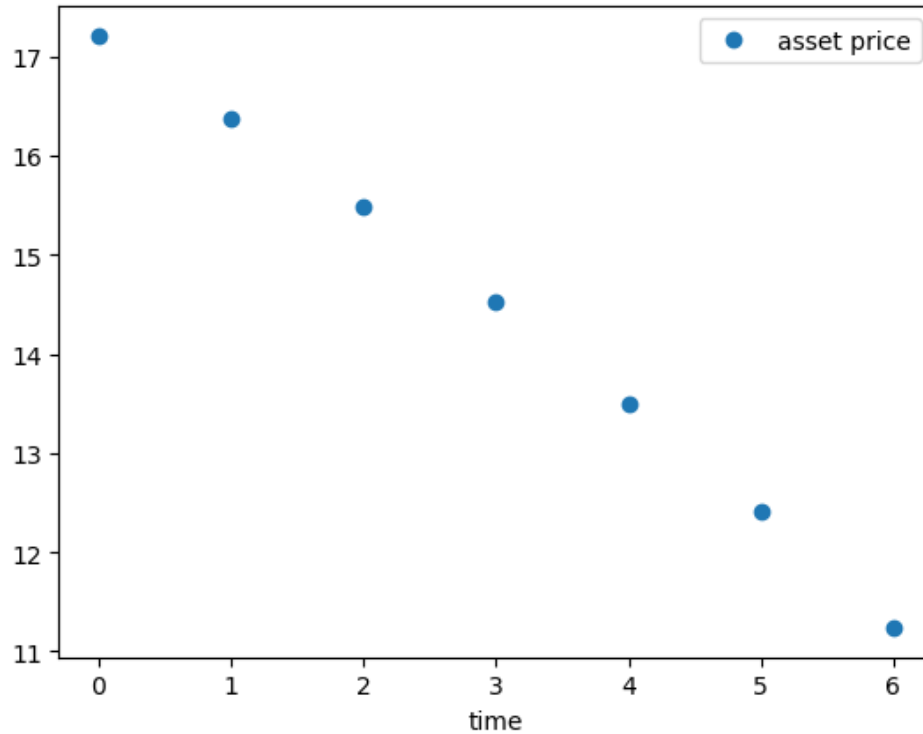
Now let's consider a cyclically growing dividend sequence:

$$d_{t+1} = 1.01 d_t + 0.1 \sin t, \quad t = 0, 1, \dots, T - 1.$$

```
T = 100
current_d = 1.0
d = []
for t in range(T+1):
    d.append(current_d)
    current_d = current_d * 1.01 + 0.1 * np.sin(t)

fig, ax = plt.subplots()
ax.plot(d, 'o-', ms=4, alpha=0.8, label='dividends')
ax.legend()
ax.set_xlabel('time')
plt.show()
```
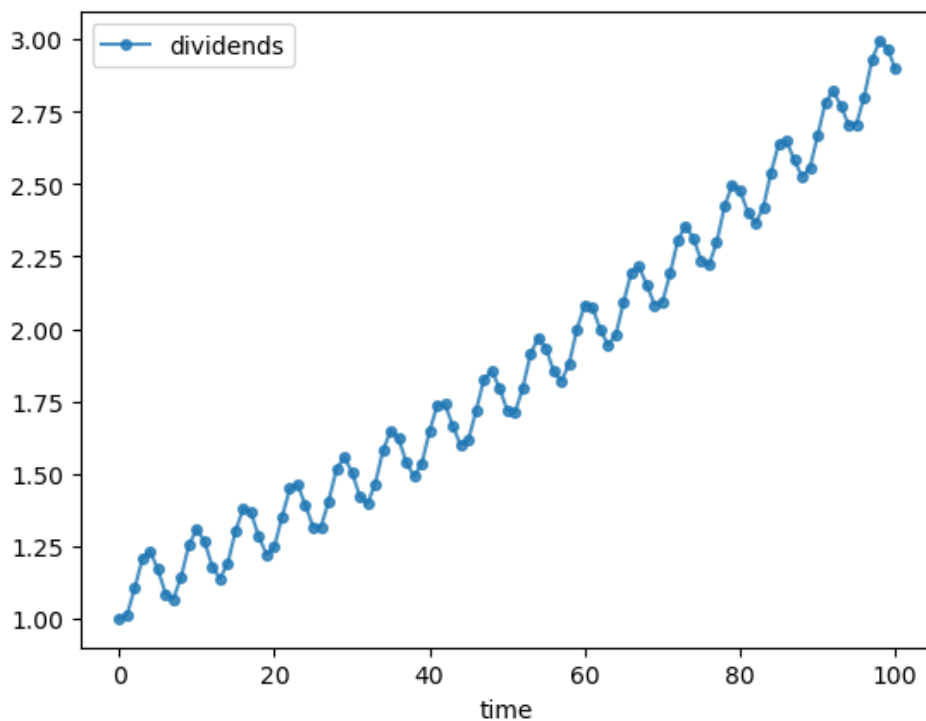
Exercise 11.3.2

Compute the corresponding asset price sequence when $p^*_{T+1} = 0$ and $\delta = 0.98$.

Solution to Exercise 11.3.2

We proceed as above after modifying parameters and consequently the matrix A.

```
δ = 0.98
p_star = 0.0
A = np.zeros((T+1, T+1))
for i in range(T+1):
    for j in range(T+1):
        if i == j:
            A[i, j] = 1
            if j < T:
                A[i, j+1] = -δ

b = np.zeros(T+1)
b[-1] = δ * p_star
p = np.linalg.solve(A, d + b)
fig, ax = plt.subplots()
```

```
ax.plot(p, 'o-', ms=4, alpha=0.8, label='asset price')
ax.legend()
ax.set_xlabel('time')
plt.show()
```
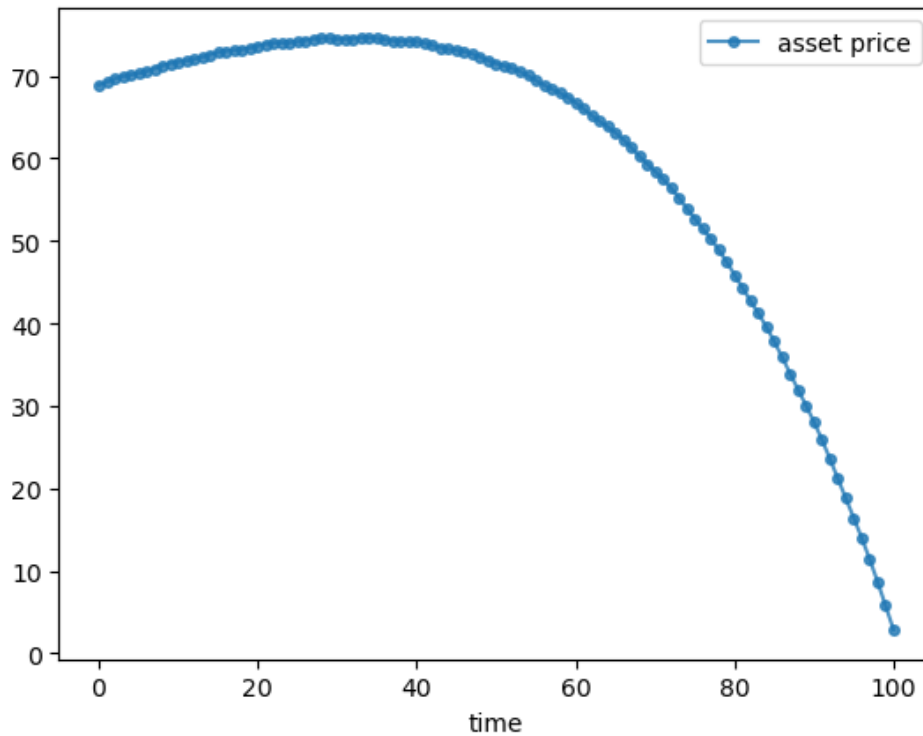

The weighted averaging associated with the present value calculation largely eliminates the cycles.

11.4 Analytical expressions

By the inverse matrix theorem, a matrix B is the inverse of A whenever AB is the identity.

It can be verified that the inverse of the matrix A in (11.3) is

$$A^{-1} = \begin{bmatrix} 1 & \delta & \delta^2 & \cdots & \delta^{T-1} & \delta^T \\ 0 & 1 & \delta & \cdots & \delta^{T-2} & \delta^{T-1} \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & \delta \\ 0 & 0 & 0 & \cdots & 0 & 1 \end{bmatrix} \tag{11.6}$$

Exercise 11.4.1

Check this by showing that AA^{-1} is equal to the identity matrix.

If we use the expression (11.6) in (11.5) and perform the indicated matrix multiplication, we shall find that

$$p_t = \sum_{s=t}^{T} \delta^{s-t} d_s + \delta^{T+1-t} p_{T+1}^* \tag{11.7}$$

Pricing formula (11.7) asserts that two components sum to the asset price p_t:

- a **fundamental component** $\sum_{s=t}^{T} \delta^{s-t} d_s$ that equals the **discounted present value** of prospective dividends

- a **bubble component** $\delta^{T+1-t} p_{T+1}^*$

The fundamental component is pinned down by the discount factor δ and the payout of the asset (in this case, dividends).

The bubble component is the part of the price that is not pinned down by fundamentals.

It is sometimes convenient to rewrite the bubble component as

$$c\delta^{-t}$$

where

$$c \equiv \delta^{T+1} p_{T+1}^*$$

11.5 More about bubbles

For a few moments, let's focus on the special case of an asset that never pays dividends, in which case

$$
\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{T-1} \\ d_T \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}
$$

In this case system (11.1) of our $T+1$ asset pricing equations takes the form of the single matrix equation

$$
\begin{bmatrix} 1 & -\delta & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & -\delta & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & -\delta & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 1 & -\delta \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{T-1} \\ p_T \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ \delta p_{T+1}^* \end{bmatrix} \tag{11.8}
$$

Evidently, if $p_{T+1}^* = 0$, a price vector p of all entries zero solves this equation and the only the **fundamental** component of our pricing formula (11.7) is present.

But let's activate the **bubble** component by setting

$$p^*_{T+1} = c\delta^{-(T+1)} \tag{11.9}$$

for some positive constant c.

In this case, when we multiply both sides of (11.8) by the matrix A^{-1} presented in equation (11.6), we find that

$$p_t = c\delta^{-t} \tag{11.10}$$

11.6 Gross rate of return

Define the gross rate of return on holding the asset from period t to period $t+1$ as

$$R_t = \frac{p_{t+1}}{p_t} \tag{11.11}$$

Substituting equation (11.10) into equation (11.11) confirms that an asset whose sole source of value is a bubble earns a gross rate of return

$$R_t = \delta^{-1} > 1, t = 0, 1, \dots, T$$

11.7 Exercises

Exercise 11.7.1

Give analytical expressions for an asset price p_t under the following settings for d and p^*_{T+1}:

1. $p^*_{T+1} = 0, d_t = g^t d_0$ (a modified version of the Gordon growth formula)

2. $p^*_{T+1} = g^{T+1} d_0, d_t = g^t d_0$ (the plain vanilla Gordon growth formula)

3. $p^*_{T+1} = 0, d_t = 0$ (price of a worthless stock)

4. $p^*_{T+1} = c\delta^{-(T+1)}, d_t = 0$ (price of a pure bubble stock)

Solution to Exercise 11.7.1

Plugging each pair of the above p^*_{T+1}, d_t into Equation (11.7) yields:

1. $p_t = \sum_{s=t}^{T} \delta^{s-t} g^s d_0$

2. $p_t = \sum_{s=t}^{T} \delta^{s-t} g^s d_0 + \delta^{T+1-t} g^{T+1} d_0$

3. $p_t = 0$

4. $p_t = c\delta^{-t}$

CONSUMPTION SMOOTHING

12.1 Overview

In this lecture, we'll study a famous model of the "consumption function" that Milton Friedman [Friedman, 1956] and Robert Hall [Hall, 1978]) proposed to fit some empirical data patterns that the original Keynesian consumption function described in this QuantEcon lecture *geometric series* missed.

In this lecture, we'll study what is often called the "consumption-smoothing model" using matrix multiplication and matrix inversion, the same tools that we used in this QuantEcon lecture *present values*.

Formulas presented in *present value formulas* are at the core of the consumption-smoothing model because we shall use them to define a consumer's "human wealth".

The key idea that inspired Milton Friedman was that a person's non-financial income, i.e., his or her wages from working, could be viewed as a dividend stream from that person's "human capital" and that standard asset-pricing formulas could be applied to compute a person's "non-financial wealth" that capitalizes the earnings stream.

Note: As we'll see in this QuantEcon lecture *equalizing difference model*, Milton Friedman had used this idea in his PhD thesis at Columbia University, eventually published as [Kuznets and Friedman, 1939] and [Friedman and Kuznets, 1945].

It will take a while for a "present value" or asset price explicilty to appear in this lecture, but when it does it will be a key actor.

12.2 Analysis

As usual, we'll start by importing some Python modules.

```python
import numpy as np
import matplotlib.pyplot as plt
from collections import namedtuple
```

The model describes a consumer who lives from time $t = 0, 1, \ldots, T$, receives a stream $\{y_t\}_{t=0}^{T}$ of non-financial income and chooses a consumption stream $\{c_t\}_{t=0}^{T}$.

We usually think of the non-financial income stream as coming from the person's salary from supplying labor.

The model takes a non-financial income stream as an input, regarding it as "exogenous" in the sense of not being determined by the model.

The consumer faces a gross interest rate of $R > 1$ that is constant over time, at which she is free to borrow or lend, up to limits that we'll describe below.

To set up the model, let

- $T \geq 2$ be a positive integer that constitutes a time-horizon.

- $y = \{y_t\}_{t=0}^{T}$ be an exogenous sequence of non-negative non-financial incomes y_t.

- $a = \{a_t\}_{t=0}^{T+1}$ be a sequence of financial wealth.

- $c = \{c_t\}_{t=0}^{T}$ be a sequence of non-negative consumption rates.

- $R \geq 1$ be a fixed gross one period rate of return on financial assets.

- $\beta \in (0, 1)$ be a fixed discount factor.

- a_0 be a given initial level of financial assets

- $a_{T+1} \geq 0$ be a terminal condition on final assets.

The sequence of financial wealth a is to be determined by the model.

We require it to satisfy two **boundary conditions**:

- it must equal an exogenous value a_0 at time 0

- it must equal or exceed an exogenous value a_{T+1} at time $T + 1$.

The **terminal condition** $a_{T+1} \geq 0$ requires that the consumer not leave the model in debt.

(We'll soon see that a utility maximizing consumer won't want to die leaving positive assets, so she'll arrange her affairs to make $a_{T+1} = 0$.)

The consumer faces a sequence of budget constraints that constrains sequences (y, c, a)

$$a_{t+1} = R(a_t + y_t - c_t), \quad t = 0, 1, \dots T \tag{12.1}$$

Equations (12.1) constitute $T + 1$ such budget constraints, one for each $t = 0, 1, \dots, T$.

Given a sequence y of non-financial incomes, a large set of pairs (a, c) of (financial wealth, consumption) sequences satisfy the sequence of budget constraints (12.1).

Our model has the following logical flow.

- start with an exogenous non-financial income sequence y, an initial financial wealth a_0, and a candidate consumption path c.

- use the system of equations (12.1) for $t = 0, \dots, T$ to compute a path a of financial wealth

- verify that a_{T+1} satisfies the terminal wealth constraint $a_{T+1} \geq 0$.

    - If it does, declare that the candidate path is **budget feasible**.

    - if the candidate consumption path is not budget feasible, propose a less greedy consumption path and start over

Below, we'll describe how to execute these steps using linear algebra –matrix inversion and multiplication.

The above procedure seems like a sensible way to find "budget-feasible" consumption paths c, i.e., paths that are consistent with the exogenous non-financial income stream y, the initial financial asset level a_0, and the terminal asset level a_{T+1}.

In general, there are **many** budget feasible consumption paths c.

Among all budget-feasible consumption paths, which one should a consumer want?

To answer this question, we shall eventually evaluate alternative budget feasible consumption paths c using the following utility functional or **welfare criterion**:

$$W = \sum_{t=0}^{T} \beta^t (g_1 c_t - \frac{g_2}{2} c_t^2) \tag{12.2}$$

where $g_1 > 0, g_2 > 0$.

When $\beta R \approx 1$, the fact that the utility function $g_1 c_t - \frac{g_2}{2} c_t^2$ has diminishing marginal utility imparts a preference for consumption that is very smooth.

Indeed, we shall see that when $\beta R = 1$ (a condition assumed by Milton Friedman [Friedman, 1956] and Robert Hall [Hall, 1978]), criterion (12.2) assigns higher welfare to smoother consumption paths.

By **smoother** we mean as close as possible to being constant over time.

The preference for smooth consumption paths that is built into the model gives it the name "consumption-smoothing model".

Let's dive in and do some calculations that will help us understand how the model works.

Here we use default parameters $R = 1.05$, $g_1 = 1$, $g_2 = 1/2$, and $T = 65$.

We create a Python **namedtuple** to store these parameters with default values.

```
ConsumptionSmoothing = namedtuple("ConsumptionSmoothing",
                     ["R", "g1", "g2", "β_seq", "T"])


def create_consumption_smoothing_model(R=1.05, g1=1, g2=1/2, T=65):
    β = 1/R
    β_seq = np.array([β**i for i in range(T+1)])
    return ConsumptionSmoothing(R, g1, g2,
                                β_seq, T)
```

12.3 Friedman-Hall consumption-smoothing model

A key object is what Milton Friedman called "human" or "non-financial" wealth at time 0:

$$h_0 \equiv \sum_{t=0}^{T} R^{-t} y_t = \begin{bmatrix} 1 & R^{-1} & \cdots & R^{-T} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_T \end{bmatrix}$$

Human or non-financial wealth at time 0 is evidently just the present value of the consumer's non-financial income stream y.

Formally it very much resembles the asset price that we computed in this QuantEcon lecture *present values*.

Indeed, this is why Milton Friedman called it "human capital".

By iterating on equation (12.1) and imposing the terminal condition

$$a_{T+1} = 0,$$

it is possible to convert a sequence of budget constraints (12.1) into a single intertemporal constraint

$$\sum_{t=0}^{T} R^{-t} c_t = a_0 + h_0. \tag{12.3}$$

Equation (12.3) says that the present value of the consumption stream equals the sum of financial and non-financial (or human) wealth.

Robert Hall [Hall, 1978] showed that when $\beta R = 1$, a condition Milton Friedman had also assumed, it is "optimal" for a consumer to smooth consumption by setting

$$c_t = c_0 \quad t = 0, 1, \ldots, T$$

(Later we'll present a "variational argument" that shows that this constant path maximizes criterion (12.2) when $\beta R = 1$.)

In this case, we can use the intertemporal budget constraint to write

$$c_t = c_0 = \left(\sum_{t=0}^{T} R^{-t} \right)^{-1} (a_0 + h_0), \quad t = 0, 1, \dots, T. \tag{12.4}$$

Equation (12.4) is the consumption-smoothing model in a nutshell.

12.4 Mechanics of consumption-smoothing model

As promised, we'll provide step-by-step instructions on how to use linear algebra, readily implemented in Python, to compute all objects in play in the consumption-smoothing model.

In the calculations below, we'll set default values of $R > 1$, e.g., $R = 1.05$, and $\beta = R^{-1}$.

12.4.1 Step 1

For a $(T+1) \times 1$ vector y, use matrix algebra to compute h_0

$$h_0 = \sum_{t=0}^{T} R^{-t} y_t = \begin{bmatrix} 1 & R^{-1} & \cdots & R^{-T} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_T \end{bmatrix}$$

12.4.2 Step 2

Compute an time 0 consumption c_0 :

$$c_t = c_0 = \left(\frac{1 - R^{-1}}{1 - R^{-(T+1)}} \right) \left(a_0 + \sum_{t=0}^{T} R^{-t} y_t \right), \quad t = 0, 1, \dots, T$$

12.4.3 Step 3

Use the system of equations (12.1) for $t = 0, \dots, T$ to compute a path a of financial wealth.

To do this, we translate that system of difference equations into a single matrix equation as follows:

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -R & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & -R & 1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & -R & 1 & 0 \\ 0 & 0 & 0 & \cdots & 0 & -R & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_T \\ a_{T+1} \end{bmatrix} = R \begin{bmatrix} y_0 + a_0 - c_0 \\ y_1 - c_0 \\ y_2 - c_0 \\ \vdots \\ y_{T-1} - c_0 \\ y_T - c_0 \end{bmatrix}$$

Multiply both sides by the inverse of the matrix on the left side to compute

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_T \\ a_{T+1} \end{bmatrix}$$

Because we have built into our calculations that the consumer leaves the model with exactly zero assets, just barely satisfying the terminal condition that $a_{T+1} \geq 0$, it should turn out that

$$a_{T+1} = 0.$$

Let's verify this with Python code.

First we implement the model with `compute_optimal`

```python
def compute_optimal(model, a0, y_seq):
    R, T = model.R, model.T

    # non-financial wealth
    h0 = model.β_seq @ y_seq      # since β = 1/R

    # c0
    c0 = (1 - 1/R) / (1 - (1/R)**(T+1)) * (a0 + h0)
    c_seq = c0*np.ones(T+1)

    # verify
    A = np.diag(-R*np.ones(T), k=-1) + np.eye(T+1)
    b = y_seq - c_seq
    b[0] = b[0] + a0

    a_seq = np.linalg.inv(A) @ b
    a_seq = np.concatenate([[a0], a_seq])

    return c_seq, a_seq, h0
```

We use an example where the consumer inherits $a_0 < 0$.

This can be interpreted as a student debt.

The non-financial process $\{y_t\}_{t=0}^T$ is constant and positive up to $t = 45$ and then becomes zero afterward.

The drop in non-financial income late in life reflects retirement from work.

```
# Financial wealth
a0 = -2      # such as "student debt"

# non-financial Income process
y_seq = np.concatenate([np.ones(46), np.zeros(20)])

cs_model = create_consumption_smoothing_model()
c_seq, a_seq, h0 = compute_optimal(cs_model, a0, y_seq)

print('check a_T+1=0:',
      np.abs(a_seq[-1] - 0) <= 1e-8)
```

```
check a_T+1=0: True
```

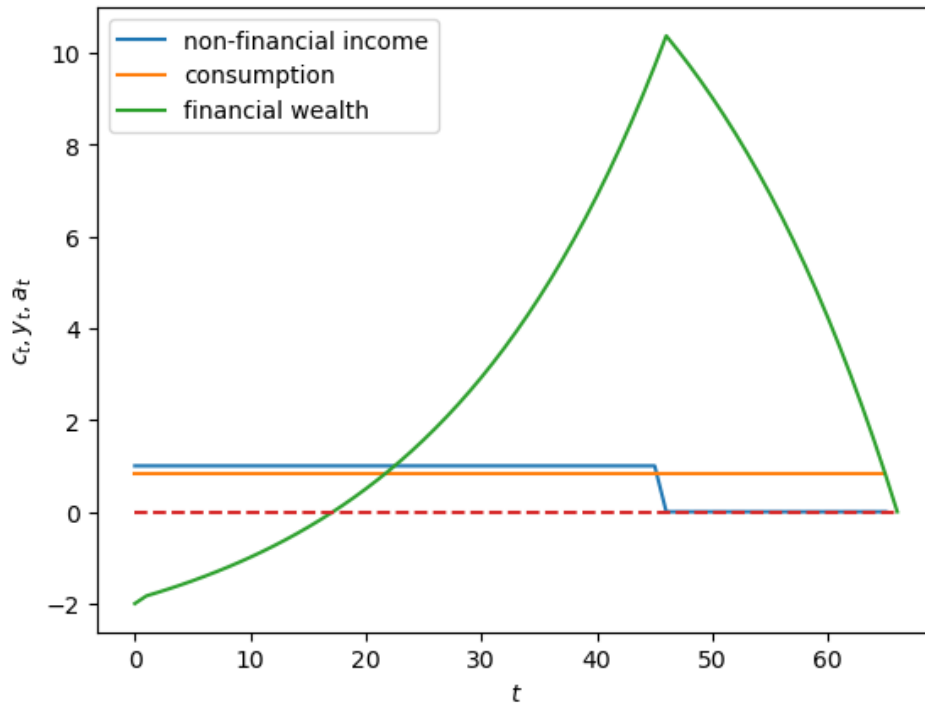The graphs below show paths of non-financial income, consumption, and financial assets.

```
# Sequence Length
T = cs_model.T

plt.plot(range(T+1), y_seq, label='non-financial income')
plt.plot(range(T+1), c_seq, label='consumption')
plt.plot(range(T+2), a_seq, label='financial wealth')
plt.plot(range(T+2), np.zeros(T+2), '--')

plt.legend()
plt.xlabel(r'$t$')
plt.ylabel(r'$c_t,y_t,a_t$')
plt.show()
```

Note that $a_{T+1} = 0$, as anticipated.

We can evaluate welfare criterion (12.2)

```python
def welfare(model, c_seq):
    β_seq, g1, g2 = model.β_seq, model.g1, model.g2

    u_seq = g1 * c_seq - g2/2 * c_seq**2
    return β_seq @ u_seq

print('Welfare:', welfare(cs_model, c_seq))
```

```
Welfare: 13.285050962183442
```

12.4.4 Experiments

In this section we describe how a consumption sequence would optimally respond to different sequences sequences of non-financial income.

First we create a function `plot_cs` that generates graphs for different instances of the consumption-smoothing model `cs_model`.

This will help us avoid rewriting code to plot outcomes for different non-financial income sequences.

```python
def plot_cs(model,    # consumption-smoothing model
            a0,       # initial financial wealth
            y_seq     # non-financial income process
           ):

    # Compute optimal consumption
    c_seq, a_seq, h0 = compute_optimal(model, a0, y_seq)

    # Sequence length
    T = cs_model.T

    # Generate plot
    plt.plot(range(T+1), y_seq, label='non-financial income')
    plt.plot(range(T+1), c_seq, label='consumption')
    plt.plot(range(T+2), a_seq, label='financial wealth')
    plt.plot(range(T+2), np.zeros(T+2), '--')

    plt.legend()
    plt.xlabel(r'$t$')
    plt.ylabel(r'$c_t,y_t,a_t$')
    plt.show()
```

In the experiments below, please study how consumption and financial asset sequences vary across different sequences for non-financial income.

Experiment 1: one-time gain/loss

We first assume a one-time windfall of W_0 in year 21 of the income sequence y.

We'll make W_0 big - positive to indicate a one-time windfall, and negative to indicate a one-time "disaster".

```python
# Windfall W_0 = 2.5
y_seq_pos = np.concatenate([np.ones(21), np.array([2.5]), np.ones(24), np.zeros(20)])

plot_cs(cs_model, a0, y_seq_pos)
```
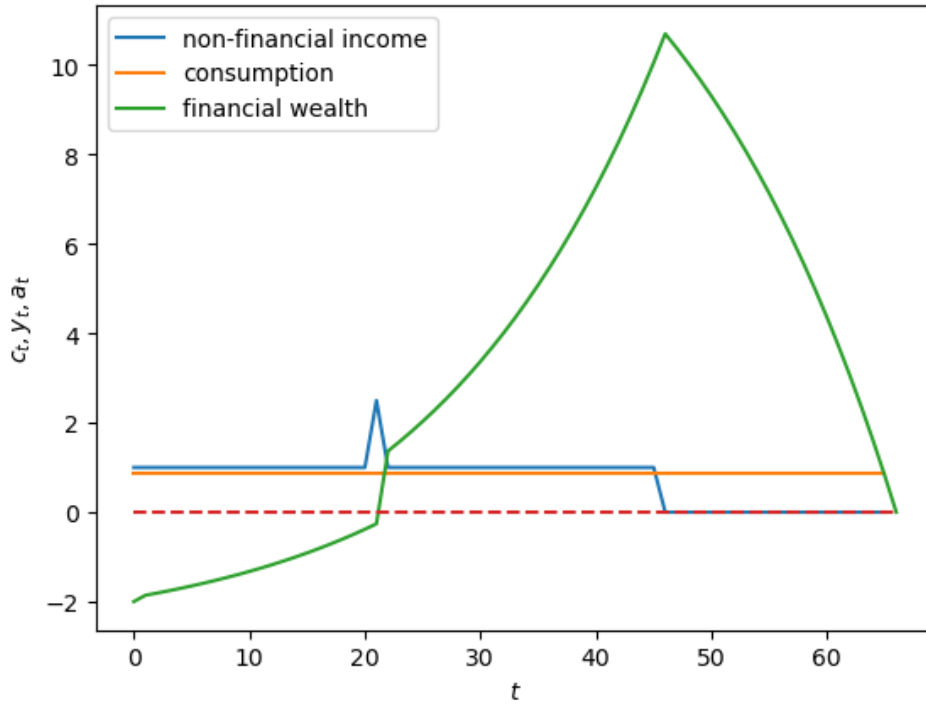
```
# Disaster W_0 = -2.5
y_seq_neg = np.concatenate([np.ones(21), np.array([-2.5]), np.ones(24), np.zeros(20)])

plot_cs(cs_model, a0, y_seq_neg)
```
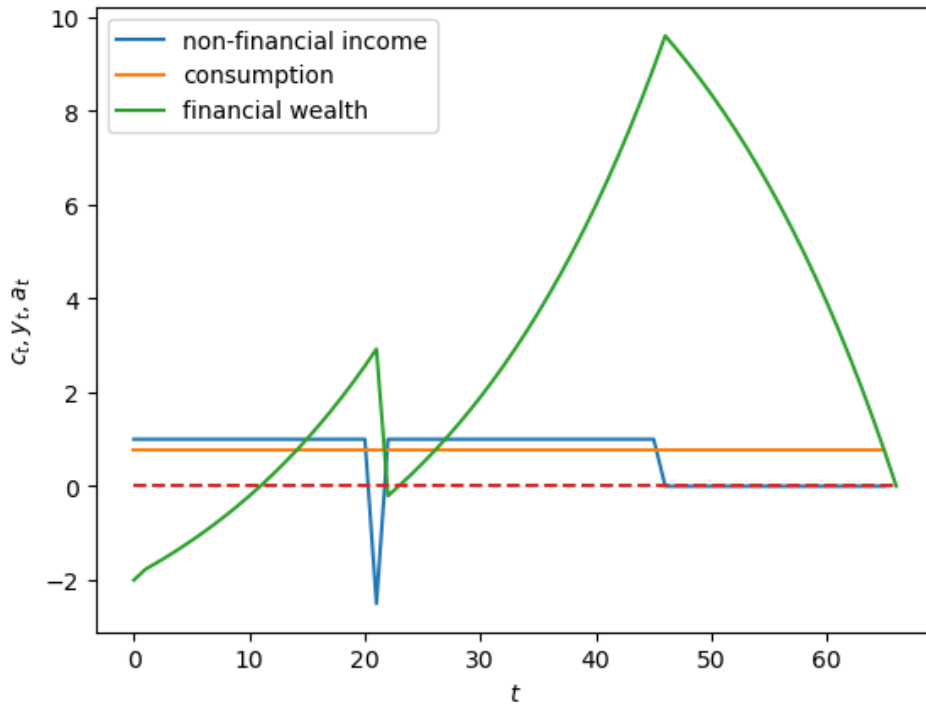
Experiment 2: permanent wage gain/loss

Now we assume a permanent increase in income of W in year 21 of the y-sequence.

Again we can study positive and negative cases

```
# Positive permanent income change W = 0.5 when t >= 21
y_seq_pos = np.concatenate(
    [np.ones(21), 1.5*np.ones(25), np.zeros(20)])

plot_cs(cs_model, a0, y_seq_pos)
```
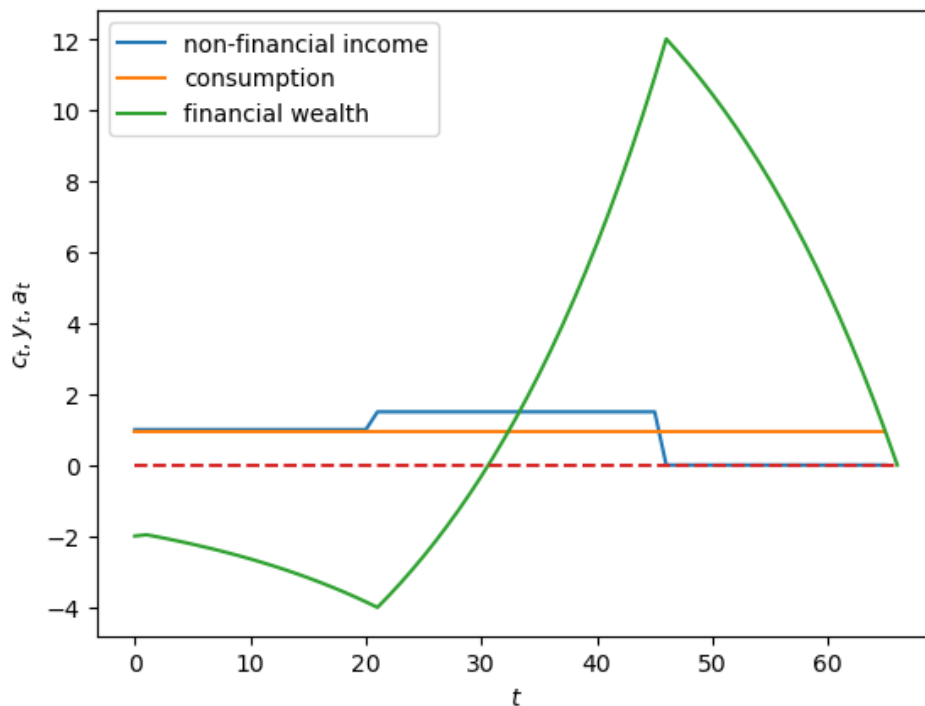


```
# Negative permanent income change W = -0.5 when t >= 21
y_seq_neg = np.concatenate(
    [np.ones(21), .5*np.ones(25), np.zeros(20)])

plot_cs(cs_model, a0, y_seq_neg)
```
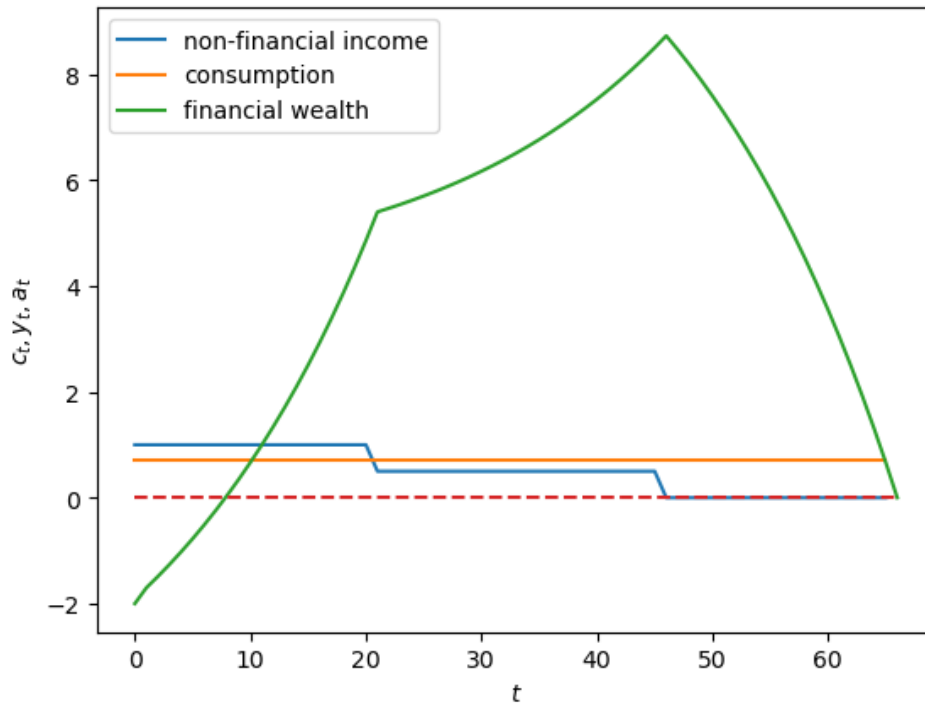
Experiment 3: a late starter

Now we simulate a y sequence in which a person gets zero for 46 years, and then works and gets 1 for the last 20 years of life (a "late starter")

```python
# Late starter
y_seq_late = np.concatenate(
    [np.zeros(46), np.ones(20)])

plot_cs(cs_model, a0, y_seq_late)
```
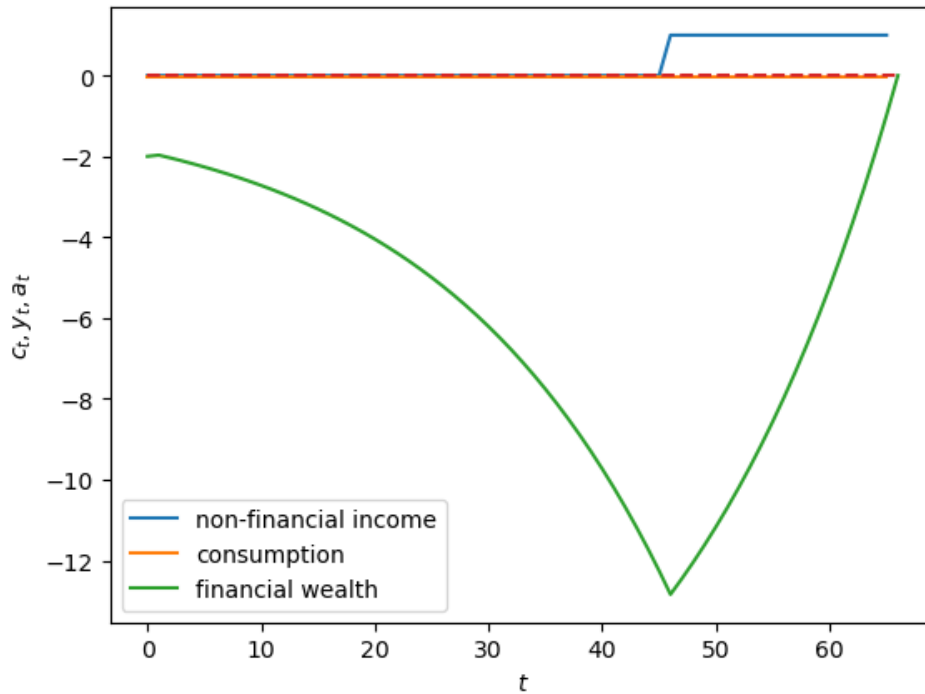
Experiment 4: geometric earner

Now we simulate a geometric y sequence in which a person gets $y_t = \lambda^t y_0$ in first 46 years.

We first experiment with $\lambda = 1.05$

```
# Geometric earner parameters where λ = 1.05
λ = 1.05
y_0 = 1
t_max = 46

# Generate geometric y sequence
geo_seq = λ ** np.arange(t_max) * y_0
y_seq_geo = np.concatenate(
            [geo_seq, np.zeros(20)])

plot_cs(cs_model, a0, y_seq_geo)
```
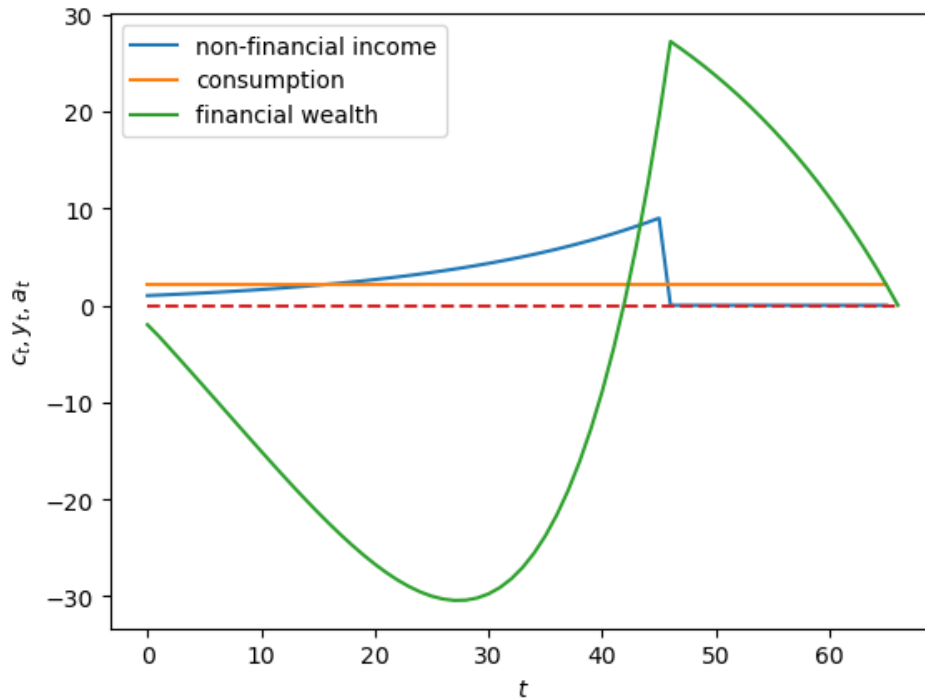
Now we show the behavior when $\lambda = 0.95$

```
λ = 0.95

geo_seq = λ ** np.arange(t_max) * y_0
y_seq_geo = np.concatenate(
            [geo_seq, np.zeros(20)])

plot_cs(cs_model, a0, y_seq_geo)
```
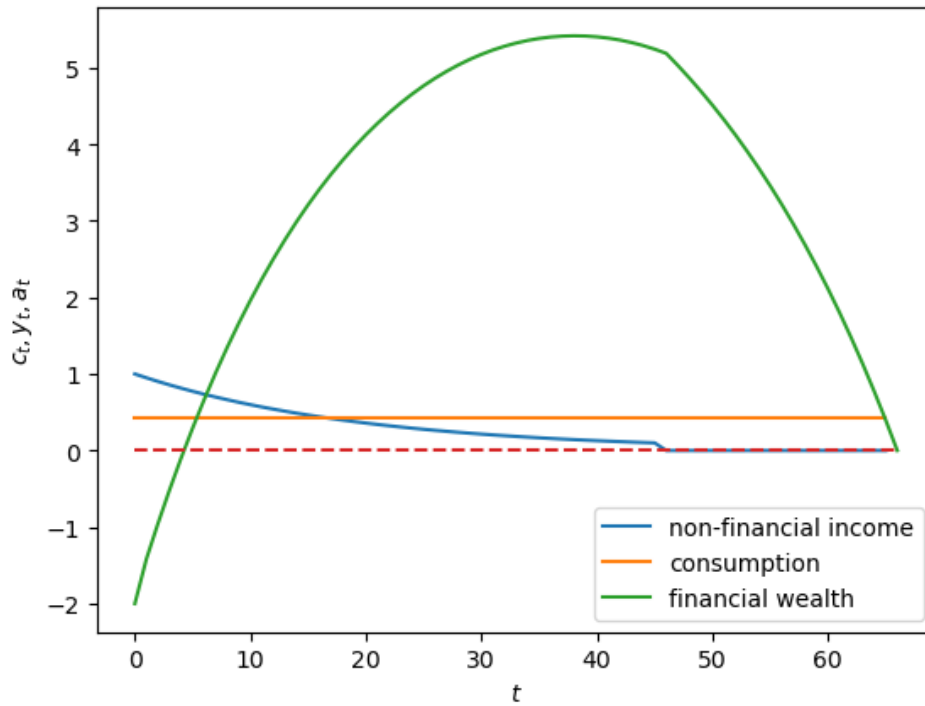
What happens when λ is negative

```
λ = -0.95

geo_seq = λ ** np.arange(t_max) * y_0
y_seq_geo = np.concatenate(
            [geo_seq, np.zeros(20)])

plot_cs(cs_model, a0, y_seq_geo)
```
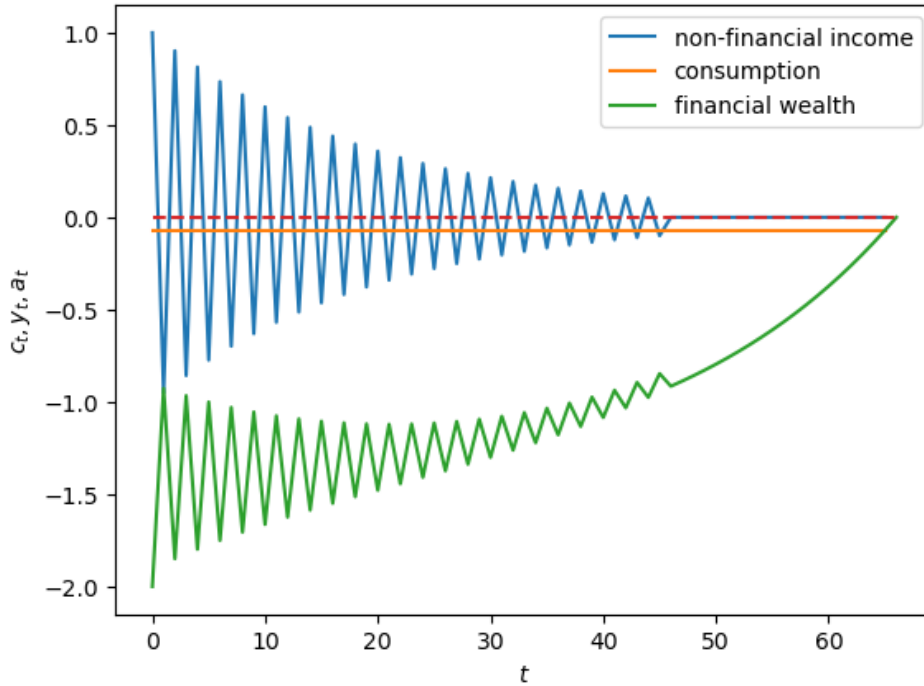
12.4.5 Feasible consumption variations

We promised to justify our claim that a constant consumption play $c_t = c_0$ for all t is optimal.

Let's do that now.

The approach we'll take is an elementary example of the "calculus of variations".

Let's dive in and see what the key idea is.

To explore what types of consumption paths are welfare-improving, we shall create an **admissible consumption path variation sequence** $\{v_t\}_{t=0}^{T}$ that satisfies

$$\sum_{t=0}^{T} R^{-t} v_t = 0$$

This equation says that the **present value** of admissible consumption path variations must be zero.

So once again, we encounter a formula for the present value of an "asset":

- we require that the present value of consumption path variations be zero.

Here we'll restrict ourselves to a two-parameter class of admissible consumption path variations of the form

$$v_t = \xi_1 \phi^t - \xi_0$$

We say two and not three-parameter class because ξ_0 will be a function of $(\phi, \xi_1; R)$ that guarantees that the variation sequence is feasible.

Let's compute that function.

We require

$$\sum_{t=0}^{T} R^{-t} \left[\xi_1 \phi^t - \xi_0 \right] = 0$$

which implies that

$$\xi_1 \sum_{t=0}^{T} \phi_t R^{-t} - \xi_0 \sum_{t=0}^{T} R^{-t} = 0$$

which implies that

$$\xi_1 \frac{1 - (\phi R^{-1})^{T+1}}{1 - \phi R^{-1}} - \xi_0 \frac{1 - R^{-(T+1)}}{1 - R^{-1}} = 0$$

which implies that

$$\xi_0 = \xi_0(\phi, \xi_1; R) = \xi_1 \left(\frac{1 - R^{-1}}{1 - R^{-(T+1)}} \right) \left(\frac{1 - (\phi R^{-1})^{T+1}}{1 - \phi R^{-1}} \right)$$

This is our formula for ξ_0.

Key Idea: if c^o is a budget-feasible consumption path, then so is $c^o + v$, where v is a budget-feasible variation.

Given R, we thus have a two parameter class of budget feasible variations v that we can use to compute alternative consumption paths, then evaluate their welfare.

Now let's compute and plot consumption path variations

```python
def compute_variation(model, ξ1, ϕ, a0, y_seq, verbose=1):
    R, T, β_seq = model.R, model.T, model.β_seq

    ξ0 = ξ1*((1 - 1/R) / (1 - (1/R)**(T+1))) * ((1 - (ϕ/R)**(T+1)) / (1 - ϕ/R))
    v_seq = np.array([(ξ1*ϕ**t - ξ0) for t in range(T+1)])

    if verbose == 1:
        print('check feasible:', np.isclose(β_seq @ v_seq, 0))    # since β = 1/R

    c_opt, _, _ = compute_optimal(model, a0, y_seq)
    cvar_seq = c_opt + v_seq

    return cvar_seq
```

We visualize variations for $\xi_1 \in \{.01, .05\}$ and $\phi \in \{.95, 1.02\}$

```python
fig, ax = plt.subplots()

ξ1s = [.01, .05]
```

```python
φs= [.95, 1.02]
colors = {.01: 'tab:blue', .05: 'tab:green'}


params = np.array(np.meshgrid(ξ1s, φs)).T.reshape(-1, 2)


for i, param in enumerate(params):
    ξ1, φ = param
    print(f'variation {i}: ξ1={ξ1}, φ={φ}')
    cvar_seq = compute_variation(model=cs_model,
                                 ξ1=ξ1, φ=φ, a0=a0,
                                 y_seq=y_seq)
    print(f'welfare={welfare(cs_model, cvar_seq)}')
    print('-'*64)
    if i % 2 == 0:
        ls = '-.'
    else:
        ls = '-'
    ax.plot(range(T+1), cvar_seq, ls=ls,
            color=colors[ξ1],
            label=fr'$\xi_1 = {ξ1}, \phi = {φ}$')

plt.plot(range(T+1), c_seq,
         color='orange', label=r'Optimal $\vec{c}$ ')

plt.legend()
plt.xlabel(r'$t$')
plt.ylabel(r'$c_t$')
plt.show()
```

```
variation 0: ξ1=0.01, φ=0.95
check feasible: True
welfare=13.28500934606484
-------------------------------------------------------------
variation 1: ξ1=0.01, φ=1.02
check feasible: True
welfare=13.284911631015444
-------------------------------------------------------------
variation 2: ξ1=0.05, φ=0.95
check feasible: True
welfare=13.284010559218514
-------------------------------------------------------------
variation 3: ξ1=0.05, φ=1.02
check feasible: True
```

```
welfare=13.28156768298361

-------------------------------------------------------------------
```
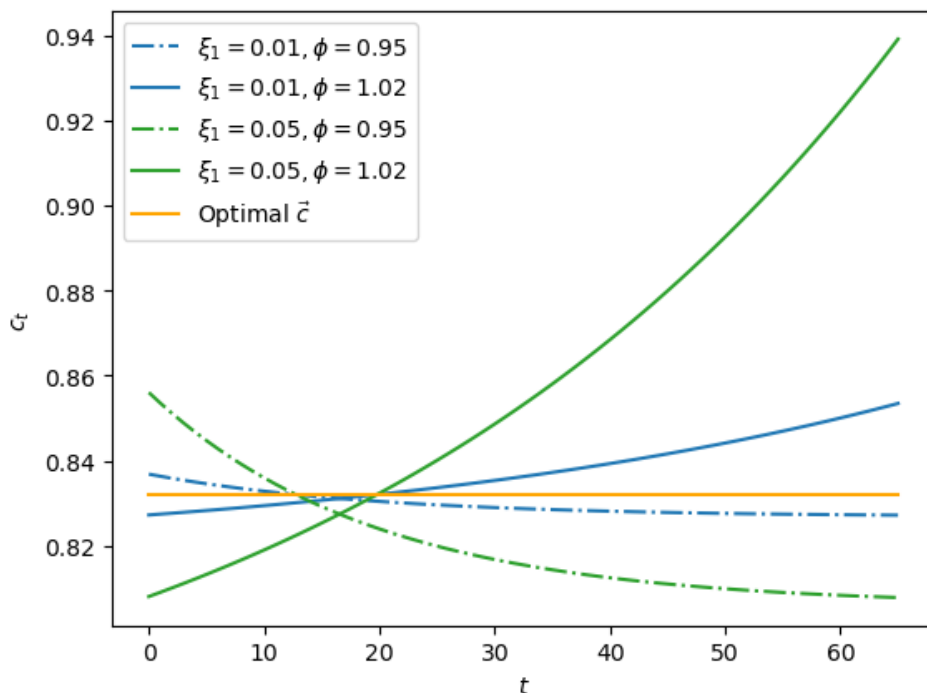

We can even use the Python `np.gradient` command to compute derivatives of welfare with respect to our two parameters.

We are teaching the key idea beneath the **calculus of variations**.

First, we define the welfare with respect to ξ_1 and ϕ

```python
def welfare_rel(ξ1, φ):
    """
    Compute welfare of variation sequence
    for given φ, ξ1 with a consumption-smoothing model
    """

    cvar_seq = compute_variation(cs_model, ξ1=ξ1,
                                 φ=φ, a0=a0,
                                 y_seq=y_seq,
                                 verbose=0)
    return welfare(cs_model, cvar_seq)

# Vectorize the function to allow array input
welfare_vec = np.vectorize(welfare_rel)
```

Then we can visualize the relationship between welfare and ξ_1 and compute its derivatives

```
ξ1_arr = np.linspace(-0.5, 0.5, 20)

plt.plot(ξ1_arr, welfare_vec(ξ1_arr, 1.02))
plt.ylabel('welfare')
plt.xlabel(r'$\xi_1$')
plt.show()

welfare_grad = welfare_vec(ξ1_arr, 1.02)
welfare_grad = np.gradient(welfare_grad)
plt.plot(ξ1_arr, welfare_grad)
plt.ylabel('derivative of welfare')
plt.xlabel(r'$\xi_1$')
plt.show()
```
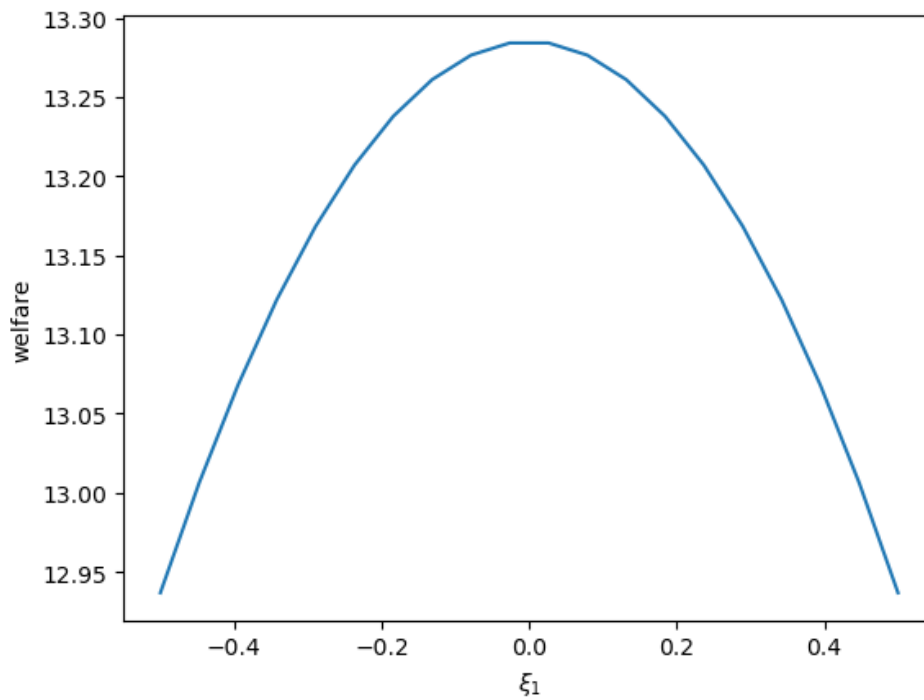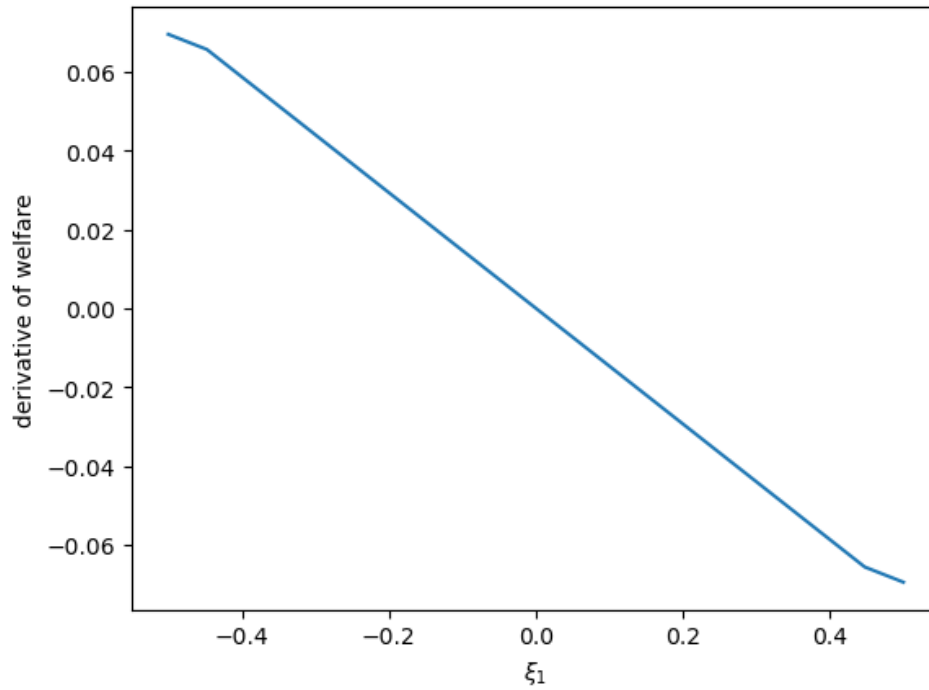
The same can be done on ϕ

```
φ_arr = np.linspace(-0.5, 0.5, 20)


plt.plot(ξ1_arr, welfare_vec(0.05, φ_arr))
plt.ylabel('welfare')
plt.xlabel(r'$\phi$')
plt.show()


welfare_grad = welfare_vec(0.05, φ_arr)
welfare_grad = np.gradient(welfare_grad)
plt.plot(ξ1_arr, welfare_grad)
plt.ylabel('derivative of welfare')
plt.xlabel(r'$\phi$')
plt.show()
```
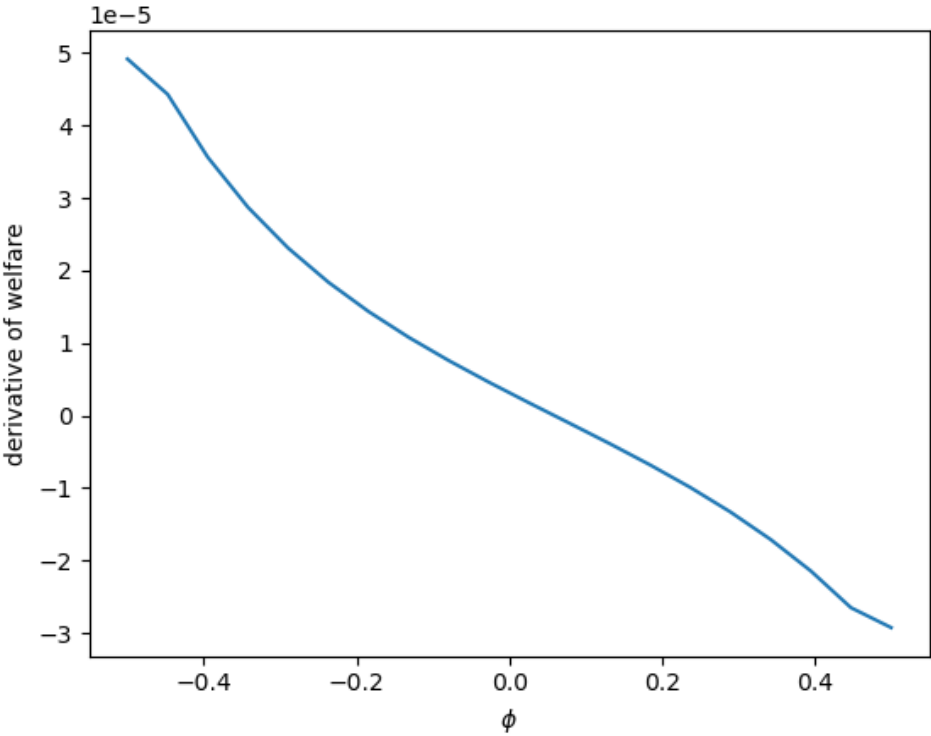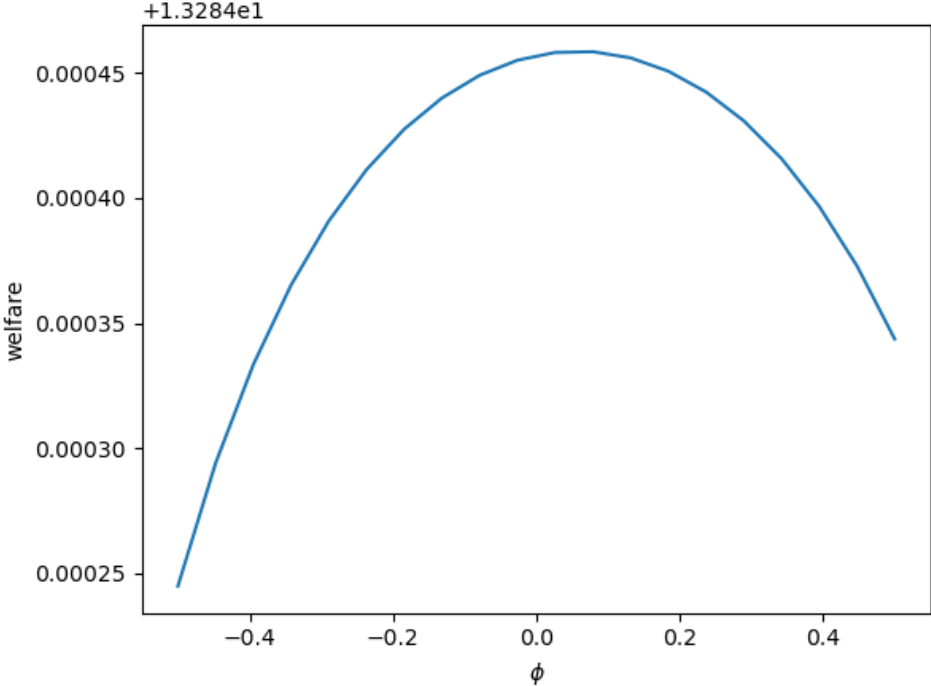
12.5 Wrapping up the consumption-smoothing model

The consumption-smoothing model of Milton Friedman [Friedman, 1956] and Robert Hall [Hall, 1978]) is a cornerstone of modern macro that has important ramifications for the size of the Keynesian "fiscal policy multiplier" described briefly in QuantEcon lecture *geometric series*.

In particular, it **lowers** the government expenditure multiplier relative to one implied by the original Keynesian consumption function presented in *geometric series*.

Friedman's work opened the door to an enlightening literature on the aggregate consumption function and associated government expenditure multipliers that remains active today.

12.6 Appendix: solving difference equations with linear algebra

In the preceding sections we have used linear algebra to solve a consumption-smoothing model.

The same tools from linear algebra –matrix multiplication and matrix inversion –can be used to study many other dynamic models.

We'll conclude this lecture by giving a couple of examples.

We'll describe a useful way of representing and "solving" linear difference equations.

To generate some y vectors, we'll just write down a linear difference equation with appropriate initial conditions and then use linear algebra to solve it.

12.6.1 First-order difference equation

We'll start with a first-order linear difference equation for $\{y_t\}_{t=0}^T$:

$$y_t = \lambda y_{t-1}, \quad t = 1, 2, \ldots, T$$

where y_0 is a given initial condition.

We can cast this set of T equations as a single matrix equation

$$
\begin{bmatrix}
1 & 0 & 0 & \cdots & 0 & 0 \\
-\lambda & 1 & 0 & \cdots & 0 & 0 \\
0 & -\lambda & 1 & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & -\lambda & 1
\end{bmatrix}
\begin{bmatrix}
y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_T
\end{bmatrix}
=
\begin{bmatrix}
\lambda y_0 \\ 0 \\ 0 \\ \vdots \\ 0
\end{bmatrix}
\tag{12.5}
$$

Multiplying both sides of (12.5) by the inverse of the matrix on the left provides the solution

$$
\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_T \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ \lambda & 1 & 0 & \cdots & 0 & 0 \\ \lambda^2 & \lambda & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ \lambda^{T-1} & \lambda^{T-2} & \lambda^{T-3} & \cdots & \lambda & 1 \end{bmatrix} \begin{bmatrix} \lambda y_0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}
\tag{12.6}
$$

Exercise 12.6.1

To get (12.6), we multiplied both sides of (12.5) by the inverse of the matrix A. Please confirm that

$$
\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ \lambda & 1 & 0 & \cdots & 0 & 0 \\ \lambda^2 & \lambda & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ \lambda^{T-1} & \lambda^{T-2} & \lambda^{T-3} & \cdots & \lambda & 1 \end{bmatrix}
$$

is the inverse of A and check that $AA^{-1} = I$

12.6.2 Second-order difference equation

A second-order linear difference equation for $\{y_t\}_{t=0}^T$ is

$$
y_t = \lambda_1 y_{t-1} + \lambda_2 y_{t-2}, \quad t = 1, 2, \ldots, T
$$

where now y_0 and y_{-1} are two given initial equations determined outside the model.

As we did with the first-order difference equation, we can cast this set of T equations as a single matrix equation

$$
\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -\lambda_1 & 1 & 0 & \cdots & 0 & 0 & 0 \\ -\lambda_2 & -\lambda_1 & 1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & -\lambda_2 & -\lambda_1 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_T \end{bmatrix} = \begin{bmatrix} \lambda_1 y_0 + \lambda_2 y_{-1} \\ \lambda_2 y_0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}
$$

Multiplying both sides by inverse of the matrix on the left again provides the solution.

Exercise 12.6.2

As an exercise, we ask you to represent and solve a **third-order linear difference equation**. How many initial conditions must you specify?

EQUALIZING DIFFERENCE MODEL

13.1 Overview

This lecture presents a model of the college-high-school wage gap in which the "time to build" a college graduate plays a key role.

Milton Friedman invented the model to study whether differences in earnings of US dentists and doctors were outcomes of competitive labor markets or whether they reflected entry barriers imposed by governments working in conjunction with doctors' professional organizations.

Chapter 4 of Jennifer Burns [Burns, 2023] describes Milton Friedman's joint work with Simon Kuznets that eventually led to the publication of [Kuznets and Friedman, 1939] and [Friedman and Kuznets, 1945].

To map Friedman's application into our model, think of our high school students as Friedman's dentists and our college graduates as Friedman's doctors.

Our presentation is "incomplete" in the sense that it is based on a single equation that would be part of set equilibrium conditions of a more fully articulated model.

This "equalizing difference" equation determines a college-high-school wage ratio that equalizes present values of a high school educated worker and a college educated worker.

The idea is that lifetime earnings somehow adjust to make a new high school worker indifferent between going to college and not going to college but instead going to work immediately.

(The job of the "other equations" in a more complete model would be to describe what adjusts to bring about this outcome.)

Our model is just one example of an "equalizing difference" theory of relative wage rates, a class of theories dating back at least to Adam Smith's **Wealth of Nations** [Smith, 2010].

For most of this lecture, the only mathematical tools that we'll use are from linear algebra, in particular, matrix multiplication and matrix inversion.

However, near the end of the lecture, we'll use calculus just in case readers want to see how computing partial derivatives could let us present some findings more concisely.

And doing that will let illustrate how good Python is at doing calculus!

But if you don't know calculus, our tools from linear algebra are certainly enough.

As usual, we'll start by importing some Python modules.

```python
import numpy as np
import matplotlib.pyplot as plt
from collections import namedtuple
from sympy import Symbol, Lambda, symbols
```

13.2 The indifference condition

The key idea is that the entry level college wage premium has to adjust to make a representative worker indifferent between going to college and not going to college.

Let

- $R > 1$ be the gross rate of return on a one-period bond

- $t = 0, 1, 2, \dots T$ denote the years that a person either works or attends college

- 0 denote the first period after high school that a person can work if he does not go to college

- T denote the last period that a person works

- w_t^h be the wage at time t of a high school graduate

- w_t^c be the wage at time t of a college graduate

- $\gamma_h > 1$ be the (gross) rate of growth of wages of a high school graduate, so that $w_t^h = w_0^h \gamma_h^t$

- $\gamma_c > 1$ be the (gross) rate of growth of wages of a college graduate, so that $w_t^c = w_0^c \gamma_c^t$

- D be the upfront monetary costs of going to college

We now compute present values that a new high school graduate earns if

- he goes to work immediately and earns wages paid to someone without a college education

- he goes to college for four years and after graduating earns wages paid to a college graduate

13.2.1 Present value of a high school educated worker

If someone goes to work immediately after high school and works for the $T+1$ years $t = 0, 1, 2, \ldots, T$, she earns present value

$$h_0 = \sum_{t=0}^{T} R^{-t} w_t^h = w_0^h \left[\frac{1 - (R^{-1}\gamma_h)^{T+1}}{1 - R^{-1}\gamma_h} \right] \equiv w_0^h A_h$$

where

$$A_h = \left[\frac{1 - (R^{-1}\gamma_h)^{T+1}}{1 - R^{-1}\gamma_h} \right].$$

The present value h_0 is the "human wealth" at the beginning of time 0 of someone who chooses not to attend college but instead to go to work immediately at the wage of a high school graduate.

13.2.2 Present value of a college-bound new high school graduate

If someone goes to college for the four years $t = 0, 1, 2, 3$ during which she earns 0, but then goes to work immediately after college and works for the $T-3$ years $t = 4, 5, \ldots, T$, she earns present value

$$c_0 = \sum_{t=4}^{T} R^{-t} w_t^c = w_0^c (R^{-1}\gamma_c)^4 \left[\frac{1 - (R^{-1}\gamma_c)^{T-3}}{1 - R^{-1}\gamma_c} \right] \equiv w_0^c A_c$$

where

$$A_c = (R^{-1}\gamma_c)^4 \left[\frac{1 - (R^{-1}\gamma_c)^{T-3}}{1 - R^{-1}\gamma_c} \right].$$

The present value c_0 is the "human wealth" at the beginning of time 0 of someone who chooses to attend college for four years and then start to work at time $t = 4$ at the wage of a college graduate.

Assume that college tuition plus four years of room and board amount to D and must be paid at time 0.

So net of monetary cost of college, the present value of attending college as of the first period after high school is

$$c_0 - D$$

We now formulate a pure **equalizing difference** model of the initial college-high school wage gap ϕ that verifies

$$w_0^c = \phi w_0^h$$

We suppose that R, γ_h, γ_c, T and also w_0^h are fixed parameters.

We start by noting that the pure equalizing difference model asserts that the college-high-school wage gap ϕ solves an "equalizing" equation that sets the present value not going to college equal to the present value of going to college:

$$h_0 = c_0 - D$$

or

$$w_0^h A_h = \phi w_0^h A_c - D. \tag{13.1}$$

This "indifference condition" is the heart of the model.

Solving equation (13.1) for the college wage premium ϕ we obtain

$$\phi = \frac{A_h}{A_c} + \frac{D}{w_0^h A_c}. \tag{13.2}$$

In a **free college** special case $D = 0$.

Here the only cost of going to college is the forgone earnings from being a high school educated worker.

In that case,

$$\phi = \frac{A_h}{A_c}.$$

In the next section we'll write Python code to compute ϕ and plot it as a function of its determinants.

13.3 Computations

We can have some fun with examples that tweak various parameters, prominently including γ_h, γ_c, R.

Now let's write some Python code to compute ϕ and plot it as a function of some of its determinants.

```python
# Define the namedtuple for the equalizing difference model
EqDiffModel = namedtuple('EqDiffModel', 'R T γ_h γ_c w_h0 D')

def create_edm(R=1.05,    # gross rate of return
               T=40,      # time horizon
               γ_h=1.01,  # high-school wage growth
               γ_c=1.01,  # college wage growth
               w_h0=1,    # initial wage (high school)
               D=10,      # cost for college
              ):

    return EqDiffModel(R, T, γ_h, γ_c, w_h0, D)

def compute_gap(model):
    R, T, γ_h, γ_c, w_h0, D = model

    A_h = (1 - (γ_h/R)**(T+1)) / (1 - γ_h/R)
    A_c = (1 - (γ_c/R)**(T-3)) / (1 - γ_c/R)  * (γ_c/R)**4
    φ = A_h / A_c + D / (w_h0 * A_c)

    return φ
```

Using vectorization instead of loops, we build some functions to help do comparative statics .

For a given instance of the class, we want to recompute ϕ when one parameter changes and others remain fixed.

Let's do an example.

```
ex1 = create_edm()
gap1 = compute_gap(ex1)


gap1
```

```
1.8041412724969135
```

Let's not charge for college and recompute ϕ.

The initial college wage premium should go down.

```
# free college
ex2 = create_edm(D=0)
gap2 = compute_gap(ex2)
gap2
```

```
1.2204649517903732
```

Let us construct some graphs that show us how the initial college-high-school wage ratio ϕ would change if one of its determinants were to change.

Let's start with the gross interest rate R.

```
R_arr = np.linspace(1, 1.2, 50)
models = [create_edm(R=r) for r in R_arr]
gaps = [compute_gap(model) for model in models]

plt.plot(R_arr, gaps)
plt.xlabel(r'$R$')
plt.ylabel(r'wage gap')
plt.show()
```
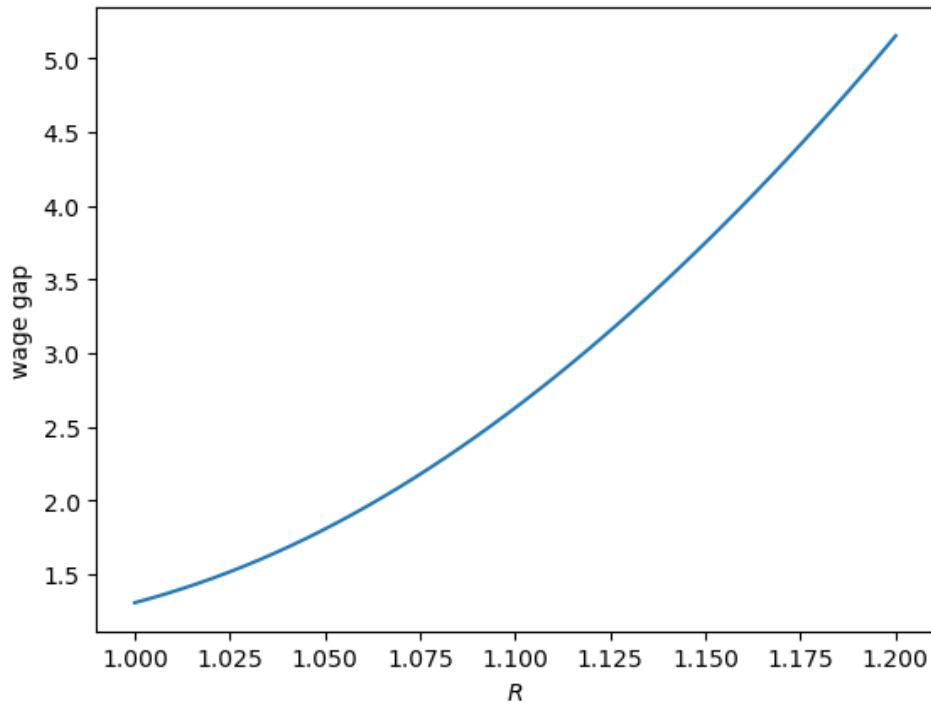
Evidently, the initial wage ratio ϕ must rise to compensate a prospective high school student for **waiting** to start receiving income –remember that while she is earning nothing in years $t = 0, 1, 2, 3$, the high school worker is earning a salary.

Not let's study what happens to the initial wage ratio ϕ if the rate of growth of college wages rises, holding constant other determinants of ϕ.

```python
yc_arr = np.linspace(1, 1.2, 50)
models = [create_edm(y_c=y_c) for y_c in yc_arr]
gaps = [compute_gap(model) for model in models]


plt.plot(yc_arr, gaps)
plt.xlabel(r'$\gamma_c$')
plt.ylabel(r'wage gap')
plt.show()
```
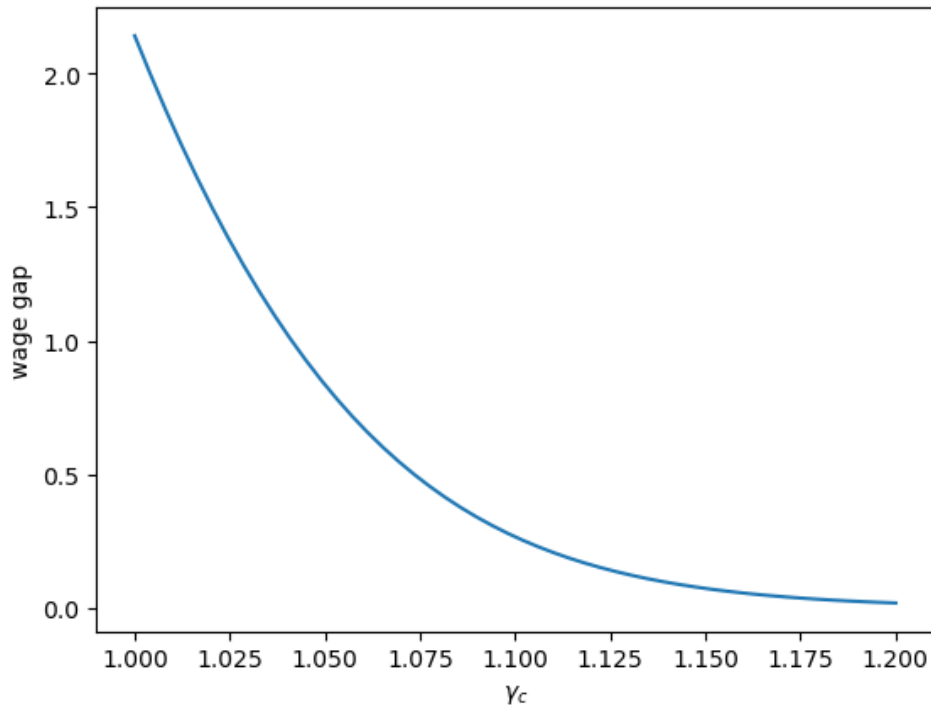
Notice how the initial wage gap falls when the rate of growth γ_c of college wages rises.

The wage gap falls to "equalize" the present values of the two types of career, one as a high school worker, the other as a college worker.

Can you guess what happens to the initial wage ratio ϕ when next we vary the rate of growth of high school wages, holding all other determinants of ϕ constant?

The following graph shows what happens.

```python
γh_arr = np.linspace(1, 1.1, 50)
models = [create_edm(γ_h=γ_h) for γ_h in γh_arr]
gaps = [compute_gap(model) for model in models]

plt.plot(γh_arr, gaps)
plt.xlabel(r'$\gamma_h$')
plt.ylabel(r'wage gap')
plt.show()
```
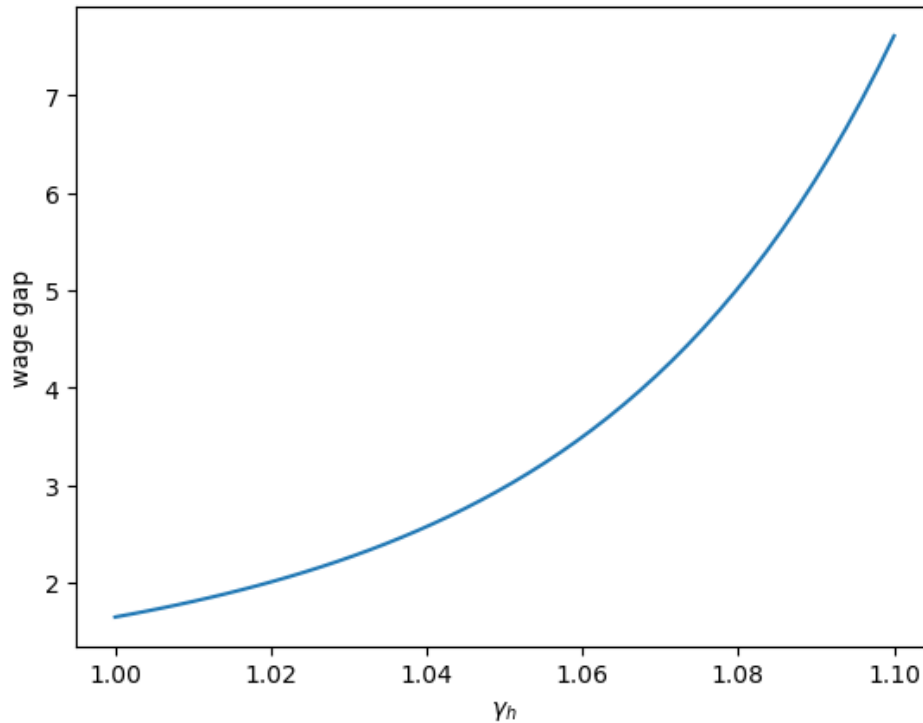
13.4 Entrepreneur-worker interpretation

We can add a parameter and reinterpret variables to get a model of entrepreneurs versus workers.

We now let h be the present value of a "worker".

We define the present value of an entrepreneur to be

$$c_0 = \pi \sum_{t=4}^{T} R^{-t} w_t^c$$

where $\pi \in (0, 1)$ is the probability that an entrepreneur's "project" succeeds.

For our model of workers and firms, we'll interpret D as the cost of becoming an entrepreneur.

This cost might include costs of hiring workers, office space, and lawyers.

What we used to call the college, high school wage gap ϕ now becomes the ratio of a successful entrepreneur's earnings to a worker's earnings.

We'll find that as π decreases, ϕ increases, indicating that the riskier it is to be an entrepreneur, the higher must be the reward for a successful project.

Now let's adopt the entrepreneur-worker interpretation of our model

```python
# Define a model of entrepreneur-worker interpretation
EqDiffModel = namedtuple('EqDiffModel', 'R T γ_h γ_c w_h0 D π')


def create_edm_π(R=1.05,    # gross rate of return
                 T=40,      # time horizon
                 γ_h=1.01,  # high-school wage growth
                 γ_c=1.01,  # college wage growth
                 w_h0=1,    # initial wage (high school)
                 D=10,      # cost for college
                 π=0        # chance of business success
               ):

    return EqDiffModel(R, T, γ_h, γ_c, w_h0, D, π)



def compute_gap(model):
    R, T, γ_h, γ_c, w_h0, D, π = model

    A_h = (1 - (γ_h/R)**(T+1)) / (1 - γ_h/R)
    A_c = (1 - (γ_c/R)**(T-3)) / (1 - γ_c/R) * (γ_c/R)**4

    # Incorprate chance of success
    A_c = π * A_c

    φ = A_h / A_c + D / (w_h0 * A_c)
    return φ
```

If the probability that a new business succeeds is 0.2, let's compute the initial wage premium for successful entrepreneurs.

```python
ex3 = create_edm_π(π=0.2)
gap3 = compute_gap(ex3)


gap3
```

```
9.020706362484567
```

Now let's study how the initial wage premium for successful entrepreneurs depend on the success probability.

```python
π_arr = np.linspace(0.2, 1, 50)
models = [create_edm_π(π=π) for π in π_arr]
gaps = [compute_gap(model) for model in models]


plt.plot(π_arr, gaps)
```

```
plt.ylabel(r'wage gap')
plt.xlabel(r'$\pi$')
plt.show()
```
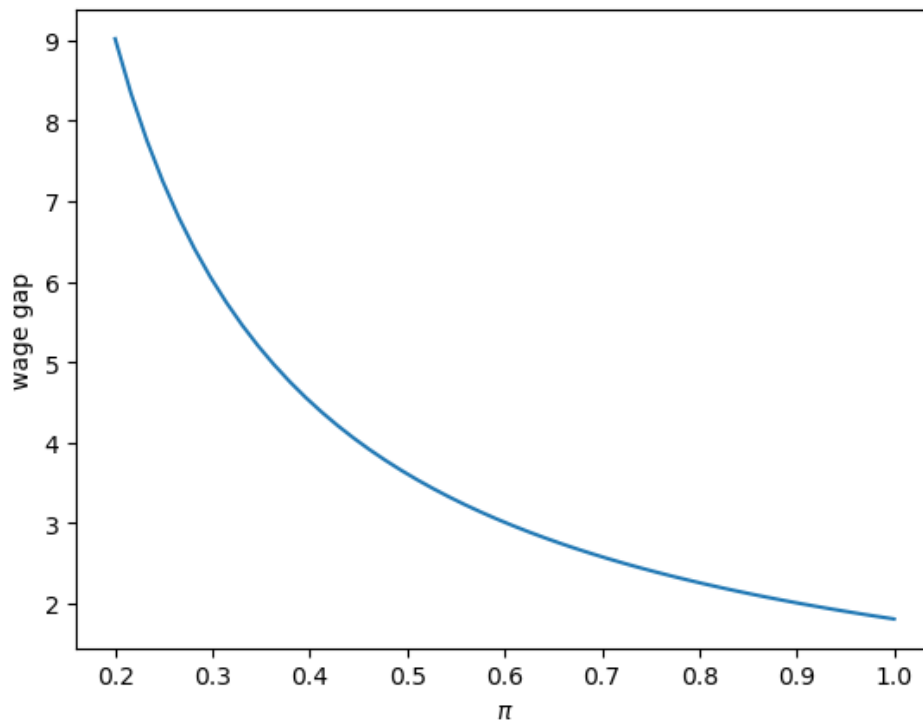

Does the graph make sense to you?

13.5 An application of calculus

So far, we have used only linear algebra and it has been a good enough tool for us to figure out how our model works.

However, someone who knows calculus might want us just to take partial derivatives.

We'll do that now.

A reader who doesn't know calculus could read no further and feel confident that applying linear algebra has taught us the main properties of the model.

But for a reader interested in how we can get Python to do all the hard work involved in computing partial derivatives, we'll say a few things about that now.

We'll use the Python module 'sympy' to compute partial derivatives of ϕ with respect to the parameters that determine it.

Define symbols

```
γ_h, γ_c, w_h0, D = symbols('\gamma_h, \gamma_c, w_0^h, D', real=True)
R, T = Symbol('R', real=True), Symbol('T', integer=True)
```

Define function A_h

```
A_h = Lambda((γ_h, R, T), (1 - (γ_h/R)**(T+1)) / (1 - γ_h/R))
A_h
```

$$\left((\gamma_h,\ R,\ T) \mapsto \frac{1 - \left(\frac{\gamma_h}{R} \right)^{T+1}}{1 - \frac{\gamma_h}{R}} \right)$$

Define function A_c

```
A_c = Lambda((γ_c, R, T), (1 - (γ_c/R)**(T-3)) / (1 - γ_c/R) * (γ_c/R)**4)
A_c
```

$$\left((\gamma_c,\ R,\ T) \mapsto \frac{\gamma_c^4 \cdot \left(1 - \left(\frac{\gamma_c}{R} \right)^{T-3} \right)}{R^4 \cdot \left(1 - \frac{\gamma_c}{R} \right)} \right)$$

Now, define ϕ

```
φ = Lambda((D, γ_h, γ_c, R, T, w_h0), A_h(γ_h, R, T)/A_c(γ_c, R, T) + D/(w_h0*A_c(γ_c,
 ↪ R, T)))
```

```
φ
```

$$\left((D,\ \gamma_h,\ \gamma_c,\ R,\ T,\ w_0^h) \mapsto \frac{DR^4 \cdot \left(1 - \frac{\gamma_c}{R} \right)}{\gamma_c^4 w_0^h \cdot \left(1 - \left(\frac{\gamma_c}{R} \right)^{T-3} \right)} + \frac{R^4 \cdot \left(1 - \frac{\gamma_c}{R} \right) \left(1 - \left(\frac{\gamma_h}{R} \right)^{T+1} \right)}{\gamma_c^4 \cdot \left(1 - \left(\frac{\gamma_c}{R} \right)^{T-3} \right) \left(1 - \frac{\gamma_h}{R} \right)} \right)$$

We begin by setting default parameter values.

```
R_value = 1.05
T_value = 40
γ_h_value, γ_c_value = 1.01, 1.01
w_h0_value = 1
D_value = 10
```

13.5. An application of calculus 261

Now let's compute $\frac{\partial \phi}{\partial D}$ and then evaluate it at the default values

```
φ_D = φ(D, γ_h, γ_c, R, T, w_h0).diff(D)
φ_D
```

$$\frac{R^4 \cdot \left(1 - \frac{\gamma_c}{R}\right)}{\gamma_c^4 w_0^h \cdot \left(1 - \left(\frac{\gamma_c}{R}\right)^{T-3}\right)}$$

```
# Numerical value at default parameters
φ_D_func = Lambda((D, γ_h, γ_c, R, T, w_h0), φ_D)
φ_D_func(D_value, γ_h_value, γ_c_value, R_value, T_value, w_h0_value)
```

$$0.058367632070654$$

Thus, as with our earlier graph, we find that raising R increases the initial college wage premium ϕ.

Compute $\frac{\partial \phi}{\partial T}$ and evaluate it at default parameters

```
φ_T = φ(D, γ_h, γ_c, R, T, w_h0).diff(T)
φ_T
```

$$\frac{DR^4 \left(\frac{\gamma_c}{R}\right)^{T-3} \cdot \left(1 - \frac{\gamma_c}{R}\right) \log\left(\frac{\gamma_c}{R}\right)}{\gamma_c^4 w_0^h \left(1 - \left(\frac{\gamma_c}{R}\right)^{T-3}\right)^2} + \frac{R^4 \left(\frac{\gamma_c}{R}\right)^{T-3} \cdot \left(1 - \frac{\gamma_c}{R}\right) \left(1 - \left(\frac{\gamma_h}{R}\right)^{T+1}\right) \log\left(\frac{\gamma_c}{R}\right)}{\gamma_c^4 \left(1 - \left(\frac{\gamma_c}{R}\right)^{T-3}\right)^2 \cdot \left(1 - \frac{\gamma_h}{R}\right)} - \frac{R^4 \left(\frac{\gamma_h}{R}\right)^{T+1} \cdot \left(1 - \frac{\gamma_c}{R}\right) \log\left(\frac{\gamma_h}{R}\right)}{\gamma_c^4 \cdot \left(1 - \left(\frac{\gamma_c}{R}\right)^{T-3}\right) \left(1 - \frac{\gamma_h}{R}\right)}$$

```
# Numerical value at default parameters
φ_T_func = Lambda((D, γ_h, γ_c, R, T, w_h0), φ_T)
φ_T_func(D_value, γ_h_value, γ_c_value, R_value, T_value, w_h0_value)
```

$$-0.00973478032996598$$

We find that raising T decreases the initial college wage premium ϕ.

This is because college graduates now have longer career lengths to "pay off" the time and other costs they paid to go to college

Let's compute $\frac{\partial \phi}{\partial \gamma_h}$ and evaluate it at default parameters.

```
φ_y_h = φ(D, y_h, y_c, R, T, w_h0).diff(y_h)
φ_y_h
```

$$-\frac{R^4 \left(\frac{\gamma_h}{R}\right)^{T+1} \cdot \left(1 - \frac{\gamma_c}{R}\right)(T+1)}{\gamma_c^4 \gamma_h \left(1 - \left(\frac{\gamma_c}{R}\right)^{T-3}\right)\left(1 - \frac{\gamma_h}{R}\right)} + \frac{R^3 \cdot \left(1 - \frac{\gamma_c}{R}\right)\left(1 - \left(\frac{\gamma_h}{R}\right)^{T+1}\right)}{\gamma_c^4 \cdot \left(1 - \left(\frac{\gamma_c}{R}\right)^{T-3}\right)\left(1 - \frac{\gamma_h}{R}\right)^2}$$

```
# Numerical value at default parameters
φ_y_h_func = Lambda((D, y_h, y_c, R, T, w_h0), φ_y_h)
φ_y_h_func(D_value, y_h_value, y_c_value, R_value, T_value, w_h0_value)
```

$$17.8590485545256$$

We find that raising γ_h increases the initial college wage premium ϕ, in line with our earlier graphical analysis.

Compute $\frac{\partial \phi}{\partial \gamma_c}$ and evaluate it numerically at default parameter values

```
φ_y_c = φ(D, y_h, y_c, R, T, w_h0).diff(y_c)
φ_y_c
```

$$\frac{DR^4 \left(\frac{\gamma_c}{R}\right)^{T-3} \cdot \left(1 - \frac{\gamma_c}{R}\right)(T-3)}{\gamma_c^5 w_0^h \left(1 - \left(\frac{\gamma_c}{R}\right)^{T-3}\right)^2} - \frac{4DR^4 \cdot \left(1 - \frac{\gamma_c}{R}\right)}{\gamma_c^5 w_0^h \cdot \left(1 - \left(\frac{\gamma_c}{R}\right)^{T-3}\right)} - \frac{DR^3}{\gamma_c^4 w_0^h \cdot \left(1 - \left(\frac{\gamma_c}{R}\right)^{T-3}\right)} + \frac{R^4 \left(\frac{\gamma_c}{R}\right)^{T-3} \cdot \left(1 - \frac{\gamma_c}{R}\right)\left(1 - \left(\frac{\gamma_h}{R}\right)^{T+1}\right)}{\gamma_c^5 \left(1 - \left(\frac{\gamma_c}{R}\right)^{T-3}\right)^2 \cdot \left(1 - \frac{\gamma_h}{R}\right)}$$

```
# Numerical value at default parameters
φ_y_c_func = Lambda((D, y_h, y_c, R, T, w_h0), φ_y_c)
φ_y_c_func(D_value, y_h_value, y_c_value, R_value, T_value, w_h0_value)
```

$$-31.6486401973376$$

We find that raising γ_c decreases the initial college wage premium ϕ, in line with our earlier graphical analysis.

Let's compute $\frac{\partial \phi}{\partial R}$ and evaluate it numerically at default parameter values

```
φ_R = φ(D, y_h, y_c, R, T, w_h0).diff(R)
φ_R
```

$$-\frac{DR^3\left(\frac{\gamma_c}{R}\right)^{T-3}\cdot\left(1-\frac{\gamma_c}{R}\right)(T-3)}{\gamma_c^4 w_0^h\left(1-\left(\frac{\gamma_c}{R}\right)^{T-3}\right)^2} + \frac{4DR^3\cdot\left(1-\frac{\gamma_c}{R}\right)}{\gamma_c^4 w_0^h\cdot\left(1-\left(\frac{\gamma_c}{R}\right)^{T-3}\right)} + \frac{DR^2}{\gamma_c^3 w_0^h\cdot\left(1-\left(\frac{\gamma_c}{R}\right)^{T-3}\right)} - \frac{R^3\left(\frac{\gamma_c}{R}\right)^{T-3}\cdot\left(1-\frac{\gamma_c}{R}\right)\left(1-\left(\frac{\gamma_h}{R}\right)^{T-1}\right)}{\gamma_c^4\left(1-\left(\frac{\gamma_c}{R}\right)^{T-3}\right)^2\cdot\left(1-\right.}$$

```
# Numerical value at default parameters
ϕ_R_func = Lambda((D, γ_h, γ_c, R, T, w_h0), ϕ_R)
ϕ_R_func(D_value, γ_h_value, γ_c_value, R_value, T_value, w_h0_value)
```

$$13.2642738659429$$

We find that raising the gross interest rate R increases the initial college wage premium ϕ, in line with our earlier graphical analysis.

FOURTEEN

A MONETARIST THEORY OF PRICE LEVELS

14.1 Overview

We'll use linear algebra first to explain and then do some experiments with a "monetarist theory of price levels".

Economists call it a "monetary" or "monetarist" theory of price levels because effects on price levels occur via a central bank's decisions to print money supply.

- a goverment's fiscal policies determine whether its *expenditures* exceed its *tax collections*

- if its expenditures exceed its tax collections, the government can instruct the central bank to cover the difference by *printing money*

- that leads to effects on the price level as price level path adjusts to equate the supply of money to the demand for money

Such a theory of price levels was described by Thomas Sargent and Neil Wallace in chapter 5 of [Sargent, 2013], which reprints a 1981 Federal Reserve Bank of Minneapolis article entitled "Unpleasant Monetarist Arithmetic".

Sometimes this theory is also called a "fiscal theory of price levels" to emphasize the importance of fiscal deficits in shaping changes in the money supply.

The theory has been extended, criticized, and applied by John Cochrane [Cochrane, 2023].

In another lecture *price level histories*, we described some European hyperinflations that occurred in the wake of World War I.

Elemental forces at work in the fiscal theory of the price level help to understand those episodes.

According to this theory, when the government persistently spends more than it collects in taxes and prints money to finance the shortfall (the "shortfall" is called the "government deficit"), it puts upward pressure on the price level and generates persistent inflation.

The "monetarist" or "fiscal theory of price levels" asserts that

- to *start* a persistent inflation the government begins persistently to run a money-financed government deficit

- to *stop* a persistent inflation the government stops persistently running a money-financed government deficit

The model in this lecture is a "rational expectations"(or "perfect foresight") version of a model that Philip Cagan [Cagan, 1956] used to study the monetary dynamics of hyperinflations.

While Cagan didn't use that "rational expectations"version of the model, Thomas Sargent [Sargent, 1982] did when he studied the Ends of Four Big Inflations in Europe after World War I.

- this lecture *fiscal theory of the price level with adaptive expectations* describes a version of the model that does not impose "rational expectations"but instead uses what Cagan and his teacher Milton Friedman called "adaptive expectations"

    - a reader of both lectures will notice that the algebra is less complicated in the present rational expectations version of the model

    - the difference in algebra complications can be traced to the following source: the adaptive expectations version of the model has more endogenous variables and more free parameters

Some of our quantitative experiments with the rational expectations version of the model are designed to illustrate how the fiscal theory explains the abrupt end of those big inflations.

In those experiments, we'll encounter an instance of a "velocity dividend"that has sometimes accompanied successful inflation stabilization programs.

To facilitate using linear matrix algebra as our main mathematical tool, we'll use a finite horizon version of the model.

As in the *present values* and *consumption smoothing* lectures, our mathematical tools are matrix multiplication and matrix inversion.

14.2 Structure of the model

The model consists of

- a function that expresses the demand for real balances of government printed money as an inverse function of the public's expected rate of inflation

- an exogenous sequence of rates of growth of the money supply. The money supply grows because the government prints it to pay for goods and services

- an equilibrium condition that equates the demand for money to the supply

- a "perfect foresight"assumption that the public's expected rate of inflation equals the actual rate of inflation.

To represent the model formally, let

- m_t be the log of the supply of nominal money balances;

- $\mu_t = m_{t+1} - m_t$ be the net rate of growth of nominal balances;

- p_t be the log of the price level;

- $\pi_t = p_{t+1} - p_t$ be the net rate of inflation between t and $t+1$;

- π_t^* be the public's expected rate of inflation between t and $t+1$;

- T the horizon –i.e., the last period for which the model will determine p_t

- π_{T+1}^* the terminal rate of inflation between times T and $T+1$.

The demand for real balances $\exp\left(m_t^d - p_t\right)$ is governed by the following version of the Cagan demand function

$$m_t^d - p_t = -\alpha\pi_t^*, \ \alpha > 0; \quad t = 0, 1, \ldots, T. \tag{14.1}$$

This equation asserts that the demand for real balances is inversely related to the public's expected rate of inflation with sensitivity α.

People somehow acquire **perfect foresight** by their having solved a forecasting problem.

This lets us set

$$\pi_t^* = \pi_t, \tag{14.2}$$

while equating demand for money to supply lets us set $m_t^d = m_t$ for all $t \geq 0$.

The preceding equations then imply

$$m_t - p_t = -\alpha(p_{t+1} - p_t) \tag{14.3}$$

To fill in details about what it means for private agents to have perfect foresight, we subtract equation (14.3) at time t from the same equation at $t+1$ to get

$$\mu_t - \pi_t = -\alpha\pi_{t+1} + \alpha\pi_t,$$

which we rewrite as a forward-looking first-order linear difference equation in π_s with μ_s as a "forcing variable":

$$\pi_t = \frac{\alpha}{1+\alpha}\pi_{t+1} + \frac{1}{1+\alpha}\mu_t, \quad t = 0, 1, \ldots, T$$

where $0 < \frac{\alpha}{1+\alpha} < 1$.

Setting $\delta = \frac{\alpha}{1+\alpha}$, let's us represent the preceding equation as

$$\pi_t = \delta\pi_{t+1} + (1-\delta)\mu_t, \quad t = 0, 1, \ldots, T$$

Write this system of $T+1$ equations as the single matrix equation

$$
\begin{bmatrix}
1 & -\delta & 0 & 0 & \cdots & 0 & 0 \\
0 & 1 & -\delta & 0 & \cdots & 0 & 0 \\
0 & 0 & 1 & -\delta & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & -\delta & 0 \\
0 & 0 & 0 & 0 & \cdots & 1 & -\delta \\
0 & 0 & 0 & 0 & \cdots & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\pi_0 \\
\pi_1 \\
\pi_2 \\
\vdots \\
\pi_{T-1} \\
\pi_T
\end{bmatrix}
= (1-\delta)
\begin{bmatrix}
\mu_0 \\
\mu_1 \\
\mu_2 \\
\vdots \\
\mu_{T-1} \\
\mu_T
\end{bmatrix}
+
\begin{bmatrix}
0 \\
0 \\
0 \\
\vdots \\
0 \\
\delta\pi_{T+1}^*
\end{bmatrix}
\tag{14.4}
$$

By multiplying both sides of equation (14.4) by the inverse of the matrix on the left side, we can calculate

$$\pi \equiv \begin{bmatrix} \pi_0 \\ \pi_1 \\ \pi_2 \\ \vdots \\ \pi_{T-1} \\ \pi_T \end{bmatrix}$$

It turns out that

$$\pi_t = (1 - \delta) \sum_{s=t}^{T} \delta^{s-t} \mu_s + \delta^{T+1-t} \pi_{T+1}^* \tag{14.5}$$

We can represent the equations

$$m_{t+1} = m_t + \mu_t, \quad t = 0, 1, \dots, T$$

as the matrix equation

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ -1 & 1 & 0 & \cdots & 0 & 0 \\ 0 & -1 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & 0 & 0 \\ 0 & 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & 0 & \cdots & -1 & 1 \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ \vdots \\ m_T \\ m_{T+1} \end{bmatrix} = \begin{bmatrix} \mu_0 \\ \mu_1 \\ \mu_2 \\ \vdots \\ \mu_{T-1} \\ \mu_T \end{bmatrix} + \begin{bmatrix} m_0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \tag{14.6}$$

Multiplying both sides of equation (14.6) with the inverse of the matrix on the left will give

$$m_t = m_0 + \sum_{s=0}^{t-1} \mu_s, \quad t = 1, \dots, T+1 \tag{14.7}$$

Equation (14.7) shows that the log of the money supply at t equals the log of the initial money supply m_0 plus accumulation of rates of money growth between times 0 and T.

14.3 Continuation values

To determine the continuation inflation rate π_{T+1}^* we shall proceed by applying the following infinite-horizon version of equation (14.5) at time $t = T + 1$:

$$\pi_t = (1 - \delta) \sum_{s=t}^{\infty} \delta^{s-t} \mu_s, \tag{14.8}$$

and by also assuming the following continuation path for μ_t beyond T:

$$\mu_{t+1} = \gamma^* \mu_t, \quad t \geq T.$$

Plugging the preceding equation into equation (14.8) at $t = T + 1$ and rearranging we can deduce that

$$\pi_{T+1}^* = \frac{1 - \delta}{1 - \delta\gamma^*} \gamma^* \mu_T \tag{14.9}$$

where we require that $|\gamma^*\delta| < 1$.

Let's implement and solve this model.

As usual, we'll start by importing some Python modules.

```python
import numpy as np
from collections import namedtuple
import matplotlib.pyplot as plt
```

First, we store parameters in a `namedtuple`:

```python
# Create the rational expectation version of Cagan model in finite time
CaganREE = namedtuple("CaganREE",
                      ["m0",      # initial money supply
                       "μ_seq",   # sequence of rate of growth
                       "α",       # sensitivity parameter
                       "δ",       # α/(1 + α)
                       "π_end"    # terminal expected inflation
                      ])


def create_cagan_model(m0=1, α=5, μ_seq=None):
    δ = α/(1 + α)
    π_end = μ_seq[-1]      # compute terminal expected inflation
    return CaganREE(m0, μ_seq, α, δ, π_end)
```

Now we can solve the model to compute π_t, m_t and p_t for $t = 1, \ldots, T+1$ using the matrix equation above

```python
def solve(model, T):
    m0, π_end, μ_seq, α, δ = (model.m0, model.π_end,
                             model.μ_seq, model.α, model.δ)

    # Create matrix representation above
    A1 = np.eye(T+1, T+1) - δ * np.eye(T+1, T+1, k=1)
    A2 = np.eye(T+1, T+1) - np.eye(T+1, T+1, k=-1)

    b1 = (1-δ) * μ_seq + np.concatenate([np.zeros(T), [δ * π_end]])
    b2 = μ_seq + np.concatenate([[m0], np.zeros(T)])

    π_seq = np.linalg.solve(A1, b1)
    m_seq = np.linalg.solve(A2, b2)

    π_seq = np.append(π_seq, π_end)
    m_seq = np.append(m0, m_seq)
```

```
    p_seq = m_seq + α * π_seq


    return π_seq, m_seq, p_seq
```

14.3.1 Some quantitative experiments

In the experiments below, we'll use formula (14.9) as our terminal condition for expected inflation.

In devising these experiments, we'll make assumptions about $\{\mu_t\}$ that are consistent with formula (14.9).

We describe several such experiments.

In all of them,

$$\mu_t = \mu^*, \quad t \geq T_1$$

so that, in terms of our notation and formula for π^*_{T+1} above, $\gamma^* = 1$.

Experiment 1: Foreseen sudden stabilization

In this experiment, we'll study how, when $\alpha > 0$, a foreseen inflation stabilization has effects on inflation that proceed it.

We'll study a situation in which the rate of growth of the money supply is μ_0 from $t = 0$ to $t = T_1$ and then permanently falls to μ^* at $t = T_1$.

Thus, let $T_1 \in (0, T)$.

So where $\mu_0 > \mu^*$, we assume that

$$\mu_{t+1} = \begin{cases} \mu_0, & t = 0, \dots, T_1 - 1 \\ \mu^*, & t \geq T_1 \end{cases}$$

We'll start by executing a version of our "experiment 1" in which the government implements a *foreseen* sudden permanent reduction in the rate of money creation at time T_1.

Let's experiment with the following parameters

```
T1 = 60
μ0 = 0.5
μ_star = 0
T = 80


μ_seq_1 = np.append(μ0*np.ones(T1+1), μ_star*np.ones(T-T1))
```

```
cm = create_cagan_model(μ_seq=μ_seq_1)

# solve the model
π_seq_1, m_seq_1, p_seq_1 = solve(cm, T)
```

Now we use the following function to plot the result

```python
def plot_sequences(sequences, labels):
    fig, axs = plt.subplots(len(sequences), 1, figsize=(5, 12), dpi=200)
    for ax, seq, label in zip(axs, sequences, labels):
        ax.plot(range(len(seq)), seq, label=label)
        ax.set_ylabel(label)
        ax.set_xlabel('$t$')
        ax.legend()
    plt.tight_layout()
    plt.show()

sequences = (μ_seq_1, π_seq_1, m_seq_1 - p_seq_1, m_seq_1, p_seq_1)
plot_sequences(sequences, (r'$\mu$', r'$\pi$', r'$m - p$', r'$m$', r'$p$'))
```
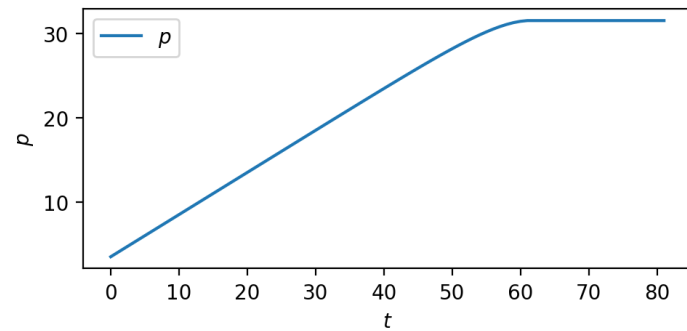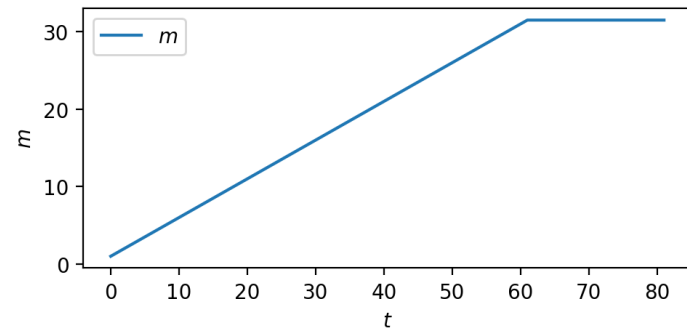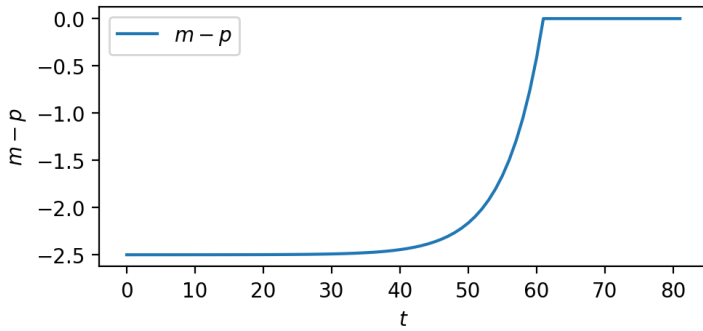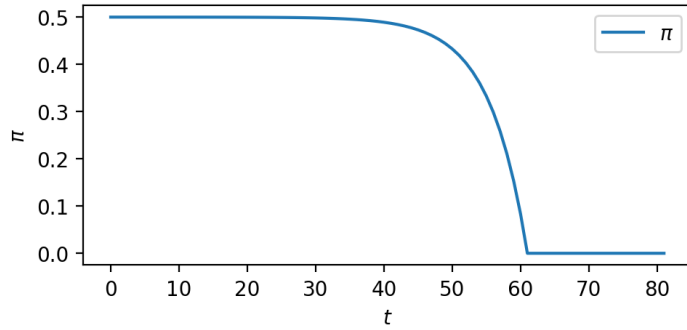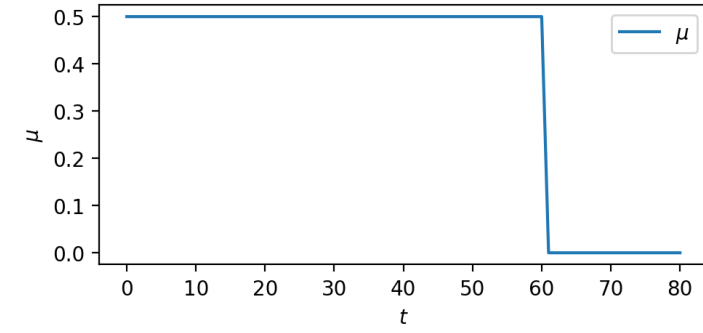
The plot of the money growth rate μ_t in the top level panel portrays a sudden reduction from .5 to 0 at time $T_1 = 60$.

This brings about a gradual reduction of the inflation rate π_t that precedes the money supply growth rate reduction at time T_1.

Notice how the inflation rate declines smoothly (i.e., continuously) to 0 at T_1 –unlike the money growth rate, it does not suddenly "jump" downward at T_1.

This is because the reduction in μ at T_1 has been foreseen from the start.

While the log money supply portrayed in the bottom panel has a kink at T_1, the log price level does not –it is "smooth" – once again a consequence of the fact that the reduction in μ has been foreseen.

To set the stage for our next experiment, we want to study the determinants of the price level a little more.

14.3.2 The log price level

We can use equations (14.1) and (14.2) to discover that the log of the price level satisfies

$$p_t = m_t + \alpha \pi_t \tag{14.10}$$

or, by using equation (14.5),

$$p_t = m_t + \alpha \left[(1 - \delta) \sum_{s=t}^{T} \delta^{s-t} \mu_s + \delta^{T+1-t} \pi_{T+1}^* \right] \tag{14.11}$$

In our next experiment, we'll study a "surprise" permanent change in the money growth that beforehand was completely unanticipated.

At time T_1 when the "surprise" money growth rate change occurs, to satisfy equation (14.10), the log of real balances jumps *upward* as π_t jumps *downward*.

But in order for $m_t - p_t$ to jump, which variable jumps, m_{T_1} or p_{T_1}?

We'll study that interesting question next.

14.3.3 What jumps?

What jumps at T_1?

Is it p_{T_1} or m_{T_1}?

If we insist that the money supply m_{T_1} is locked at its value $m_{T_1}^1$ inherited from the past, then formula (14.10) implies that the price level jumps downward at time T_1, to coincide with the downward jump in π_{T_1}

An alternative assumption about the money supply level is that as part of the "inflation stabilization", the government resets m_{T_1} according to

$$m_{T_1}^2 - m_{T_1}^1 = \alpha(\pi_{T_1}^1 - \pi_{T_1}^2), \tag{14.12}$$

which describes how the government could reset the money supply at T_1 in response to the jump in expected inflation associated with monetary stabilization.

Doing this would let the price level be continuous at T_1.

By letting money jump according to equation (14.12) the monetary authority prevents the price level from *falling* at the moment that the unanticipated stabilization arrives.

In various research papers about stabilizations of high inflations, the jump in the money supply described by equation (14.12) has been called "the velocity dividend" that a government reaps from implementing a regime change that sustains a permanently lower inflation rate.

Technical details about whether p or m jumps at T_1

We have noted that with a constant expected forward sequence $\mu_s = \bar{\mu}$ for $s \geq t$, $\pi_t = \bar{\mu}$.

A consequence is that at T_1, either m or p must "jump" at T_1.

We'll study both cases.

m_{T_1} does not jump.

$$m_{T_1} = m_{T_1 - 1} + \mu_0$$
$$\pi_{T_1} = \mu^*$$
$$p_{T_1} = m_{T_1} + \alpha \pi_{T_1}$$

Simply glue the sequences $t \leq T_1$ and $t > T_1$.

m_{T_1} jumps.

We reset m_{T_1} so that $p_{T_1} = \left(m_{T_1 - 1} + \mu_0\right) + \alpha \mu_0$, with $\pi_{T_1} = \mu^*$.

Then,

$$m_{T_1} = p_{T_1} - \alpha \pi_{T_1} = \left(m_{T_1 - 1} + \mu_0\right) + \alpha \left(\mu_0 - \mu^*\right)$$

We then compute for the remaining $T - T_1$ periods with $\mu_s = \mu^*$, $\forall s \geq T_1$ and the initial condition m_{T_1} from above.

We are now technically equipped to discuss our next experiment.

Experiment 2: an unforeseen sudden stabilization

This experiment deviates a little bit from a pure version of our "perfect foresight" assumption by assuming that a sudden permanent reduction in μ_t like that analyzed in experiment 1 is completely unanticipated.

Such a completely unanticipated shock is popularly known as an "MIT shock".

The mental experiment involves switching at time T_1 from an initial "continuation path" for $\{\mu_t, \pi_t\}$ to another path that involves a permanently lower inflation rate.

Initial Path: $\mu_t = \mu_0$ for all $t \geq 0$. So this path is for $\{\mu_t\}_{t=0}^{\infty}$; the associated path for π_t has $\pi_t = \mu_0$.

Revised Continuation Path Where $\mu_0 > \mu^*$, we construct a continuation path $\{\mu_s\}_{s=T_1}^{\infty}$ by setting $\mu_s = \mu^*$ for all $s \geq T_1$. The perfect foresight continuation path for π is $\pi_s = \mu^*$

To capture a "completely unanticipated permanent shock to the $\{\mu_t\}$ process at time T_1, we simply glue the μ_t, π_t that emerges under path 2 for $t \geq T_1$ to the μ_t, π_t path that had emerged under path 1 for $t = 0, \ldots, T_1 - 1$.

We can do the MIT shock calculations mostly by hand.

Thus, for path 1, $\pi_t = \mu_0$ for all $t \in [0, T_1 - 1]$, while for path 2, $\mu_s = \mu^*$ for all $s \geq T_1$.

We now move on to experiment 2, our "MIT shock", completely unforeseen sudden stabilization.

We set this up so that the $\{\mu_t\}$ sequences that describe the sudden stabilization are identical to those for experiment 1, the foreseen sudden stabilization.

The following code does the calculations and plots outcomes.

```
# path 1
μ_seq_2_path1 = μ0 * np.ones(T+1)

cm1 = create_cagan_model(μ_seq=μ_seq_2_path1)
π_seq_2_path1, m_seq_2_path1, p_seq_2_path1 = solve(cm1, T)

# continuation path
μ_seq_2_cont = μ_star * np.ones(T-T1)

cm2 = create_cagan_model(m0=m_seq_2_path1[T1+1],
                         μ_seq=μ_seq_2_cont)
π_seq_2_cont, m_seq_2_cont1, p_seq_2_cont1 = solve(cm2, T-1-T1)


# regime 1 - simply glue π_seq, μ_seq
μ_seq_2 = np.concatenate((μ_seq_2_path1[:T1+1],
                          μ_seq_2_cont))
π_seq_2 = np.concatenate((π_seq_2_path1[:T1+1],
```

```
                              π_seq_2_cont))
m_seq_2_regime1 = np.concatenate((m_seq_2_path1[:T1+1],
                                  m_seq_2_cont1))
p_seq_2_regime1 = np.concatenate((p_seq_2_path1[:T1+1],
                                  p_seq_2_cont1))


# regime 2 - reset m_T1
m_T1 = (m_seq_2_path1[T1] + μ0) + cm2.α*(μ0 - μ_star)


cm3 = create_cagan_model(m0=m_T1, μ_seq=μ_seq_2_cont)
π_seq_2_cont2, m_seq_2_cont2, p_seq_2_cont2 = solve(cm3, T-1-T1)


m_seq_2_regime2 = np.concatenate((m_seq_2_path1[:T1+1],
                                  m_seq_2_cont2))
p_seq_2_regime2 = np.concatenate((p_seq_2_path1[:T1+1],
                                  p_seq_2_cont2))
```
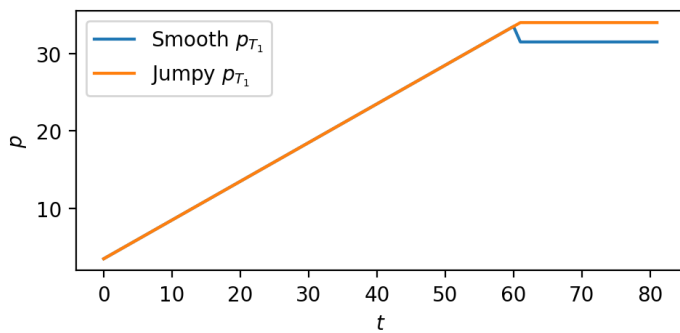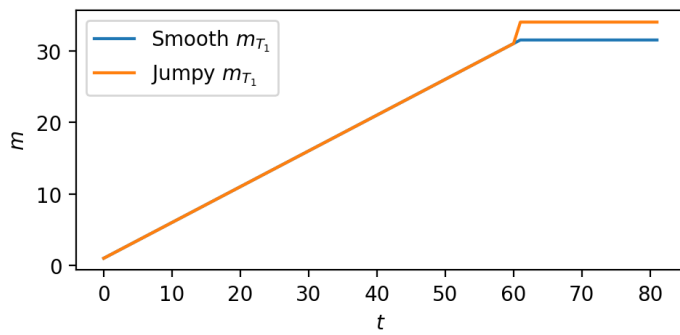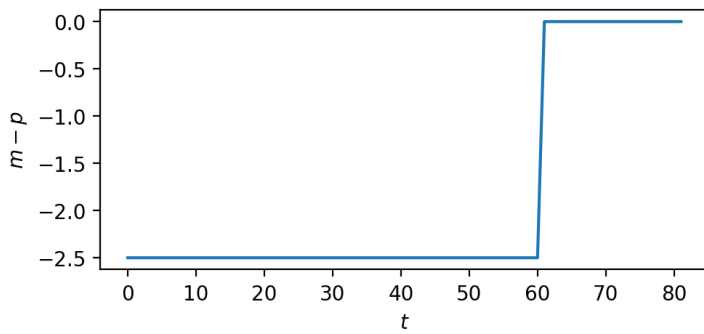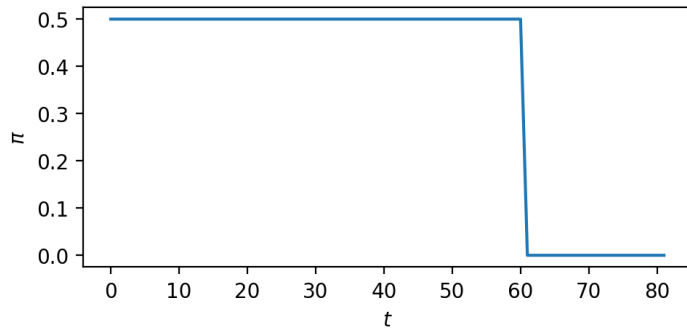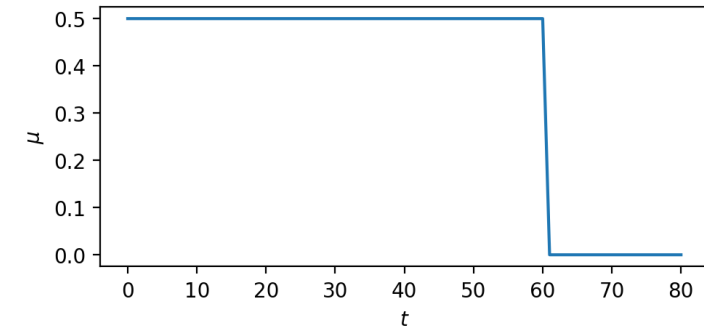
We invite you to compare these graphs with corresponding ones for the foreseen stabilization analyzed in experiment 1 above.

Note how the inflation graph in the second panel is now identical to the money growth graph in the top panel, and how now the log of real balances portrayed in the third panel jumps upward at time T_1.

The bottom two panels plot m and p under two possible ways that m_{T_1} might adjust as required by the upward jump in $m - p$ at T_1.

- the orange line lets m_{T_1} jump upward in order to make sure that the log price level p_{T_1} does not fall.

- the blue line lets p_{T_1} fall while stopping the money supply from jumping.

Here is a way to interpret what the government is doing when the orange line policy is in place.

The government prints money to finance expenditure with the "velocity dividend" that it reaps from the increased demand for real balances brought about by the permanent decrease in the rate of growth of the money supply.

The next code generates a multi-panel graph that includes outcomes of both experiments 1 and 2.

That allows us to assess how important it is to understand whether the sudden permanent drop in μ_t at $t = T_1$ is fully unanticipated, as in experiment 1, or completely unanticipated, as in experiment 2.
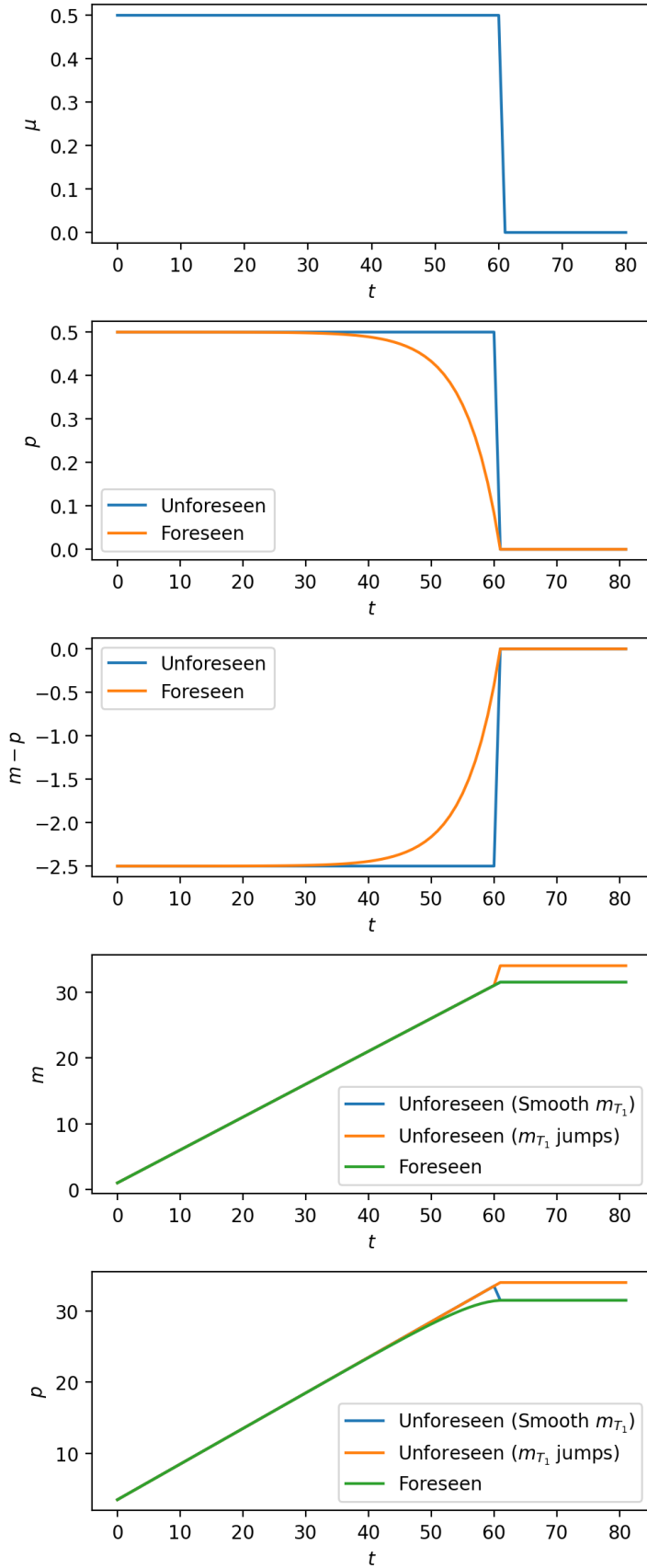
It is instructive to compare the preceding graphs with graphs of log price levels and inflation rates for data from four big inflations described in *this lecture*.

In particular, in the above graphs, notice how a gradual fall in inflation precedes the "sudden stop" when it has been anticipated long beforehand, but how inflation instead falls abruptly when the permanent drop in money supply growth is unanticipated.

It seems to the author team at quantecon that the drops in inflation near the ends of the four hyperinflations described in *this lecture* more closely resemble outcomes from the experiment 2 "unforeseen stabilization".

(It is fair to say that the preceding informal pattern recognition exercise should be supplemented with a more formal structural statistical analysis.)

Experiment 3

Foreseen gradual stabilization

Instead of a foreseen sudden stabilization of the type studied with experiment 1, it is also interesting to study the consequences of a foreseen gradual stabilization.

Thus, suppose that $\phi \in (0,1)$, that $\mu_0 > \mu^*$, and that for $t = 0, \dots, T-1$

$$\mu_t = \phi^t \mu_0 + (1 - \phi^t)\mu^*.$$

Next we perform an experiment in which there is a perfectly foreseen *gradual* decrease in the rate of growth of the money supply.

The following code does the calculations and plots the results.

```
# parameters
ϕ = 0.9
μ_seq_stab = np.array([ϕ**t * μ0 + (1-ϕ**t)*μ_star for t in range(T)])
μ_seq_stab = np.append(μ_seq_stab, μ_star)


cm4 = create_cagan_model(μ_seq=μ_seq_stab)


π_seq_4, m_seq_4, p_seq_4 = solve(cm4, T)


sequences = (μ_seq_stab, π_seq_4,
             m_seq_4 - p_seq_4, m_seq_4, p_seq_4)
plot_sequences(sequences, (r'$\mu$', r'$\pi$',
                           r'$m - p$', r'$m$', r'$p$'))
```
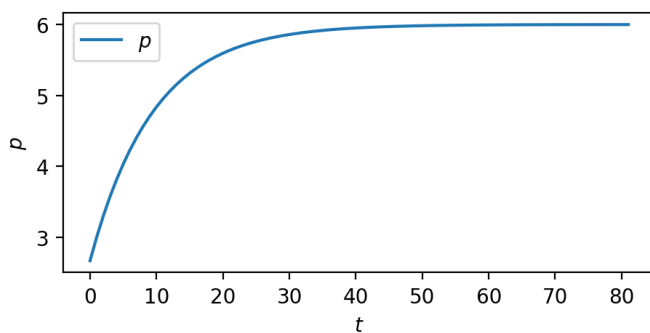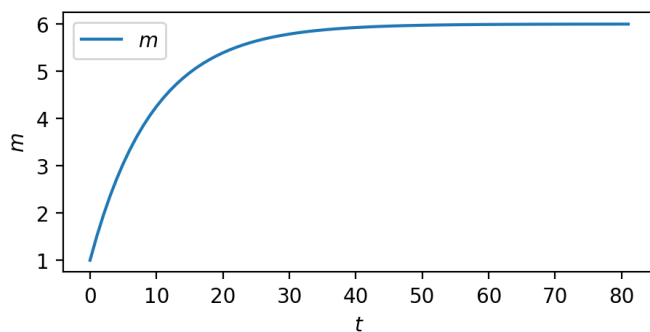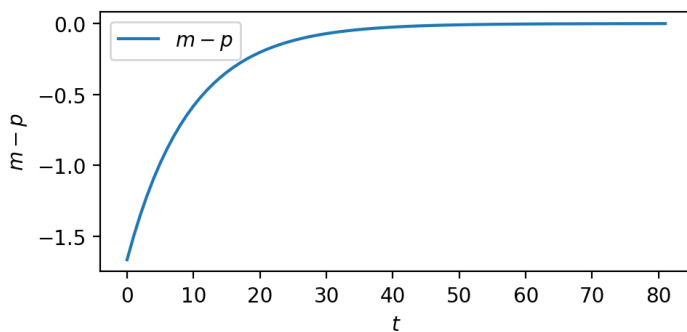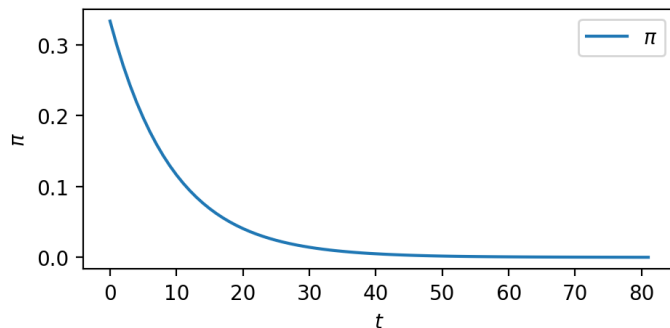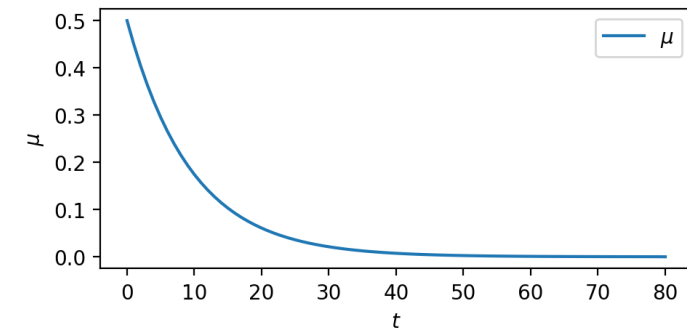
14.4 Sequel

Another lecture *monetarist theory of price levels with adaptive expectations* describes an "adaptive expectations" version of Cagan's model.

The dynamics become more complicated and so does the algebra.

Nowadays, the "rational expectations" version of the model is more popular among central bankers and economists advising them.

FIFTEEN

MONETARIST THEORY OF PRICE LEVELS WITH ADAPTIVE EXPECTATIONS

15.1 Overview

This lecture is a sequel or prequel to *A Monetarist Theory of Price Levels*.

We'll use linear algebra to do some experiments with an alternative "monetarist"or "fiscal"theory of price levels.

Like the model in *A Monetarist Theory of Price Levels*, the model asserts that when a government persistently spends more than it collects in taxes and prints money to finance the shortfall, it puts upward pressure on the price level and generates persistent inflation.

Instead of the "perfect foresight"or "rational expectations"version of the model in *A Monetarist Theory of Price Levels*, our model in the present lecture is an "adaptive expectations"version of a model that [Cagan, 1956] used to study the monetary dynamics of hyperinflations.

It combines these components:

- a demand function for real money balances that asserts that the logarithm of the quantity of real balances demanded depends inversely on the public's expected rate of inflation

- an **adaptive expectations** model that describes how the public's anticipated rate of inflation responds to past values of actual inflation

- an equilibrium condition that equates the demand for money to the supply

- an exogenous sequence of rates of growth of the money supply

Our model stays quite close to Cagan's original specification.

As in *Present Values* and *Consumption Smoothing*, the only linear algebra operations that we'll be using are matrix multiplication and matrix inversion.

To facilitate using linear matrix algebra as our principal mathematical tool, we'll use a finite horizon version of the model.

15.2 Structure of the model

Let

- m_t be the log of the supply of nominal money balances;

- $\mu_t = m_{t+1} - m_t$ be the net rate of growth of nominal balances;

- p_t be the log of the price level;

- $\pi_t = p_{t+1} - p_t$ be the net rate of inflation between t and $t + 1$;

- π_t^* be the public's expected rate of inflation between t and $t + 1$;

- T the horizon –i.e., the last period for which the model will determine p_t

- π_0^* public's initial expected rate of inflation between time 0 and time 1.

The demand for real balances $\exp\left(m_t^d - p_t\right)$ is governed by the following version of the Cagan demand function

$$m_t^d - p_t = -\alpha \pi_t^* \, , \quad \alpha > 0; \quad t = 0, 1, \ldots, T. \tag{15.1}$$

This equation asserts that the demand for real balances is inversely related to the public's expected rate of inflation with sensitivity α.

Equating the logarithm m_t^d of the demand for money to the logarithm m_t of the supply of money in equation (15.1) and solving for the logarithm p_t of the price level gives

$$p_t = m_t + \alpha \pi_t^* \tag{15.2}$$

Taking the difference between equation (15.2) at time $t + 1$ and at time t gives

$$\pi_t = \mu_t + \alpha \pi_{t+1}^* - \alpha \pi_t^* \tag{15.3}$$

We assume that the expected rate of inflation π_t^* is governed by the following adaptive expectations scheme proposed by [Friedman, 1956] and [Cagan, 1956], where $\lambda \in [0, 1]$ denotes the weight on expected inflation.

$$\pi_{t+1}^* = \lambda \pi_t^* + (1 - \lambda) \pi_t \tag{15.4}$$

As exogenous inputs into the model, we take initial conditions m_0, π_0^* and a money growth sequence $\mu = \{\mu_t\}_{t=0}^T$.

As endogenous outputs of our model we want to find sequences $\pi = \{\pi_t\}_{t=0}^T, p = \{p_t\}_{t=0}^T$ as functions of the exogenous inputs.

We'll do some mental experiments by studying how the model outputs vary as we vary the model inputs.

15.3 Representing key equations with linear algebra

We begin by writing the equation (15.4) adaptive expectations model for π_t^* for $t = 0, \dots, T$ as

$$
\begin{bmatrix}
1 & 0 & 0 & \cdots & 0 & 0 \\
-\lambda & 1 & 0 & \cdots & 0 & 0 \\
0 & -\lambda & 1 & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & -\lambda & 1
\end{bmatrix}
\begin{bmatrix}
\pi_0^* \\
\pi_1^* \\
\pi_2^* \\
\vdots \\
\pi_{T+1}^*
\end{bmatrix}
= (1-\lambda)
\begin{bmatrix}
0 & 0 & 0 & \cdots & 0 \\
1 & 0 & 0 & \cdots & 0 \\
0 & 1 & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & \cdots & \vdots \\
0 & 0 & 0 & \cdots & 1
\end{bmatrix}
\begin{bmatrix}
\pi_0 \\
\pi_1 \\
\pi_2 \\
\vdots \\
\pi_T
\end{bmatrix}
+
\begin{bmatrix}
\pi_0^* \\
0 \\
0 \\
\vdots \\
0
\end{bmatrix}
$$

Write this equation as

$$
A\pi^* = (1-\lambda)B\pi + \pi_0^* \tag{15.5}
$$

where the $(T+2) \times (T+2)$ matrix A, the $(T+2) \times (T+1)$ matrix B, and the vectors π^*, π_0, π_0^* are defined implicitly by aligning these two equations.

Next we write the key equation (15.3) in matrix notation as

$$
\begin{bmatrix}
\pi_0 \\
\pi_1 \\
\pi_1 \\
\vdots \\
\pi_T
\end{bmatrix}
=
\begin{bmatrix}
\mu_0 \\
\mu_1 \\
\mu_2 \\
\vdots \\
\mu_T
\end{bmatrix}
+
\begin{bmatrix}
-\alpha & \alpha & 0 & \cdots & 0 & 0 \\
0 & -\alpha & \alpha & \cdots & 0 & 0 \\
0 & 0 & -\alpha & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \cdots & \alpha & 0 \\
0 & 0 & 0 & \cdots & -\alpha & \alpha
\end{bmatrix}
\begin{bmatrix}
\pi_0^* \\
\pi_1^* \\
\pi_2^* \\
\vdots \\
\pi_{T+1}^*
\end{bmatrix}
$$

Represent the previous equation system in terms of vectors and matrices as

$$
\pi = \mu + C\pi^* \tag{15.6}
$$

where the $(T+1) \times (T+2)$ matrix C is defined implicitly to align this equation with the preceding equation system.

15.4 Harvesting insights from our matrix formulation

We now have all of the ingredients we need to solve for π as a function of μ, π_0, π_0^*.

Combine equations (15.5)and (15.6) to get

$$
A\pi^* = (1-\lambda)B\pi + \pi_0^*
$$
$$
= (1-\lambda)B\left[\mu + C\pi^*\right] + \pi_0^*
$$

which implies that

$$
\left[A - (1-\lambda)BC\right]\pi^* = (1-\lambda)B\mu + \pi_0^*
$$

Multiplying both sides of the above equation by the inverse of the matrix on the left side gives

$$
\pi^* = \left[A - (1-\lambda)BC\right]^{-1}\left[(1-\lambda)B\mu + \pi_0^*\right] \tag{15.7}
$$

Having solved equation (15.7) for π^*, we can use equation (15.6) to solve for π:

$$\pi = \mu + C\pi^*$$

We have thus solved for two of the key endogenous time series determined by our model, namely, the sequence π^* of expected inflation rates and the sequence π of actual inflation rates.

Knowing these, we can then quickly calculate the associated sequence p of the logarithm of the price level from equation (15.2).

Let's fill in the details for this step.

Since we now know μ it is easy to compute m.

Thus, notice that we can represent the equations

$$m_{t+1} = m_t + \mu_t, \quad t = 0, 1, \dots, T$$

as the matrix equation

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ -1 & 1 & 0 & \cdots & 0 & 0 \\ 0 & -1 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & 0 & 0 \\ 0 & 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & 0 & \cdots & -1 & 1 \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ \vdots \\ m_T \\ m_{T+1} \end{bmatrix} = \begin{bmatrix} \mu_0 \\ \mu_1 \\ \mu_2 \\ \vdots \\ \mu_{T-1} \\ \mu_T \end{bmatrix} + \begin{bmatrix} m_0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \tag{15.8}$$

Multiplying both sides of equation (15.8) with the inverse of the matrix on the left will give

$$m_t = m_0 + \sum_{s=0}^{t-1} \mu_s, \quad t = 1, \dots, T+1 \tag{15.9}$$

Equation (15.9) shows that the log of the money supply at t equals the log m_0 of the initial money supply plus accumulation of rates of money growth between times 0 and t.

We can then compute p_t for each t from equation (15.2).

We can write a compact formula for p as

$$p = m + \alpha \hat{\pi}^*$$

where

$$\hat{\pi}^* = \begin{bmatrix} \pi_0^* \\ \pi_1^* \\ \pi_2^* \\ \vdots \\ \pi_T^* \end{bmatrix},$$

which is just π^* with the last element dropped.

15.5 Forecast errors and model computation

Our computations will verify that

$$\hat{\pi}^* \neq \pi,$$

so that in general

$$\pi_t^* \neq \pi_t, \quad t = 0, 1, \dots, T \tag{15.10}$$

This outcome is typical in models in which adaptive expectations hypothesis like equation (15.4) appear as a component.

In *A Monetarist Theory of Price Levels*, we studied a version of the model that replaces hypothesis (15.4) with a "perfect foresight" or "rational expectations" hypothesis.

But now, let's dive in and do some computations with the adaptive expectations version of the model.

As usual, we'll start by importing some Python modules.

```
import numpy as np
from collections import namedtuple
import matplotlib.pyplot as plt
```

```
Cagan_Adaptive = namedtuple("Cagan_Adaptive",
                            ["α", "m0", "Eπ0", "T", "λ"])

def create_cagan_adaptive_model(α = 5, m0 = 1, Eπ0 = 0.5, T=80, λ = 0.9):
    return Cagan_Adaptive(α, m0, Eπ0, T, λ)

md = create_cagan_adaptive_model()
```

We solve the model and plot variables of interests using the following functions.

```
def solve_cagan_adaptive(model, μ_seq):
    " Solve the Cagan model in finite time. "
    α, m0, Eπ0, T, λ = model

    A = np.eye(T+2, T+2) - λ*np.eye(T+2, T+2, k=-1)
    B = np.eye(T+2, T+1, k=-1)
    C = -α*np.eye(T+1, T+2) + α*np.eye(T+1, T+2, k=1)
    Eπ0_seq = np.append(Eπ0, np.zeros(T+1))

    # Eπ_seq is of length T+2
    Eπ_seq = np.linalg.solve(A - (1-λ)*B @ C, (1-λ) * B @ μ_seq + Eπ0_seq)
```

(continues on next page)

```
    # π_seq is of length T+1
    π_seq = μ_seq + C @ Eπ_seq

    D = np.eye(T+1, T+1) - np.eye(T+1, T+1, k=-1) # D is the coefficient matrix in␣
↪Equation (14.8)
    m0_seq = np.append(m0, np.zeros(T))

    # m_seq is of length T+2
    m_seq = np.linalg.solve(D, μ_seq + m0_seq)
    m_seq = np.append(m0, m_seq)

    # p_seq is of length T+2
    p_seq = m_seq + α * Eπ_seq

    return π_seq, Eπ_seq, m_seq, p_seq
```

```python
def solve_and_plot(model, μ_seq):

    π_seq, Eπ_seq, m_seq, p_seq = solve_cagan_adaptive(model, μ_seq)

    T_seq = range(model.T+2)

    fig, ax = plt.subplots(5, 1, figsize=[5, 12], dpi=200)
    ax[0].plot(T_seq[:-1], μ_seq)
    ax[1].plot(T_seq[:-1], π_seq, label=r'$\pi_t$')
    ax[1].plot(T_seq, Eπ_seq, label=r'$\pi^{*}_{t}$')
    ax[2].plot(T_seq, m_seq - p_seq)
    ax[3].plot(T_seq, m_seq)
    ax[4].plot(T_seq, p_seq)

    y_labs = [r'$\mu$', r'$\pi$', r'$m - p$', r'$m$', r'$p$']
    subplot_title = [r'Money supply growth', r'Inflation', r'Real balances', r'Money␣
↪supply', r'Price level']

    for i in range(5):
        ax[i].set_xlabel(r'$t$')
        ax[i].set_ylabel(y_labs[i])
        ax[i].set_title(subplot_title[i])

    ax[1].legend()
    plt.tight_layout()
    plt.show()
```

```
    return π_seq, Eπ_seq, m_seq, p_seq
```

15.6 Technical condition for stability

In constructing our examples, we shall assume that (λ, α) satisfy

$$\left| \frac{\lambda - \alpha(1-\lambda)}{1 - \alpha(1-\lambda)} \right| < 1 \tag{15.11}$$

The source of this condition is the following string of deductions:

$$
\begin{aligned}
\pi_t &= \mu_t + \alpha\pi_{t+1}^* - \alpha\pi_t^* \\
\pi_{t+1}^* &= \lambda\pi_t^* + (1-\lambda)\pi_t \\
\pi_t &= \frac{\mu_t}{1 - \alpha(1-\lambda)} - \frac{\alpha(1-\lambda)}{1 - \alpha(1-\lambda)}\pi_t^* \\
\implies \pi_t^* &= \frac{1}{\alpha(1-\lambda)}\mu_t - \frac{1 - \alpha(1-\lambda)}{\alpha(1-\lambda)}\pi_t \\
\pi_{t+1} &= \frac{\mu_{t+1}}{1 - \alpha(1-\lambda)} - \frac{\alpha(1-\lambda)}{1 - \alpha(1-\lambda)}\left(\lambda\pi_t^* + (1-\lambda)\pi_t\right) \\
&= \frac{\mu_{t+1}}{1 - \alpha(1-\lambda)} - \frac{\lambda}{1 - \alpha(1-\lambda)}\mu_t + \frac{\lambda - \alpha(1-\lambda)}{1 - \alpha(1-\lambda)}\pi_t
\end{aligned}
$$

By assuring that the coefficient on π_t is less than one in absolute value, condition (15.11) assures stability of the dynamics of $\{\pi_t\}$ described by the last line of our string of deductions.

The reader is free to study outcomes in examples that violate condition (15.11).

```
print(np.abs((md.λ - md.α*(1-md.λ))/(1 - md.α*(1-md.λ))))
```

```
0.8
```

15.7 Experiments

Now we'll turn to some experiments.

15.7.1 Experiment 1

We'll study a situation in which the rate of growth of the money supply is μ_0 from $t = 0$ to $t = T_1$ and then permanently falls to μ^* at $t = T_1$.

Thus, let $T_1 \in (0, T)$.

So where $\mu_0 > \mu^*$, we assume that

$$
\mu_t = \begin{cases} \mu_0, & t = 0, \dots, T_1 - 1 \\ \mu^*, & t \geq T_1 \end{cases}
$$

Notice that we studied exactly this experiment in a rational expectations version of the model in *A Monetarist Theory of Price Levels*.

So by comparing outcomes across the two lectures, we can learn about consequences of assuming adaptive expectations, as we do here, instead of rational expectations as we assumed in that other lecture.

```
# Parameters for the experiment 1
T1 = 60
μ0 = 0.5
μ_star = 0

μ_seq_1 = np.append(μ0*np.ones(T1), μ_star*np.ones(md.T+1-T1))

# solve and plot
π_seq_1, Eπ_seq_1, m_seq_1, p_seq_1 = solve_and_plot(md, μ_seq_1)
```
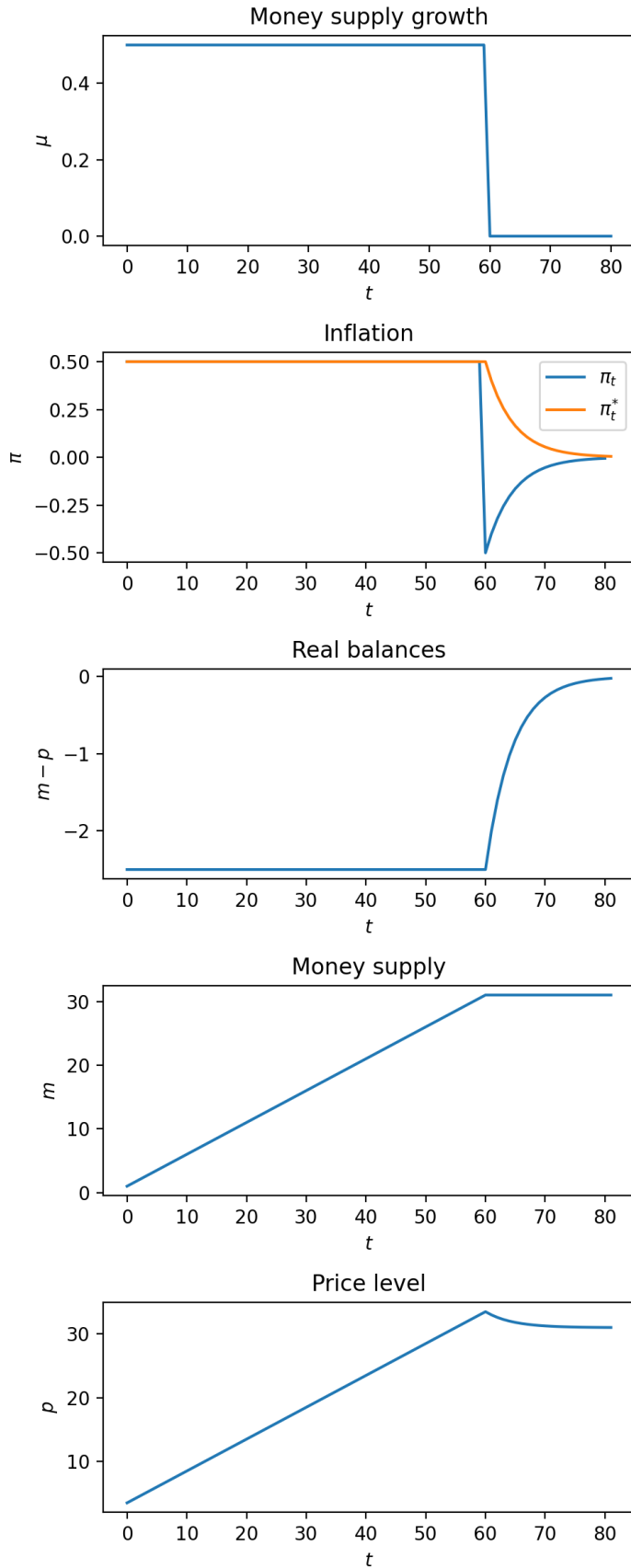
We invite the reader to compare outcomes with those under rational expectations studied in *A Monetarist Theory of Price Levels*.

Please note how the actual inflation rate π_t "overshoots" its ultimate steady-state value at the time of the sudden reduction in the rate of growth of the money supply at time T_1.

We invite you to explain to yourself the source of this overshooting and why it does not occur in the rational expectations version of the model.

15.7.2 Experiment 2

Now we'll do a different experiment, namely, a gradual stabilization in which the rate of growth of the money supply smoothly decline from a high value to a persistently low value.

While price level inflation eventually falls, it falls more slowly than the driving force that ultimately causes it to fall, namely, the falling rate of growth of the money supply.

The sluggish fall in inflation is explained by how anticipated inflation π_t^* persistently exceeds actual inflation π_t during the transition from a high inflation to a low inflation situation.

```
# parameters
ϕ = 0.9
μ_seq_2 = np.array([ϕ**t * μ0 + (1-ϕ**t)*μ_star for t in range(md.T)])
μ_seq_2 = np.append(μ_seq_2, μ_star)


# solve and plot
π_seq_2, Eπ_seq_2, m_seq_2, p_seq_2 = solve_and_plot(md, μ_seq_2)
```
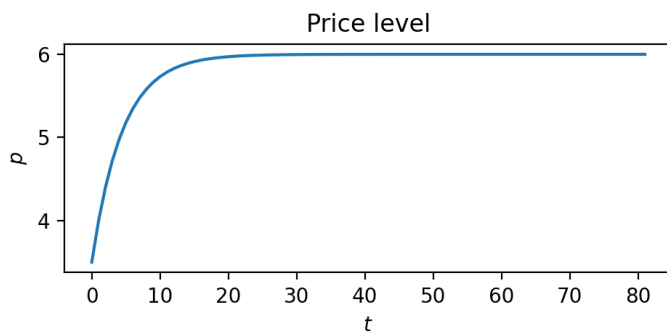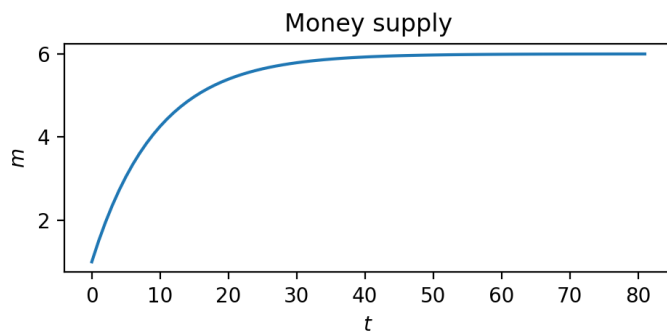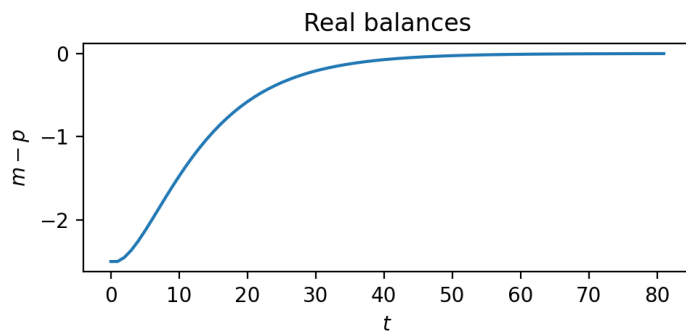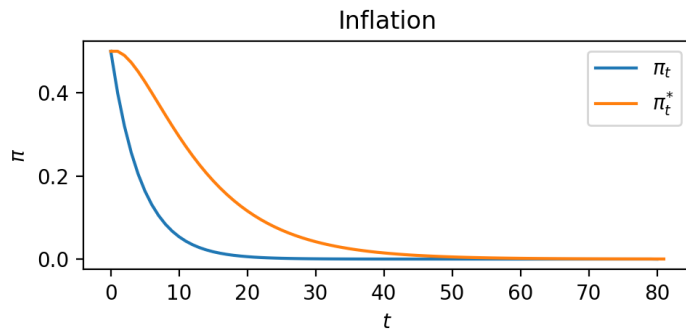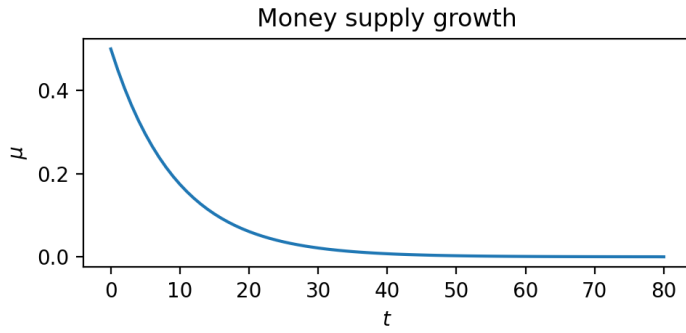
Part V

Linear Dynamics: Infinite Horizons

EIGENVALUES AND EIGENVECTORS

16.1 Overview

Eigenvalues and eigenvectors are a relatively advanced topic in linear algebra.

At the same time, these concepts are extremely useful for

- economic modeling (especially dynamics!)

- statistics

- some parts of applied mathematics

- machine learning

- and many other fields of science.

In this lecture we explain the basics of eigenvalues and eigenvectors and introduce the Neumann Series Lemma.

We assume in this lecture that students are familiar with matrices and understand *the basics of matrix algebra*.

We will use the following imports:

```python
import matplotlib.pyplot as plt
import numpy as np
from numpy.linalg import matrix_power
from matplotlib.lines import Line2D
from matplotlib.patches import FancyArrowPatch
from mpl_toolkits.mplot3d import proj3d
```

16.2 Matrices as transformations

Let's start by discussing an important concept concerning matrices.

16.2.1 Mapping vectors to vectors

One way to think about a matrix is as a rectangular collection of numbers.

Another way to think about a matrix is as a *map* (i.e., as a function) that transforms vectors to new vectors.

To understand the second point of view, suppose we multiply an $n \times m$ matrix A with an $m \times 1$ column vector x to obtain an $n \times 1$ column vector y:

$$Ax = y$$

If we fix A and consider different choices of x, we can understand A as a map transforming x to Ax.

Because A is $n \times m$, it transforms m-vectors to n-vectors.

We can write this formally as $A \colon \mathbb{R}^m \to \mathbb{R}^n$.

You might argue that if A is a function then we should write $A(x) = y$ rather than $Ax = y$ but the second notation is more conventional.

16.2.2 Square matrices

Let's restrict our discussion to square matrices.

In the above discussion, this means that $m = n$ and A maps $\mathbb{R}^n$ to itself.

This means A is an $n \times n$ matrix that maps (or "transforms") a vector x in $\mathbb{R}^n$ to a new vector $y = Ax$ also in $\mathbb{R}^n$.

Example 16.2.1

$$\begin{bmatrix} 2 & 1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \end{bmatrix} = \begin{bmatrix} 5 \\ 2 \end{bmatrix}$$

Here, the matrix

$$A = \begin{bmatrix} 2 & 1 \\ -1 & 1 \end{bmatrix}$$

transforms the vector $x = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$ to the vector $y = \begin{bmatrix} 5 \\ 2 \end{bmatrix}$.

Let's visualize this using Python:

```python
A = np.array([[2,  1],
              [-1, 1]])
```

```python
from math import sqrt

fig, ax = plt.subplots()
# Set the axes through the origin

for spine in ['left', 'bottom']:
    ax.spines[spine].set_position('zero')
for spine in ['right', 'top']:
    ax.spines[spine].set_color('none')

ax.set(xlim=(-2, 6), ylim=(-2, 4), aspect=1)

vecs = ((1, 3), (5, 2))
c = ['r', 'black']
for i, v in enumerate(vecs):
    ax.annotate('', xy=v, xytext=(0, 0),
                arrowprops=dict(color=c[i],
                shrink=0,
                alpha=0.7,
                width=0.5))

ax.text(0.2 + 1, 0.2 + 3, 'x=$(1,3)$')
ax.text(0.2 + 5, 0.2 + 2, 'Ax=$(5,2)$')

ax.annotate('', xy=(sqrt(10/29) * 5, sqrt(10/29) * 2), xytext=(0, 0),
            arrowprops=dict(color='purple',
                            shrink=0,
                            alpha=0.7,
                            width=0.5))

ax.annotate('', xy=(1, 2/5), xytext=(1/3, 1),
            arrowprops={'arrowstyle': '->',
                        'connectionstyle': 'arc3,rad=-0.3'},
            horizontalalignment='center')
ax.text(0.8, 0.8, f'θ', fontsize=14)

plt.show()
```
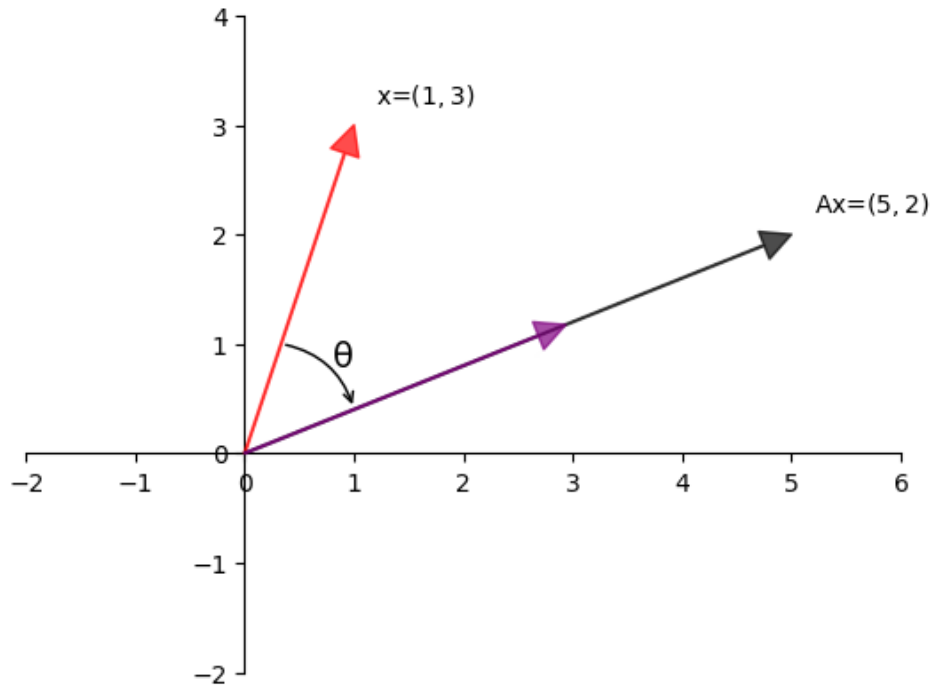
One way to understand this transformation is that A

- first rotates x by some angle θ and

- then scales it by some scalar γ to obtain the image y of x.

16.3 Types of transformations

Let's examine some standard transformations we can perform with matrices.

Below we visualize transformations by thinking of vectors as points instead of arrows.

We consider how a given matrix transforms

- a grid of points and

- a set of points located on the unit circle in $\mathbb{R}^2$.

To build the transformations we will use two functions, called `grid_transform` and `circle_transform`.

Each of these functions visualizes the actions of a given 2×2 matrix A.

16.3.1 Scaling

A matrix of the form

$$\begin{bmatrix} \alpha & 0 \\ 0 & \beta \end{bmatrix}$$

scales vectors across the x-axis by a factor α and along the y-axis by a factor β.

Here we illustrate a simple example where $\alpha = \beta = 3$.

```
A = np.array([[3, 0],   # scaling by 3 in both directions
              [0, 3]])
grid_transform(A)
circle_transform(A)
```
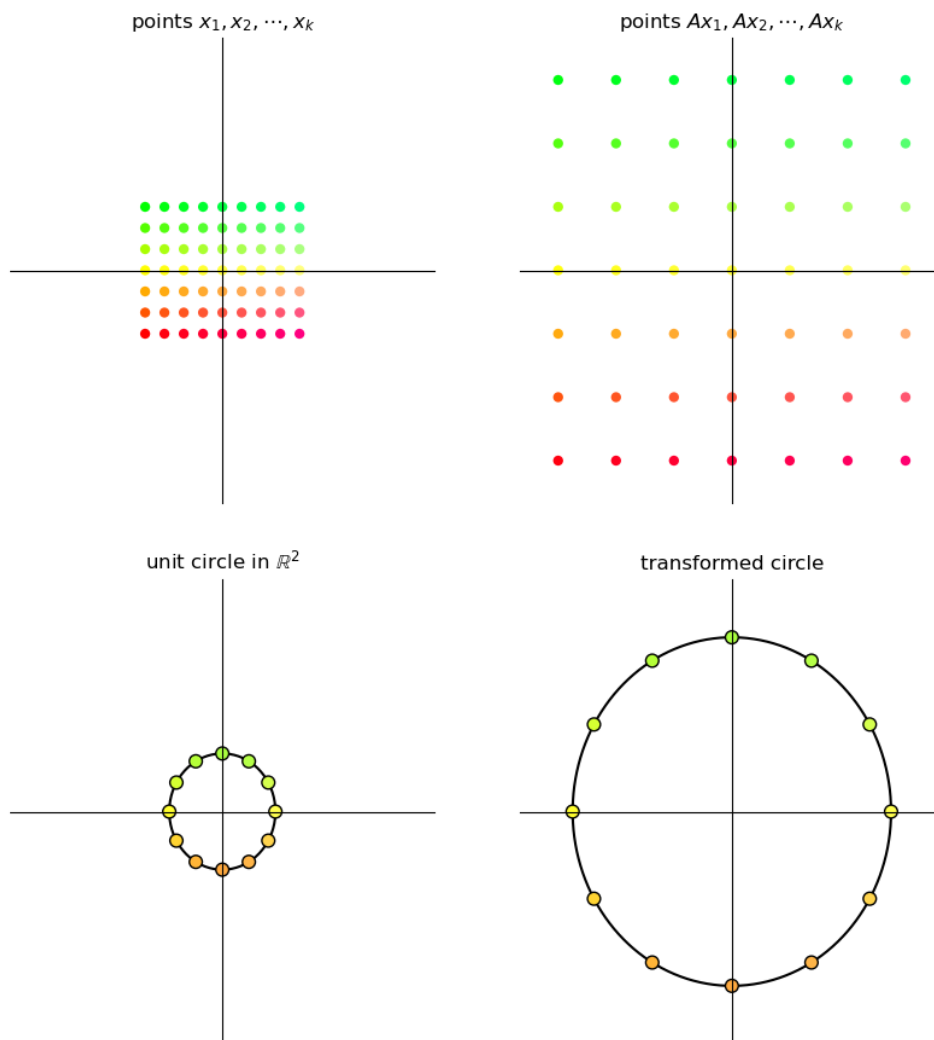
16.3.2 Shearing

A "shear" matrix of the form

$$\begin{bmatrix} 1 & \lambda \\ 0 & 1 \end{bmatrix}$$

stretches vectors along the x-axis by an amount proportional to the y-coordinate of a point.

```
A = np.array([[1, 2],      # shear along x-axis
              [0, 1]])
grid_transform(A)
circle_transform(A)
```
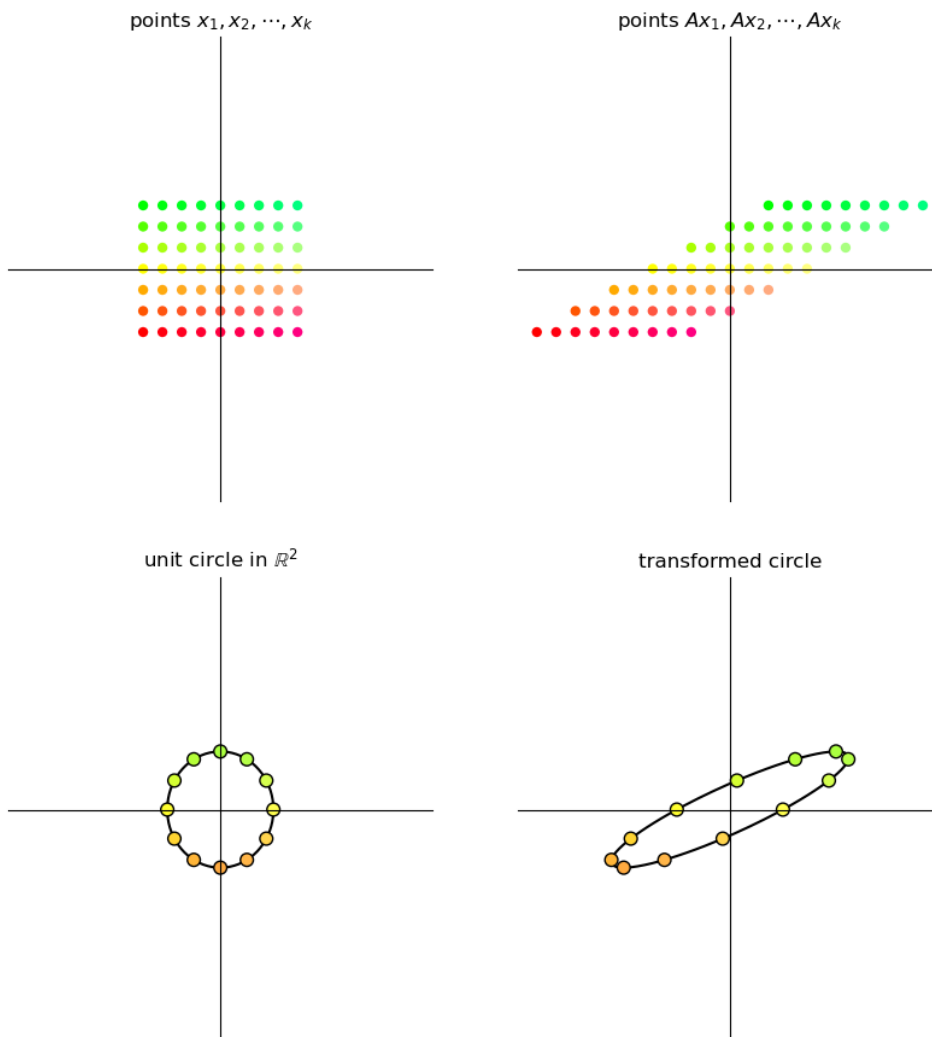
16.3.3 Rotation

A matrix of the form

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

is called a *rotation matrix.*

This matrix rotates vectors clockwise by an angle θ.

```
θ = np.pi/4   # 45 degree clockwise rotation
A = np.array([[np.cos(θ), np.sin(θ)],
              [-np.sin(θ), np.cos(θ)]])
grid_transform(A)
```
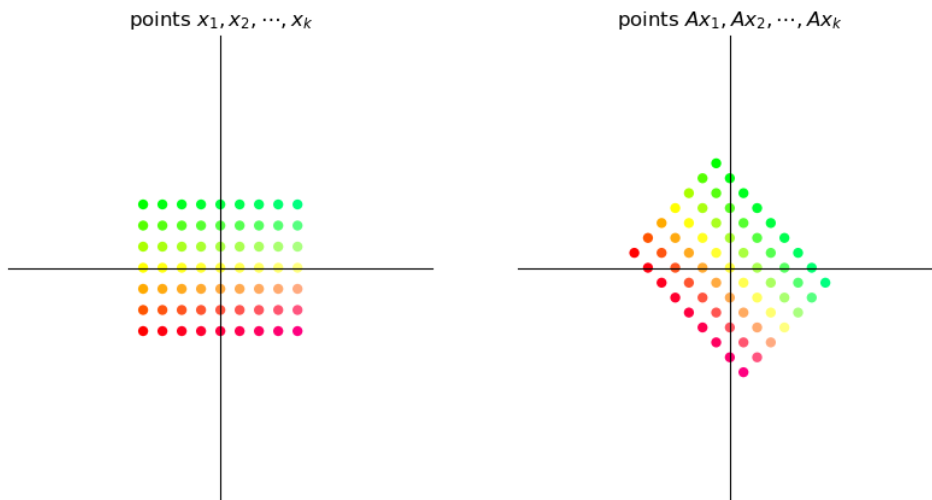

16.3.4 Permutation

The permutation matrix

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

interchanges the coordinates of a vector.

```
A = np.column_stack([[0, 1], [1, 0]])
grid_transform(A)
```

points $x_1, x_2, \cdots, x_k$ points $Ax_1, Ax_2, \cdots, Ax_k$
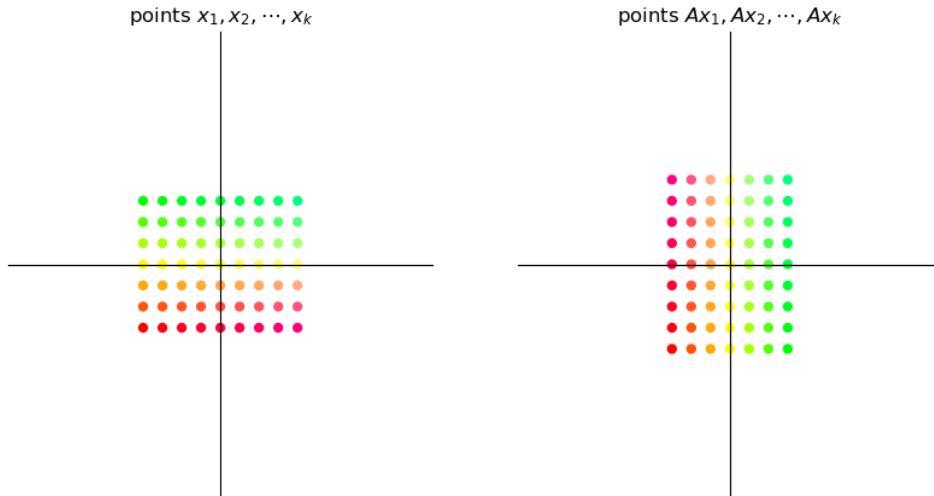
More examples of common transition matrices can be found here.

16.4 Matrix multiplication as composition

Since matrices act as functions that transform one vector to another, we can apply the concept of function composition to matrices as well.

16.4.1 Linear compositions

Consider the two matrices

$$A = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$$

What will the output be when we try to obtain ABx for some 2×1 vector x?

$$\underbrace{\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}}_{B} \overbrace{\begin{bmatrix} 1 \\ 3 \end{bmatrix}}^{x} \rightarrow \underbrace{\begin{bmatrix} 0 & 1 \\ -1 & -2 \end{bmatrix}}_{AB} \overbrace{\begin{bmatrix} 1 \\ 3 \end{bmatrix}}^{x} \rightarrow \overbrace{\begin{bmatrix} 3 \\ -7 \end{bmatrix}}^{y}$$

$$\underbrace{\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}}_{B} \overbrace{\begin{bmatrix} 1 \\ 3 \end{bmatrix}}^{x} \rightarrow \underbrace{\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}}_{A} \overbrace{\begin{bmatrix} 7 \\ 3 \end{bmatrix}}^{Bx} \rightarrow \overbrace{\begin{bmatrix} 3 \\ -7 \end{bmatrix}}^{y}$$

We can observe that applying the transformation AB on the vector x is the same as first applying B on x and then applying A on the vector Bx.

Thus the matrix product AB is the composition of the matrix transformations A and B

This means first apply transformation B and then transformation A.

When we matrix multiply an $n \times m$ matrix A with an $m \times k$ matrix B the obtained matrix product is an $n \times k$ matrix AB.

Thus, if A and B are transformations such that $A \colon \mathbb{R}^m \to \mathbb{R}^n$ and $B \colon \mathbb{R}^k \to \mathbb{R}^m$, then AB transforms $\mathbb{R}^k$ to $\mathbb{R}^n$.

Viewing matrix multiplication as composition of maps helps us understand why, under matrix multiplication, AB is generally not equal to BA.

(After all, when we compose functions, the order usually matters.)

16.4.2 Examples

Let A be the 90° clockwise rotation matrix given by $\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$ and let B be a shear matrix along the x-axis given by $\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$.

We will visualize how a grid of points changes when we apply the transformation AB and then compare it with the transformation BA.

```
A = np.array([[0, 1],      # 90 degree clockwise rotation
              [-1, 0]])
B = np.array([[1, 2],      # shear along x-axis
              [0, 1]])
```

Shear then rotate

```
grid_composition_transform(A, B)   # transformation AB
```

Rotate then shear

```
grid_composition_transform(B,A)          # transformation BA
```
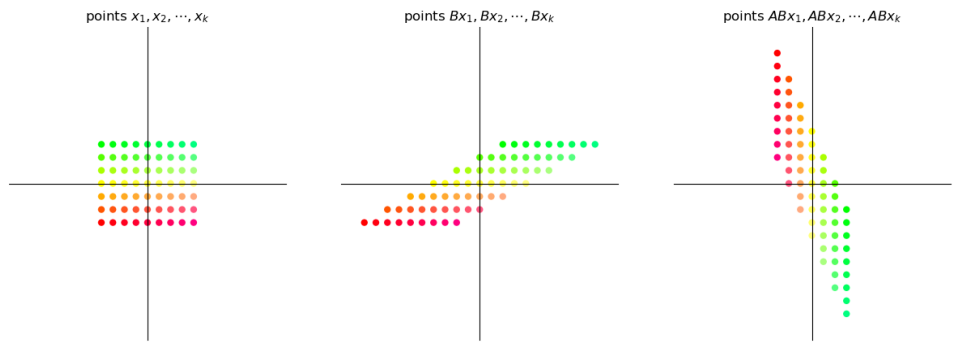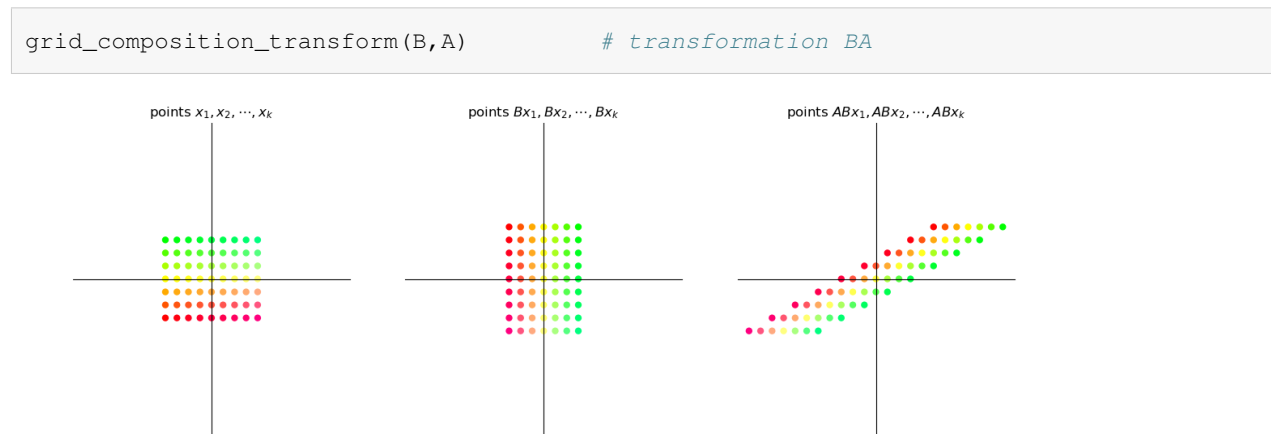

It is evident that the transformation AB is not the same as the transformation BA.

16.5 Iterating on a fixed map

In economics (and especially in dynamic modeling), we are often interested in analyzing behavior where we repeatedly apply a fixed matrix.

For example, given a vector v and a matrix A, we are interested in studying the sequence

$$v, \quad Av, \quad AAv = A^2v, \quad \dots$$

Let's first see examples of a sequence of iterates $(A^k v)_{k \geq 0}$ under different maps A.

```python
def plot_series(A, v, n):

    B = np.array([[1, -1],
                  [1, 0]])

    fig, ax = plt.subplots()

    ax.set(xlim=(-4, 4), ylim=(-4, 4))
    ax.set_xticks([])
    ax.set_yticks([])
    for spine in ['left', 'bottom']:
        ax.spines[spine].set_position('zero')
    for spine in ['right', 'top']:
        ax.spines[spine].set_color('none')

    θ = np.linspace(0, 2 * np.pi, 150)
```

(continues on next page)

```python
    r = 2.5
    x = r * np.cos(θ)
    y = r * np.sin(θ)
    x1 = x.reshape(1, -1)
    y1 = y.reshape(1, -1)
    xy = np.concatenate((x1, y1), axis=0)

    ellipse = B @ xy
    ax.plot(ellipse[0, :], ellipse[1, :], color='black',
            linestyle=(0, (5, 10)), linewidth=0.5)

    # Initialize holder for trajectories
    colors = plt.cm.rainbow(np.linspace(0, 1, 20))

    for i in range(n):
        iteration = matrix_power(A, i) @ v
        v1 = iteration[0]
        v2 = iteration[1]
        ax.scatter(v1, v2, color=colors[i])
        if i == 0:
            ax.text(v1+0.25, v2, f'$v$')
        elif i == 1:
            ax.text(v1+0.25, v2, f'$Av$')
        elif 1 < i < 4:
            ax.text(v1+0.25, v2, f'$A^{i}v$')
    plt.show()
```

```python
A = np.array([[sqrt(3) + 1, -2],
              [1, sqrt(3) - 1]])
A = (1/(2*sqrt(2))) * A
v = (-3, -3)
n = 12

plot_series(A, v, n)
```
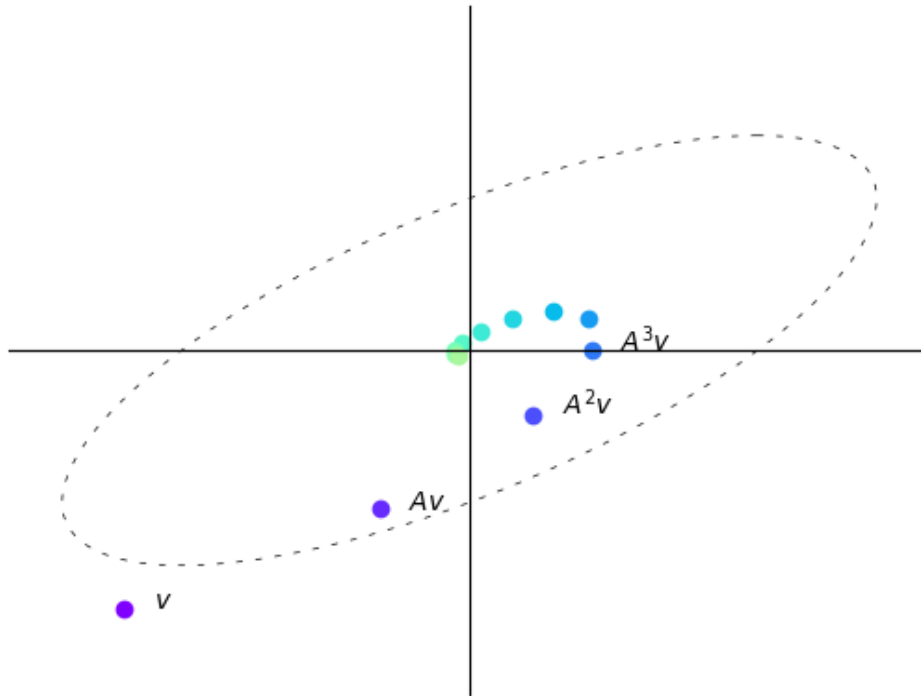
Here with each iteration the vectors get shorter, i.e., move closer to the origin.

In this case, repeatedly multiplying a vector by A makes the vector "spiral in".

```
B = np.array([[sqrt(3) + 1, -2],
              [1, sqrt(3) - 1]])
B = (1/2) * B
v = (2.5, 0)
n = 12

plot_series(B, v, n)
```
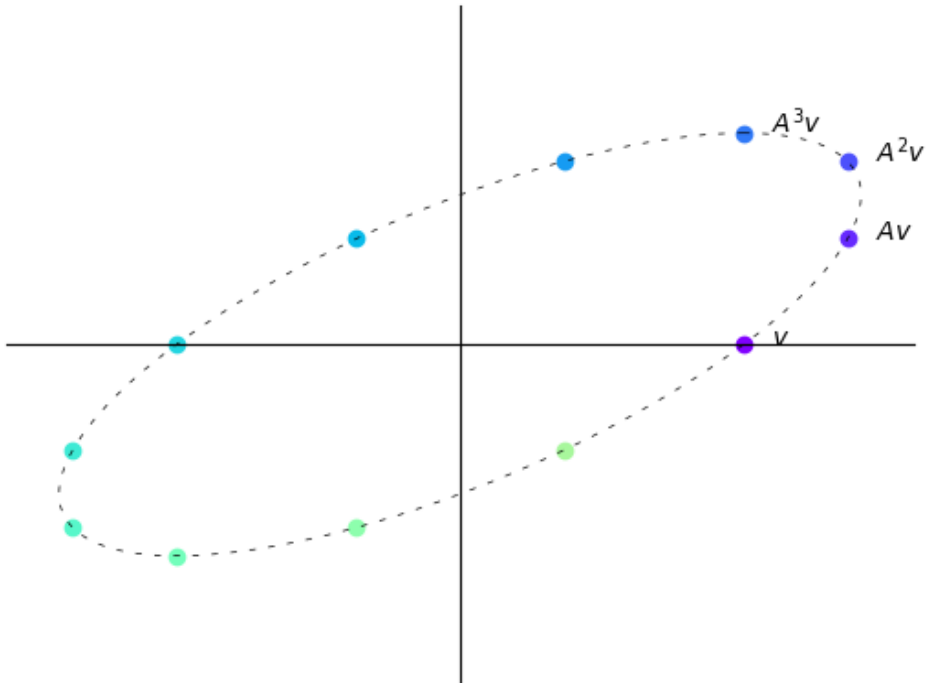
Here with each iteration vectors do not tend to get longer or shorter.

In this case, repeatedly multiplying a vector by A simply "rotates it around an ellipse".

```
B = np.array([[sqrt(3) + 1, -2],
              [1, sqrt(3) - 1]])
B = (1/sqrt(2)) * B
v = (-1, -0.25)
n = 6

plot_series(B, v, n)
```
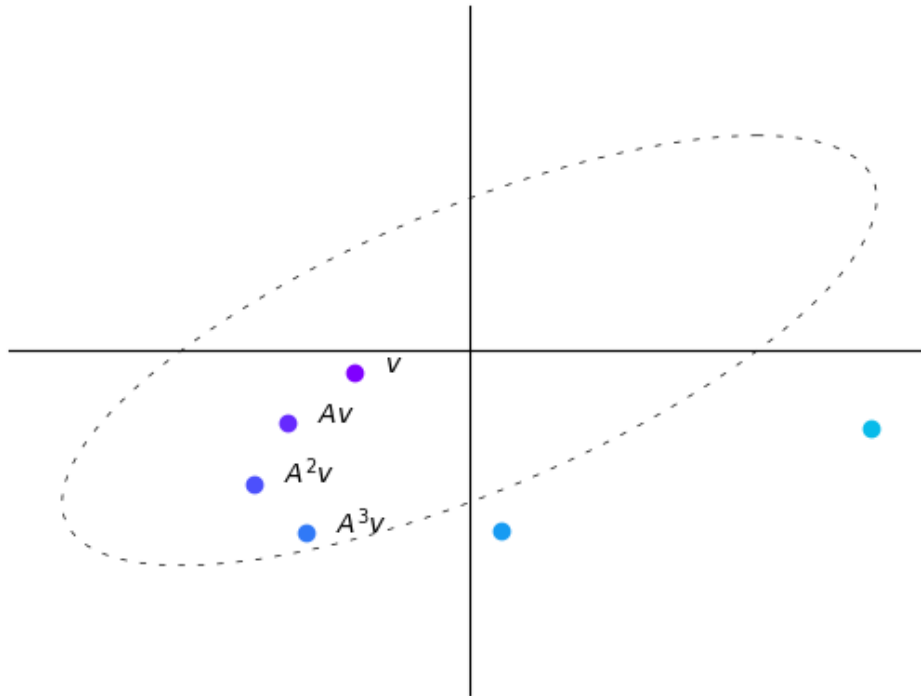
Here with each iteration vectors tend to get longer, i.e., farther from the origin.

In this case, repeatedly multiplying a vector by A makes the vector "spiral out".

We thus observe that the sequence $(A^k v)_{k \geq 0}$ behaves differently depending on the map A itself.

We now discuss the property of A that determines this behavior.

16.6 Eigenvalues

In this section we introduce the notions of eigenvalues and eigenvectors.

16.6.1 Definitions

Let A be an $n \times n$ square matrix.

If λ is scalar and v is a non-zero n-vector such that

$$Av = \lambda v.$$

Then we say that λ is an *eigenvalue* of A, and v is the corresponding *eigenvector*.

Thus, an eigenvector of A is a nonzero vector v such that when the map A is applied, v is merely scaled.

The next figure shows two eigenvectors (blue arrows) and their images under A (red arrows).

As expected, the image Av of each v is just a scaled version of the original

```
from numpy.linalg import eig

A = [[1, 2],
     [2, 1]]
A = np.array(A)
evals, evecs = eig(A)
evecs = evecs[:, 0], evecs[:, 1]

fig, ax = plt.subplots(figsize=(10, 8))
# Set the axes through the origin
for spine in ['left', 'bottom']:
    ax.spines[spine].set_position('zero')
for spine in ['right', 'top']:
    ax.spines[spine].set_color('none')
# ax.grid(alpha=0.4)

xmin, xmax = -3, 3
ymin, ymax = -3, 3
ax.set(xlim=(xmin, xmax), ylim=(ymin, ymax))

# Plot each eigenvector
for v in evecs:
    ax.annotate('', xy=v, xytext=(0, 0),
                arrowprops=dict(facecolor='blue',
                shrink=0,
                alpha=0.6,
                width=0.5))

# Plot the image of each eigenvector
for v in evecs:
    v = A @ v
    ax.annotate('', xy=v, xytext=(0, 0),
                arrowprops=dict(facecolor='red',
                shrink=0,
                alpha=0.6,
                width=0.5))

# Plot the lines they run through
x = np.linspace(xmin, xmax, 3)
for v in evecs:
    a = v[1] / v[0]
    ax.plot(x, a * x, 'b-', lw=0.4)
```

```
plt.show()
```
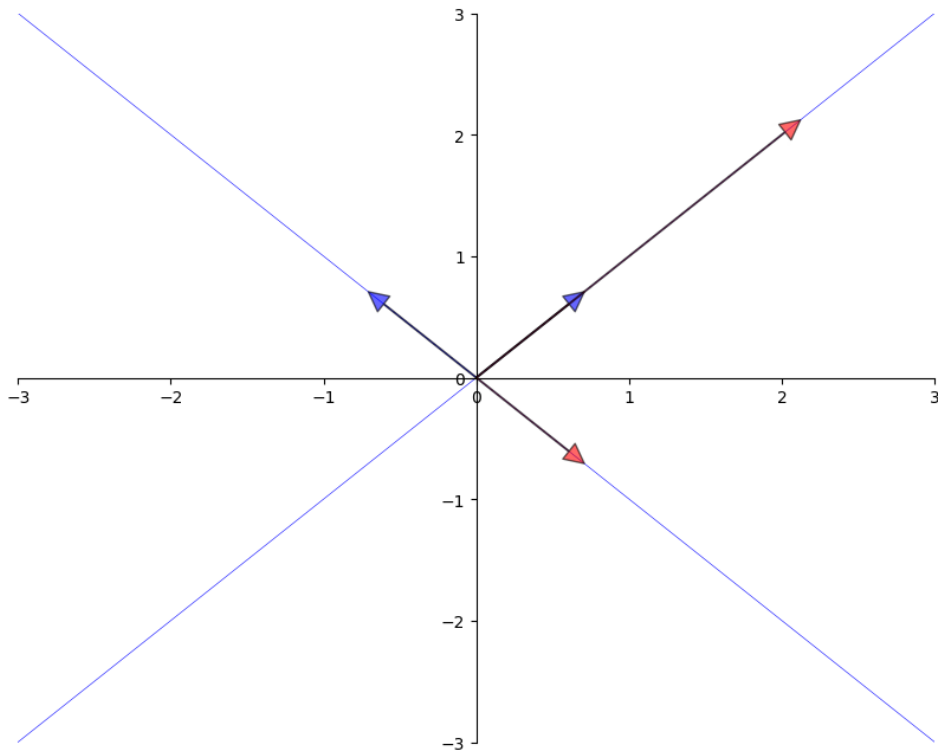

16.6.2 Complex values

So far our definition of eigenvalues and eigenvectors seems straightforward.

There is one complication we haven't mentioned yet:

When solving $Av = \lambda v$,

- λ is allowed to be a complex number and

- v is allowed to be an n-vector of complex numbers.

We will see some examples below.

16.6.3 Some mathematical details

We note some mathematical details for more advanced readers.

(Other readers can skip to the next section.)

The eigenvalue equation is equivalent to $(A - \lambda I)v = 0$.

This equation has a nonzero solution v only when the columns of $A - \lambda I$ are linearly dependent.

This in turn is equivalent to stating the determinant is zero.

Hence, to find all eigenvalues, we can look for λ such that the determinant of $A - \lambda I$ is zero.

This problem can be expressed as one of solving for the roots of a polynomial in λ of degree n.

This in turn implies the existence of n solutions in the complex plane, although some might be repeated.

16.6.4 Facts

Some nice facts about the eigenvalues of a square matrix A are as follows:

1. the determinant of A equals the product of the eigenvalues

2. the trace of A (the sum of the elements on the principal diagonal) equals the sum of the eigenvalues

3. if A is symmetric, then all of its eigenvalues are real

4. if A is invertible and $\lambda_1, \dots, \lambda_n$ are its eigenvalues, then the eigenvalues of A^{-1} are $1/\lambda_1, \dots, 1/\lambda_n$.

A corollary of the last statement is that a matrix is invertible if and only if all its eigenvalues are nonzero.

16.6.5 Computation

Using NumPy, we can solve for the eigenvalues and eigenvectors of a matrix as follows

```python
from numpy.linalg import eig

A = ((1, 2),
     (2, 1))

A = np.array(A)
evals, evecs = eig(A)
evals  # eigenvalues
```

```
array([ 3., -1.])
```

```
evecs   # eigenvectors
```

```
array([[ 0.70710678, -0.70710678],
       [ 0.70710678,  0.70710678]])
```

Note that the *columns* of `evecs` are the eigenvectors.

Since any scalar multiple of an eigenvector is an eigenvector with the same eigenvalue (which can be verified), the `eig` routine normalizes the length of each eigenvector to one.

The eigenvectors and eigenvalues of a map A determine how a vector v is transformed when we repeatedly multiply by A.

This is discussed further later.

16.7 The Neumann Series Lemma

In this section we present a famous result about series of matrices that has many applications in economics.

16.7.1 Scalar series

Here's a fundamental result about series:

If a is a number and $|a| < 1$, then

$$\sum_{k=0}^{\infty} a^k = \frac{1}{1-a} = (1-a)^{-1} \tag{16.1}$$

For a one-dimensional linear equation $x = ax + b$ where x is unknown we can thus conclude that the solution x^* is given by:

$$x^* = \frac{b}{1-a} = \sum_{k=0}^{\infty} a^k b$$

16.7.2 Matrix series

A generalization of this idea exists in the matrix setting.

Consider the system of equations $x = Ax + b$ where A is an $n \times n$ square matrix and x and b are both column vectors in $\mathbb{R}^n$.

Using matrix algebra we can conclude that the solution to this system of equations will be given by:

$$x^* = (I - A)^{-1}b \tag{16.2}$$

What guarantees the existence of a unique vector x^* that satisfies (16.2)?

The following is a fundamental result in functional analysis that generalizes (16.1) to a multivariate case.

Theorem 16.7.1 (Neumann Series Lemma)

Let A be a square matrix and let A^k be the k-th power of A.

Let $r(A)$ be the **spectral radius** of A, defined as $\max_i |\lambda_i|$, where

- $\{\lambda_i\}_i$ is the set of eigenvalues of A and

- $|\lambda_i|$ is the modulus of the complex number λ_i

Neumann's Theorem states the following: If $r(A) < 1$, then $I - A$ is invertible, and

$$(I - A)^{-1} = \sum_{k=0}^{\infty} A^k$$

We can see the Neumann Series Lemma in action in the following example.

```
A = np.array([[0.4, 0.1],
              [0.7, 0.2]])

evals, evecs = eig(A)    # finding eigenvalues and eigenvectors

r = max(abs(λ) for λ in evals)    # compute spectral radius
print(r)
```

```
0.5828427124746189
```

The spectral radius $r(A)$ obtained is less than 1.

Thus, we can apply the Neumann Series Lemma to find $(I - A)^{-1}$.

```
I = np.identity(2)   # 2 x 2 identity matrix
B = I - A
```

```
B_inverse = np.linalg.inv(B)   # direct inverse method
```

```
A_sum = np.zeros((2, 2))   # power series sum of A
A_power = I
for i in range(50):
```

16.7. The Neumann Series Lemma 315

```
    A_sum += A_power
    A_power = A_power @ A
```

Let's check equality between the sum and the inverse methods.

```
np.allclose(A_sum, B_inverse)
```

```
True
```

Although we truncate the infinite sum at $k = 50$, both methods give us the same result which illustrates the result of the Neumann Series Lemma.

16.8 Exercises

Exercise 16.8.1

Power iteration is a method for finding the greatest absolute eigenvalue of a diagonalizable matrix.

The method starts with a random vector b_0 and repeatedly applies the matrix A to it

$$b_{k+1} = \frac{Ab_k}{\|Ab_k\|}$$

A thorough discussion of the method can be found here.

In this exercise, first implement the power iteration method and use it to find the greatest absolute eigenvalue and its corresponding eigenvector.

Then visualize the convergence.

Solution to Exercise 16.8.1

Here is one solution.

We start by looking into the distance between the eigenvector approximation and the true eigenvector.

```
# Define a matrix A
A = np.array([[1, 0, 3],
              [0, 2, 0],
              [3, 0, 1]])
```

```python
num_iters = 20

# Define a random starting vector b
b = np.random.rand(A.shape[1])

# Get the leading eigenvector of matrix A
eigenvector = np.linalg.eig(A)[1][:, 0]

errors = []
res = []

# Power iteration loop
for i in range(num_iters):
    # Multiply b by A
    b = A @ b
    # Normalize b
    b = b / np.linalg.norm(b)
    # Append b to the list of eigenvector approximations
    res.append(b)
    err = np.linalg.norm(np.array(b)
                         - eigenvector)
    errors.append(err)

greatest_eigenvalue = np.dot(A @ b, b) / np.dot(b, b)
print(f'The approximated greatest absolute eigenvalue is \
        {greatest_eigenvalue:.2f}')
print('The real eigenvalue is', np.linalg.eig(A)[0])

# Plot the eigenvector approximations for each iteration
plt.figure(figsize=(10, 6))
plt.xlabel('iterations')
plt.ylabel('error')
_ = plt.plot(errors)
```

```
The approximated greatest absolute eigenvalue is        4.00
The real eigenvalue is [ 4. -2.  2.]
```

Then we can look at the trajectory of the eigenvector approximation.

```python
# Set up the figure and axis for 3D plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```
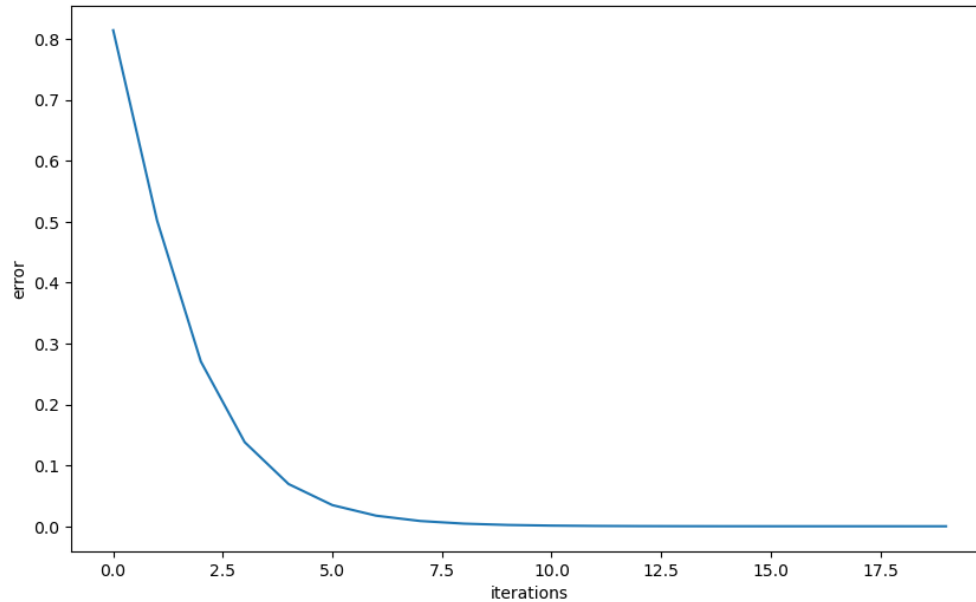
Fig. 16.1: Power iteration

```python
# Plot the eigenvectors
ax.scatter(eigenvector[0],
           eigenvector[1],
           eigenvector[2],
           color='r', s=80)


for i, vec in enumerate(res):
    ax.scatter(vec[0], vec[1], vec[2],
               color='b',
               alpha=(i+1)/(num_iters+1),
               s=80)


ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.tick_params(axis='both', which='major', labelsize=7)


points = [plt.Line2D([0], [0], linestyle='none',
                     c=i, marker='o') for i in ['r', 'b']]
ax.legend(points, ['actual eigenvector',
                   r'approximated eigenvector ($b_k$)'])
ax.set_box_aspect(aspect=None, zoom=0.8)
```

```
plt.show()
```
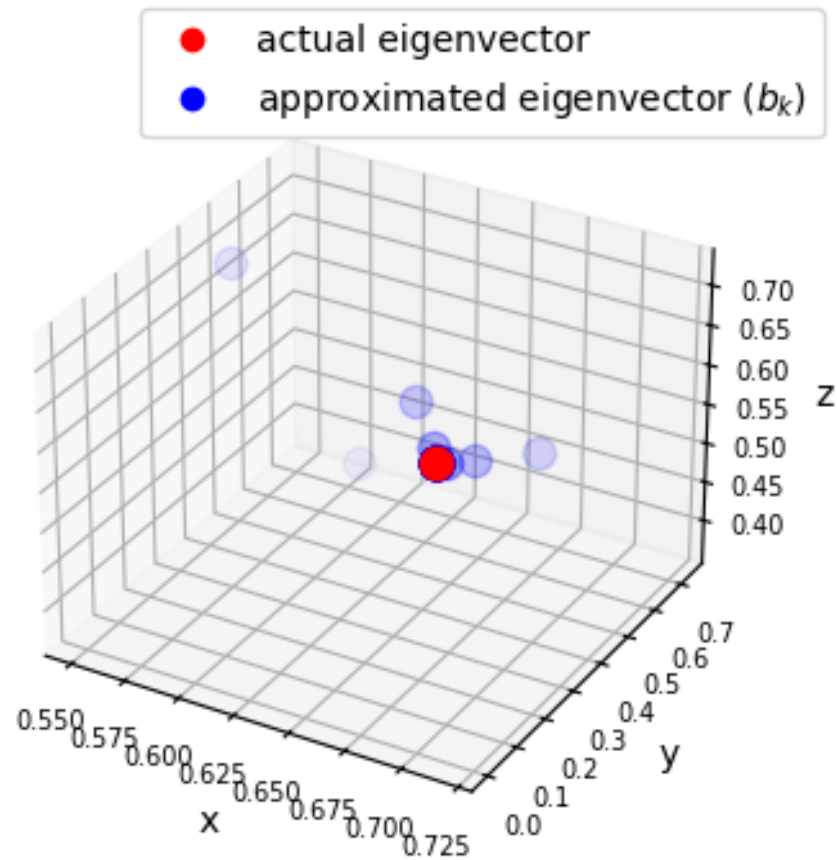

Fig. 16.2: Power iteration trajectory

Exercise 16.8.2

We have discussed the trajectory of the vector v after being transformed by A.

Consider the matrix $A = \begin{bmatrix} 1 & 2 \\ 1 & 1 \end{bmatrix}$ and the vector $v = \begin{bmatrix} 2 \\ -2 \end{bmatrix}$.

Try to compute the trajectory of v after being transformed by A for $n = 4$ iterations and plot the result.

Solution to Exercise 16.8.2

```python
A = np.array([[1, 2],
              [1, 1]])
v = (0.4, -0.4)
n = 11

# Compute eigenvectors and eigenvalues
eigenvalues, eigenvectors = np.linalg.eig(A)

print(f'eigenvalues:\n {eigenvalues}')
print(f'eigenvectors:\n {eigenvectors}')

plot_series(A, v, n)
```

```
eigenvalues:
 [ 2.41421356 -0.41421356]
eigenvectors:
 [[ 0.81649658 -0.81649658]
 [ 0.57735027  0.57735027]]
```
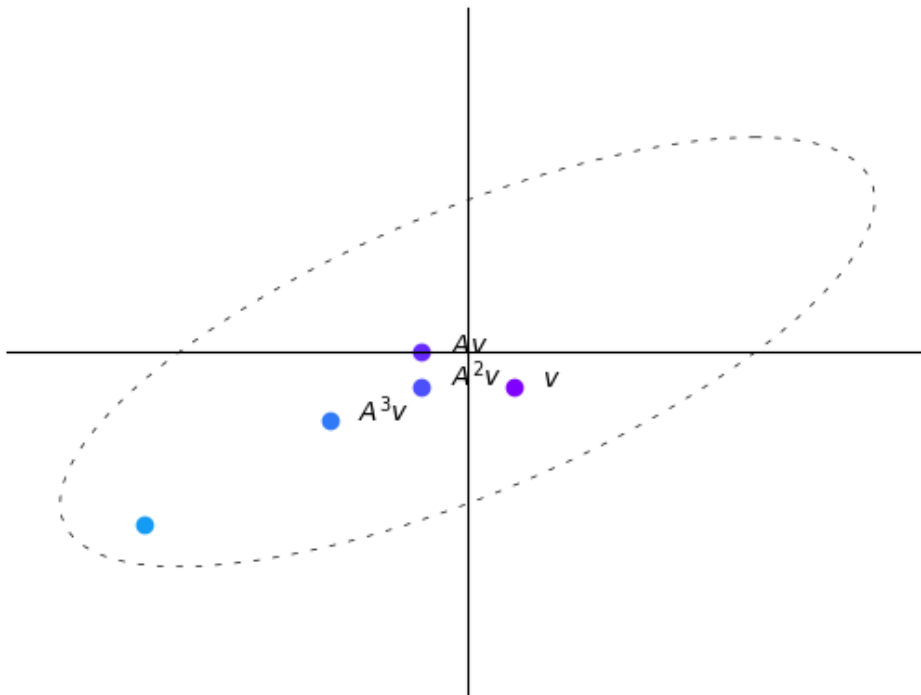

The result seems to converge to the eigenvector of A with the largest eigenvalue.

Let's use a vector field to visualize the transformation brought by A.

(This is a more advanced topic in linear algebra, please step ahead if you are comfortable with the math.)

```python
# Create a grid of points
x, y = np.meshgrid(np.linspace(-5, 5, 15),
                   np.linspace(-5, 5, 20))


# Apply the matrix A to each point in the vector field
vec_field = np.stack([x, y])
u, v = np.tensordot(A, vec_field, axes=1)


# Plot the transformed vector field
c = plt.streamplot(x, y, u - x, v - y,
                   density=1, linewidth=None, color='#A23BEC')
c.lines.set_alpha(0.5)
c.arrows.set_alpha(0.5)


# Draw eigenvectors
origin = np.zeros((2, len(eigenvectors)))
parameters = {'color': ['b', 'g'], 'angles': 'xy',
              'scale_units': 'xy', 'scale': 0.1, 'width': 0.01}
plt.quiver(*origin, eigenvectors[0],
           eigenvectors[1], **parameters)
plt.quiver(*origin, - eigenvectors[0],
           - eigenvectors[1], **parameters)


colors = ['b', 'g']
lines = [Line2D([0], [0], color=c, linewidth=3) for c in colors]
labels = ["2.4 eigenspace", "0.4 eigenspace"]
plt.legend(lines, labels, loc='center left',
           bbox_to_anchor=(1, 0.5))


plt.xlabel("x")
plt.ylabel("y")
plt.grid()
plt.gca().set_aspect('equal', adjustable='box')
plt.show()
```

Note that the vector field converges to the eigenvector of A with the largest eigenvalue and diverges from the eigenvector of A with the smallest eigenvalue.

In fact, the eigenvectors are also the directions in which the matrix A stretches or shrinks the space.

Specifically, the eigenvector with the largest eigenvalue is the direction in which the matrix A stretches the space the most.

16.8. Exercises
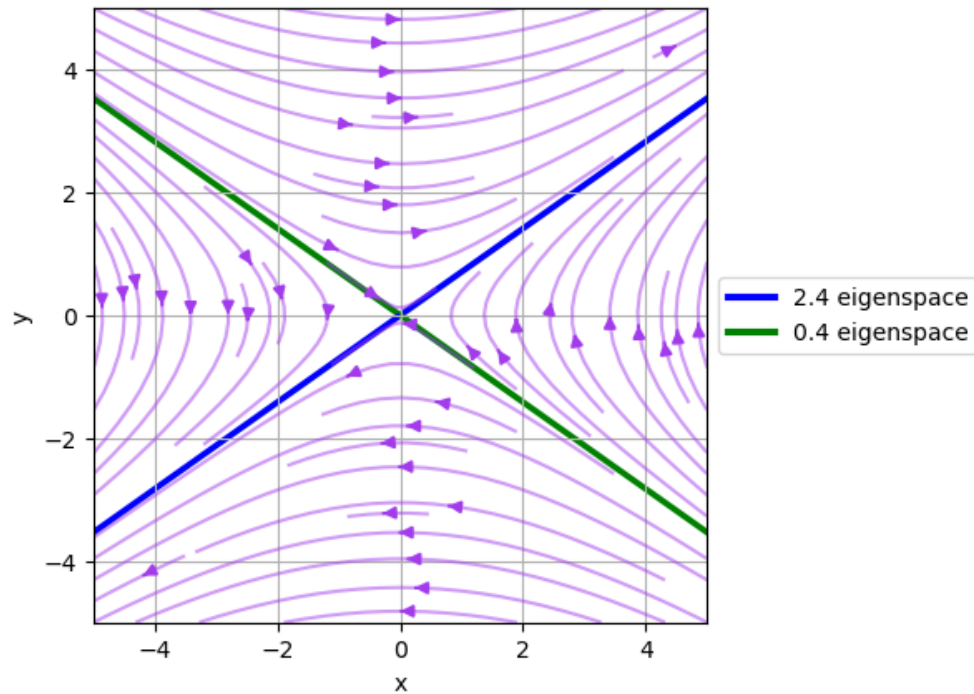
Fig. 16.3: Convergence towards eigenvectors

We will see more intriguing examples in the following exercise.

Exercise 16.8.3

Previously, we demonstrated the trajectory of the vector v after being transformed by A for three different matrices.

Use the visualization in the previous exercise to explain the trajectory of the vector v after being transformed by A for the three different matrices.

Solution to Exercise 16.8.3

Here is one solution

```
figure, ax = plt.subplots(1, 3, figsize=(15, 5))
A = np.array([[sqrt(3) + 1, -2],
             [1, sqrt(3) - 1]])
A = (1/(2*sqrt(2))) * A

B = np.array([[sqrt(3) + 1, -2],
             [1, sqrt(3) - 1]])
```

```python
B = (1/2) * B


C = np.array([[sqrt(3) + 1, -2],
              [1, sqrt(3) - 1]])
C = (1/sqrt(2)) * C


examples = [A, B, C]


for i, example in enumerate(examples):
    M = example

    # Compute right eigenvectors and eigenvalues
    eigenvalues, eigenvectors = np.linalg.eig(M)
    print(f'Example {i+1}:\n')
    print(f'eigenvalues:\n {eigenvalues}')
    print(f'eigenvectors:\n {eigenvectors}\n')


    eigenvalues_real = eigenvalues.real
    eigenvectors_real = eigenvectors.real


    # Create a grid of points
    x, y = np.meshgrid(np.linspace(-20, 20, 15),
                       np.linspace(-20, 20, 20))


    # Apply the matrix A to each point in the vector field
    vec_field = np.stack([x, y])
    u, v = np.tensordot(M, vec_field, axes=1)


    # Plot the transformed vector field
    c = ax[i].streamplot(x, y, u - x, v - y, density=1,
                         linewidth=None, color='#A23BEC')
    c.lines.set_alpha(0.5)
    c.arrows.set_alpha(0.5)


    # Draw eigenvectors
    parameters = {'color': ['b', 'g'], 'angles': 'xy',
                  'scale_units': 'xy', 'scale': 1,
                  'width': 0.01, 'alpha': 0.5}
    origin = np.zeros((2, len(eigenvectors)))
    ax[i].quiver(*origin, eigenvectors_real[0],
                 eigenvectors_real[1], **parameters)
    ax[i].quiver(*origin,
                 - eigenvectors_real[0],
```

```
                - eigenvectors_real[1],
                **parameters)

    ax[i].set_xlabel("x-axis")
    ax[i].set_ylabel("y-axis")
    ax[i].grid()
    ax[i].set_aspect('equal', adjustable='box')

plt.show()
```

```
Example 1:

eigenvalues:
 [0.61237244+0.35355339j 0.61237244-0.35355339j]
eigenvectors:
 [[0.81649658+0.j         0.81649658-0.j        ]
 [0.40824829-0.40824829j 0.40824829+0.40824829j]]
```

```
Example 2:

eigenvalues:
 [0.8660254+0.5j 0.8660254-0.5j]
eigenvectors:
 [[0.81649658+0.j         0.81649658-0.j        ]
 [0.40824829-0.40824829j 0.40824829+0.40824829j]]

Example 3:

eigenvalues:
 [1.22474487+0.70710678j 1.22474487-0.70710678j]
eigenvectors:
 [[0.81649658+0.j         0.81649658-0.j        ]
 [0.40824829-0.40824829j 0.40824829+0.40824829j]]
```

The vector fields explain why we observed the trajectories of the vector v multiplied by A iteratively before.

The pattern demonstrated here is because we have complex eigenvalues and eigenvectors.

We can plot the complex plane for one of the matrices using `Arrow3D` class retrieved from stackoverflow.

```python
class Arrow3D(FancyArrowPatch):
    def __init__(self, xs, ys, zs, *args, **kwargs):
        super().__init__((0, 0), (0, 0), *args, **kwargs)
```
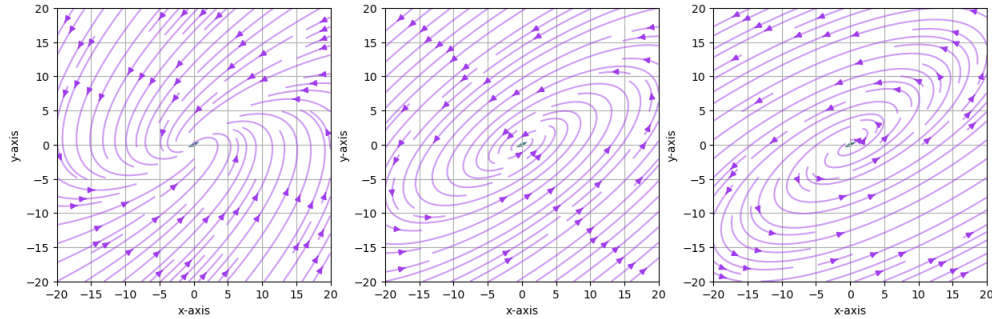
Fig. 16.4: Vector fields of the three matrices

```python
        self._verts3d = xs, ys, zs


    def do_3d_projection(self):
        xs3d, ys3d, zs3d = self._verts3d
        xs, ys, zs = proj3d.proj_transform(xs3d, ys3d, zs3d,
                                           self.axes.M)
        self.set_positions((0.1*xs[0], 0.1*ys[0]),
                           (0.1*xs[1], 0.1*ys[1]))


        return np.min(zs)



eigenvalues, eigenvectors = np.linalg.eig(A)


# Create meshgrid for vector field
x, y = np.meshgrid(np.linspace(-2, 2, 15),
                   np.linspace(-2, 2, 15))


# Calculate vector field (real and imaginary parts)
u_real = A[0][0] * x + A[0][1] * y
v_real = A[1][0] * x + A[1][1] * y
u_imag = np.zeros_like(x)
v_imag = np.zeros_like(y)


# Create 3D figure
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
vlength = np.linalg.norm(eigenvectors)
ax.quiver(x, y, u_imag, u_real-x, v_real-y, v_imag-u_imag,
          colors='b', alpha=0.3, length=.2,
          arrow_length_ratio=0.01)
```

```python
arrow_prop_dict = dict(mutation_scale=5,
                       arrowstyle='-|>', shrinkA=0, shrinkB=0)

# Plot 3D eigenvectors
for c, i in zip(['b', 'g'], [0, 1]):
    a = Arrow3D([0, eigenvectors[0][i].real],
                [0, eigenvectors[1][i].real],
                [0, eigenvectors[1][i].imag],
                color=c, **arrow_prop_dict)
    ax.add_artist(a)

# Set axis labels and title
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('Im')
ax.set_box_aspect(aspect=None, zoom=0.8)

plt.draw()
plt.show()
```
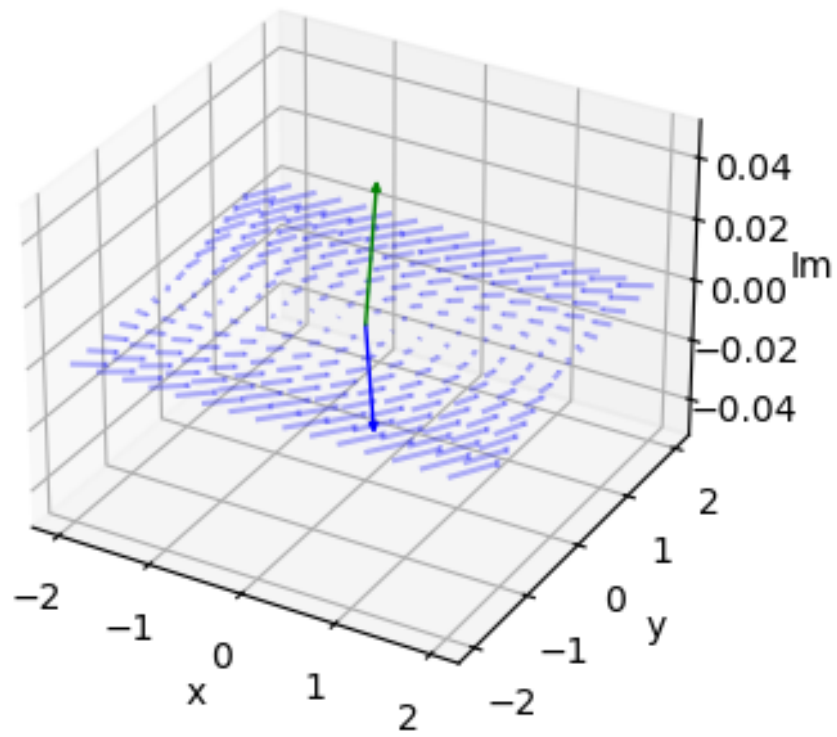
Fig. 16.5: 3D plot of the vector field

COMPUTING SQUARE ROOTS

17.1 Introduction

Chapter 24 of [Russell, 2004] about early Greek mathematics and astronomy contains this fascinating passage:

> The square root of 2, which was the first irrational to be discovered, was known to the early Pythagoreans, and ingenious methods of approximating to its value were discovered. The best was as follows: Form two columns of numbers, which we will call the a's and the b's; each starts with a 1. The next a, at each stage, is formed by adding the last a and the b already obtained; the next b is formed by adding twice the previous a to the previous b. The first 6 pairs so obtained are $(1, 1), (2, 3), (5, 7), (12, 17), (29, 41), (70, 99)$. In each pair, $2a - b$ is 1 or -1. Thus b/a is nearly the square root of two, and at each fresh step it gets nearer. For instance, the reader may satisy himself that the square of $99/70$ is very nearly equal to 2.

This lecture drills down and studies this ancient method for computing square roots by using some of the matrix algebra that we've learned in earlier quantecon lectures.

In particular, this lecture can be viewed as a sequel to *Eigenvalues and Eigenvectors*.

It provides an example of how eigenvectors isolate *invariant subspaces* that help construct and analyze solutions of linear difference equations.

When vector x_t starts in an invariant subspace, iterating the different equation keeps x_{t+j} in that subspace for all $j \geq 1$.

Invariant subspace methods are used throughout applied economic dynamics, for example, in the lecture *Money Financed Government Deficits and Price Levels*.

Our approach here is to illustrate the method with an ancient example, one that ancient Greek mathematicians used to compute square roots of positive integers.

17.2 Perfect squares and irrational numbers

An integer is called a **perfect square** if its square root is also an integer.

An ordered sequence of perfect squares starts with

$$4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, \ldots$$

If an integer is not a perfect square, then its square root is an irrational number –i.e., it cannot be expressed as a ratio of two integers, and its decimal expansion is indefinite.

The ancient Greeks invented an algorithm to compute square roots of integers, including integers that are not perfect squares.

Their method involved

- computing a particular sequence of integers $\{y_t\}_{t=0}^{\infty}$;
- computing $\lim_{t \to \infty} \left(\frac{y_{t+1}}{y_t} \right) = \bar{r}$;
- deducing the desired square root from $\bar{r}$.

In this lecture, we'll describe this method.

We'll also use invariant subspaces to describe variations on this method that are faster.

17.3 Second-order linear difference equations

Before telling how the ancient Greeks computed square roots, we'll provide a quick introduction to second-order linear difference equations.

We'll study the following second-order linear difference equation

$$y_t = a_1 y_{t-1} + a_2 y_{t-2}, \quad t \geq 0 \tag{17.1}$$

where (y_{-1}, y_{-2}) is a pair of given initial conditions.

Equation (17.1) is actually an infinite number of linear equations in the sequence $\{y_t\}_{t=0}^{\infty}$.

There is one equation each for $t = 0, 1, 2, \ldots$.

We could follow an approach taken in the lecture on *present values* and stack all of these equations into a single matrix equation that we would then solve by using matrix inversion.

Note: In the present instance, the matrix equation would multiply a countably infinite dimensional square matrix by a countably infinite dimensional vector. With some qualifications, matrix multiplication and inversion tools apply to such an equation.

But we won't pursue that approach here.

Instead, we'll seek to find a time-invariant function that *solves* our difference equation, meaning that it provides a formula for a $\{y_t\}_{t=0}^{\infty}$ sequence that satisfies equation (17.1) for each $t \geq 0$.

We seek an expression for $y_t, t \geq 0$ as functions of the initial conditions (y_{-1}, y_{-2}):

$$y_t = g((y_{-1}, y_{-2}); t), \quad t \geq 0. \tag{17.2}$$

We call such a function g a *solution* of the difference equation (17.1).

One way to discover a solution is to use a guess and verify method.

We shall begin by considering a special initial pair of initial conditions that satisfy

$$y_{-1} = \delta y_{-2} \tag{17.3}$$

where δ is a scalar to be determined.

For initial condition that satisfy (17.3) equation (17.1) impllies that

$$y_0 = \left(a_1 + \frac{a_2}{\delta}\right) y_{-1}. \tag{17.4}$$

We want

$$\left(a_1 + \frac{a_2}{\delta}\right) = \delta \tag{17.5}$$

which we can rewrite as the *characteristic equation*

$$\delta^2 - a_1 \delta - a_2 = 0. \tag{17.6}$$

Applying the quadratic formula to solve for the roots of (17.6) we find that

$$\delta = \frac{a_1 \pm \sqrt{a_1^2 + 4a_2}}{2}. \tag{17.7}$$

For either of the two δ's that satisfy equation (17.7), a solution of difference equation (17.1) is

$$y_t = \delta^t y_0, \forall t \geq 0 \tag{17.8}$$

provided that we set

$$y_0 = \delta y_{-1}.$$

The *general* solution of difference equation (17.1) takes the form

$$y_t = \eta_1 \delta_1^t + \eta_2 \delta_2^t \tag{17.9}$$

where δ_1, δ_2 are the two solutions (17.7) of the characteristic equation (17.6), and η_1, η_2 are two constants chosen to satisfy

$$\begin{bmatrix} y_{-1} \\ y_{-2} \end{bmatrix} = \begin{bmatrix} \delta_1^{-1} & \delta_2^{-1} \\ \delta_1^{-2} & \delta_2^{-2} \end{bmatrix} \begin{bmatrix} \eta_1 \\ \eta_2 \end{bmatrix} \tag{17.10}$$

17.3. Second-order linear difference equations **331**

or

$$\begin{bmatrix} \eta_1 \\ \eta_2 \end{bmatrix} = \begin{bmatrix} \delta_1^{-1} & \delta_2^{-1} \\ \delta_1^{-2} & \delta_2^{-2} \end{bmatrix}^{-1} \begin{bmatrix} y_{-1} \\ y_{-2} \end{bmatrix} \tag{17.11}$$

Sometimes we are free to choose the initial conditions (y_{-1}, y_{-2}), in which case we use system (17.10) to find the associated (η_1, η_2).

If we choose (y_{-1}, y_{-2}) to set $(\eta_1, \eta_2) = (1, 0)$, then $y_t = \delta_1^t$ for all $t \geq 0$.

If we choose (y_{-1}, y_{-2}) to set $(\eta_1, \eta_2) = (0, 1)$, then $y_t = \delta_2^t$ for all $t \geq 0$.

Soon we'll relate the preceding calculations to components an eigen decomposition of a transition matrix that represents difference equation (17.1) in a very convenient way.

We'll turn to that after we describe how Ancient Greeks figured out how to compute square roots of positive integers that are not perfect squares.

17.4 Algorithm of the Ancient Greeks

Let σ be a positive integer greater than 1.

So $\sigma \in \mathcal{J} \equiv \{2, 3, ...\}$.

We want an algorithm to compute the square root of $\sigma \in \mathcal{J}$.

If $\sqrt{\sigma} \in \mathcal{J}$, σ is said to be a *perfect square*.

If $\sqrt{\sigma} \notin \mathcal{J}$, it turns out that it is irrational.

Ancient Greeks used a recursive algorithm to compute square roots of integers that are not perfect squares.

The algorithm iterates on a second-order linear difference equation in the sequence $\{y_t\}_{t=0}^\infty$:

$$y_t = 2y_{t-1} - (1 - \sigma)y_{t-2}, \quad t \geq 0 \tag{17.12}$$

together with a pair of integers that are initial conditions for y_{-1}, y_{-2}.

First, we'll deploy some techniques for solving the difference equations that are also deployed in Samuelson Multiplier-Accelerator.

The characteristic equation associated with difference equation (17.12) is

$$c(x) \equiv x^2 - 2x + (1 - \sigma) = 0 \tag{17.13}$$

(Notice how this is an instance of equation (17.6) above.)

Factoring the right side of equation (17.13), we obtain

$$c(x) = (x - \lambda_1)(x - \lambda_2) = 0 \tag{17.14}$$

where

$$c(x) = 0$$

for $x = \lambda_1$ or $x = \lambda_2$.

These two special values of x are sometimes called zeros or roots of $c(x)$.

By applying the quadratic formula to solve for the roots the characteristic equation (17.13), we find that

$$\lambda_1 = 1 + \sqrt{\sigma}, \quad \lambda_2 = 1 - \sqrt{\sigma}. \tag{17.15}$$

Formulas (17.15) indicate that λ_1 and λ_2 are each functions of a single variable, namely, $\sqrt{\sigma}$, the object that we along with some Ancient Greeks want to compute.

Ancient Greeks had an indirect way of exploiting this fact to compute square roots of a positive integer.

They did this by starting from particular initial conditions y_{-1}, y_{-2} and iterating on the difference equation (17.12).

Solutions of difference equation (17.12) take the form

$$y_t = \lambda_1^t \eta_1 + \lambda_2^t \eta_2$$

where η_1 and η_2 are chosen to satisfy prescribed initial conditions y_{-1}, y_{-2}:

$$\begin{aligned} \lambda_1^{-1}\eta_1 + \lambda_2^{-1}\eta_2 &= y_{-1} \\ \lambda_1^{-2}\eta_1 + \lambda_2^{-2}\eta_2 &= y_{-2} \end{aligned} \tag{17.16}$$

System (17.16) of simultaneous linear equations will play a big role in the remainder of this lecture.

Since $\lambda_1 = 1 + \sqrt{\sigma} > 1 > \lambda_2 = 1 - \sqrt{\sigma}$, it follows that for *almost all* (but not all) initial conditions

$$\lim_{t \to \infty} \left(\frac{y_{t+1}}{y_t} \right) = 1 + \sqrt{\sigma}.$$

Thus,

$$\sqrt{\sigma} = \lim_{t \to \infty} \left(\frac{y_{t+1}}{y_t} \right) - 1.$$

However, notice that if $\eta_1 = 0$, then

$$\lim_{t \to \infty} \left(\frac{y_{t+1}}{y_t} \right) = 1 - \sqrt{\sigma}$$

so that

$$\sqrt{\sigma} = 1 - \lim_{t \to \infty} \left(\frac{y_{t+1}}{y_t} \right).$$

Actually, if $\eta_1 = 0$, it follows that

$$\sqrt{\sigma} = 1 - \left(\frac{y_{t+1}}{y_t} \right) \quad \forall t \geq 0,$$

so that convergence is immediate and there is no need to take limits.

Symmetrically, if $\eta_2 = 0$, it follows that

$$\sqrt{\sigma} = \left(\frac{y_{t+1}}{y_t} \right) - 1 \quad \forall t \geq 0$$

so again, convergence is immediate, and we have no need to compute a limit.

System (17.16) of simultaneous linear equations can be used in various ways.

- we can take y_{-1}, y_{-2} as given initial conditions and solve for η_1, η_2;

- we can instead take η_1, η_2 as given and solve for initial conditions y_{-1}, y_{-2}.

Notice how we used the second approach above when we set η_1, η_2 either to $(0, 1)$, for example, or $(1, 0)$, for example.

In taking this second approach, we constructed an *invariant subspace* of $\mathbf{R}^2$.

Here is what is going on.

For $t \geq 0$ and for most pairs of initial conditions $(y_{-1}, y_{-2}) \in \mathbf{R}^2$ for equation (17.12), y_t can be expressed as a linear combination of y_{t-1} and y_{t-2}.

But for some special initial conditions $(y_{-1}, y_{-2}) \in \mathbf{R}^2$, y_t can be expressed as a linear function of y_{t-1} only.

These special initial conditions require that y_{-1} be a linear function of y_{-2}.

We'll study these special initial conditions soon.

But first let's write some Python code to iterate on equation (17.12) starting from an arbitrary $(y_{-1}, y_{-2}) \in \mathbf{R}^2$.

17.5 Implementation

We now implement the above algorithm to compute the square root of σ.

In this lecture, we use the following import:

```python
import numpy as np
import matplotlib.pyplot as plt
```

```python
def solve_λs(coefs):
    # Calculate the roots using numpy.roots
    λs = np.roots(coefs)

    # Sort the roots for consistency
    return sorted(λs, reverse=True)


def solve_η(λ_1, λ_2, y_neg1, y_neg2):
    # Solve the system of linear equation
```

```python
    A = np.array([
        [1/λ_1, 1/λ_2],
        [1/(λ_1**2), 1/(λ_2**2)]
    ])
    b = np.array((y_neg1, y_neg2))
    ηs = np.linalg.solve(A, b)

    return ηs


def solve_sqrt(σ, coefs, y_neg1, y_neg2, t_max=100):
    # Ensure σ is greater than 1
    if σ <= 1:
        raise ValueError("σ must be greater than 1")

    # Characteristic roots
    λ_1, λ_2 = solve_λs(coefs)

    # Solve for η_1 and η_2
    η_1, η_2 = solve_η(λ_1, λ_2, y_neg1, y_neg2)

    # Compute the sequence up to t_max
    t = np.arange(t_max + 1)
    y = (λ_1 ** t) * η_1 + (λ_2 ** t) * η_2

    # Compute the ratio y_{t+1} / y_t for large t
    sqrt_σ_estimate = (y[-1] / y[-2]) - 1

    return sqrt_σ_estimate


# Use σ = 2 as an example
σ = 2

# Encode characteristic equation
coefs = (1, -2, (1 - σ))

# Solve for the square root of σ
sqrt_σ = solve_sqrt(σ, coefs, y_neg1=2, y_neg2=1)

# Calculate the deviation
dev = abs(sqrt_σ-np.sqrt(σ))
print(f"sqrt({σ}) is approximately {sqrt_σ:.5f} (error: {dev:.5f})")
```

```
    sqrt(2) is approximately 1.41421 (error: 0.00000)
```

Now we consider cases where $(\eta_1, \eta_2) = (0, 1)$ and $(\eta_1, \eta_2) = (1, 0)$

```python
# Compute λ_1, λ_2
λ_1, λ_2 = solve_λs(coefs)
print(f'Roots for the characteristic equation are ({λ_1:.5f}, {λ_2:.5f}))')
```

```
    Roots for the characteristic equation are (2.41421, -0.41421))
```

```python
# Case 1: η_1, η_2 = (0, 1)
ηs = (0, 1)

# Compute y_{t} and y_{t-1} with t >= 0
y = lambda t, ηs: (λ_1 ** t) * ηs[0] + (λ_2 ** t) * ηs[1]
sqrt_σ = 1 - y(1, ηs) / y(0, ηs)

print(f"For η_1, η_2 = (0, 1), sqrt_σ = {sqrt_σ:.5f}")
```

```
    For η_1, η_2 = (0, 1), sqrt_σ = 1.41421
```

```python
# Case 2: η_1, η_2 = (1, 0)
ηs = (1, 0)
sqrt_σ = y(1, ηs) / y(0, ηs) - 1

print(f"For η_1, η_2 = (1, 0), sqrt_σ = {sqrt_σ:.5f}")
```

```
    For η_1, η_2 = (1, 0), sqrt_σ = 1.41421
```

We find that convergence is immediate.

Next, we'll represent the preceding analysis by first vectorizing our second-order difference equation (17.12) and then using eigendecompositions of an associated state transition matrix.

17.6 Vectorizing the difference equation

Represent (17.12) with the first-order matrix difference equation

$$\begin{bmatrix} y_{t+1} \\ y_t \end{bmatrix} = \begin{bmatrix} 2 & -(1-\sigma) \\ 1 & 0 \end{bmatrix} \begin{bmatrix} y_t \\ y_{t-1} \end{bmatrix}$$

or

$$x_{t+1} = M x_t$$

where

$$M = \begin{bmatrix} 2 & -(1-\sigma) \\ 1 & 0 \end{bmatrix}, \quad x_t = \begin{bmatrix} y_t \\ y_{t-1} \end{bmatrix}$$

Construct an eigendecomposition of M:

$$M = V \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} V^{-1} \tag{17.17}$$

where columns of V are eigenvectors corresponding to eigenvalues λ_1 and λ_2.

The eigenvalues can be ordered so that $\lambda_1 > 1 > \lambda_2$.

Write equation (17.12) as

$$x_{t+1} = V \Lambda V^{-1} x_t$$

Now we implement the algorithm above.

First we write a function that iterates M

```python
def iterate_M(x_0, M, num_steps, dtype=np.float64):

    # Eigendecomposition of M
    Λ, V = np.linalg.eig(M)
    V_inv = np.linalg.inv(V)

    # Initialize the array to store results
    xs = np.zeros((x_0.shape[0],
                   num_steps + 1))

    # Perform the iterations
    xs[:, 0] = x_0
    for t in range(num_steps):
        xs[:, t + 1] = M @ xs[:, t]
```

```
    return xs, Λ, V, V_inv

# Define the state transition matrix M
M = np.array([
        [2, -(1 - σ)],
        [1, 0]])

# Initial condition vector x_0
x_0 = np.array([2, 2])

# Perform the iteration
xs, Λ, V, V_inv = iterate_M(x_0, M, num_steps=100)

print(f"eigenvalues:\n{Λ}")
print(f"eigenvectors:\n{V}")
print(f"inverse eigenvectors:\n{V_inv}")
```

```
eigenvalues:
[ 2.41421356 -0.41421356]
eigenvectors:
[[ 0.92387953 -0.38268343]
 [ 0.38268343  0.92387953]]
inverse eigenvectors:
[[ 0.92387953  0.38268343]
 [-0.38268343  0.92387953]]
```

Let's compare the eigenvalues to the roots (17.15) of equation (17.13) that we computed above.

```
roots = solve_λs((1, -2, (1 - σ)))
print(f"roots: {np.round(roots, 8)}")
```

```
roots: [ 2.41421356 -0.41421356]
```

Hence we confirmed (17.17).

Information about the square root we are after is also contained in the two eigenvectors.

Indeed, each eigenvector is just a two-dimensional subspace of $\mathbb{R}^3$ pinned down by dynamics of the form

$$y_t = \lambda_i y_{t-1}, \quad i = 1, 2 \tag{17.18}$$

that we encountered above in equation (17.8) above.

In equation (17.18), the ith λ_i equals the $V_{i,1}/V_{i,2}$.

The following graph verifies this for our example.
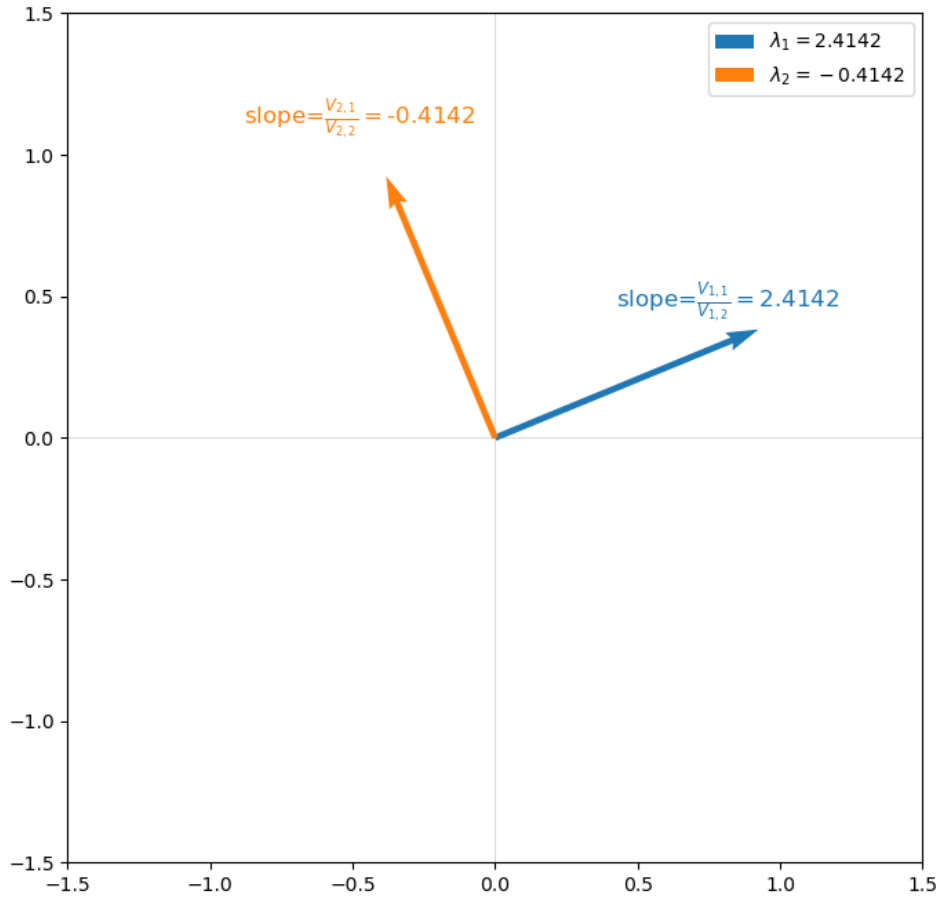
17.7 Invariant subspace approach

The preceding calculation indicates that we can use the eigenvectors V to construct 2-dimensional *invariant subspaces*.

We'll pursue that possibility now.

Define the transformed variables

$$x_t^* = V^{-1} x_t$$

Evidently, we can recover x_t from x_t^*:

$$x_t = V x_t^*$$

The following notations and equations will help us.

Let

$$V = \begin{bmatrix} V_{1,1} & V_{1,2} \\ V_{2,1} & V_{2,2} \end{bmatrix}, \quad V^{-1} = \begin{bmatrix} V^{1,1} & V^{1,2} \\ V^{2,1} & V^{2,2} \end{bmatrix}$$

Notice that it follows from

$$\begin{bmatrix} V^{1,1} & V^{1,2} \\ V^{2,1} & V^{2,2} \end{bmatrix} \begin{bmatrix} V_{1,1} & V_{1,2} \\ V_{2,1} & V_{2,2} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

that

$$V^{2,1}V_{1,1} + V^{2,2}V_{2,1} = 0$$

and

$$V^{1,1}V_{1,2} + V^{1,2}V_{2,2} = 0.$$

These equations will be very useful soon.

Notice that

$$\begin{bmatrix} x^*_{1,t+1} \\ x^*_{2,t+1} \end{bmatrix} = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \begin{bmatrix} x^*_{1,t} \\ x^*_{2,t} \end{bmatrix}$$

To deactivate λ_1 we want to set

$$x^*_{1,0} = 0.$$

This can be achieved by setting

$$x_{2,0} = -(V^{1,2})^{-1}V^{1,1}x_{1,0} = V_{2,2}V_{1,2}^{-1}x_{1,0}. \qquad (17.19)$$

To deactivate λ_2, we want to set

$$x^*_{2,0} = 0$$

This can be achieved by setting

$$x_{2,0} = -(V^{2,2})^{-1}V^{2,1}x_{1,0} = V_{2,1}V_{1,1}^{-1}x_{1,0}. \qquad (17.20)$$

Let's verify (17.19) and (17.20) below

To deactivate λ_1 we use (17.19)

```
xd_1 = np.array((x_0[0],
                V[1,1]/V[0,1] * x_0[0]),
               dtype=np.float64)

# Compute x_{1,0}^*
np.round(V_inv @ xd_1, 8)
```

```
array([-0.        , -5.22625186])
```

We find $x_{1,0}^* = 0$.

Now we deactivate λ_2 using (17.20)

```python
xd_2 = np.array((x_0[0],
                V[1,0]/V[0,0] * x_0[0]),
                dtype=np.float64)

# Compute x_{2,0}^*
np.round(V_inv @ xd_2, 8)
```

```
array([2.1647844, 0.        ])
```

We find $x_{2,0}^* = 0$.

```python
# Simulate with muted λ1 λ2.
num_steps = 10
xs_λ1 = iterate_M(xd_1, M, num_steps)[0]
xs_λ2 = iterate_M(xd_2, M, num_steps)[0]

# Compute ratios y_t / y_{t-1}
ratios_λ1 = xs_λ1[1, 1:] / xs_λ1[1, :-1]
ratios_λ2 = xs_λ2[1, 1:] / xs_λ2[1, :-1]
```

The following graph shows the ratios y_t/y_{t-1} for the two cases.

We find that the ratios converge to λ_2 in the first case and λ_1 in the second case.
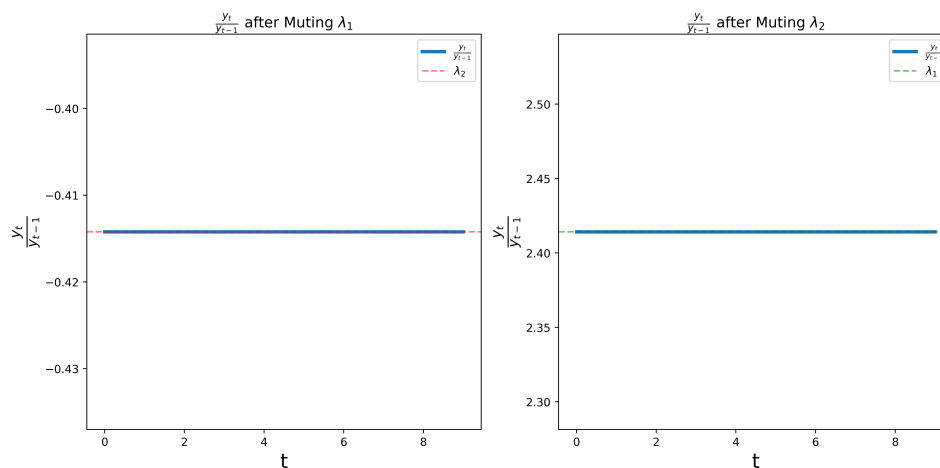
17.8 Concluding remarks

This lecture sets the stage for many other applications of the *invariant subspace* methods.

All of these exploit very similar equations based on eigen decompositions.

We shall encounter equations very similar to (17.19) and (17.20) in *Money Financed Government Deficits and Price Levels* and in many other places in dynamic economic theory.

17.9 Exercise

Exercise 17.9.1

Please use matrix algebra to formulate the method described by Bertrand Russell at the beginning of this lecture.

1. Define a state vector $x_t = \begin{bmatrix} a_t \\ b_t \end{bmatrix}$.

2. Formulate a first-order vector difference equation for x_t of the form $x_{t+1} = Ax_t$ and compute the matrix A.

3. Use the system $x_{t+1} = Ax_t$ to replicate the sequence of a_t's and b_t's described by Bertrand Russell.

4. Compute the eigenvectors and eigenvalues of A and compare them to corresponding objects computed in the text of this lecture.

Solution to Exercise 17.9.1

Here is one soluition.

According to the quote, we can formulate

$$
\begin{aligned}
a_{t+1} &= a_t + b_t \\
b_{t+1} &= 2a_t + b_t
\end{aligned}
\tag{17.21}
$$

with $x_0 = \begin{bmatrix} a_0 \\ b_0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

By (17.21), we can write matrix A as

$$
A = \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix}
$$

Then $x_{t+1} = Ax_t$ for $t \in \{0, \dots, 5\}$

```python
# Define the matrix A
A = np.array([[1, 1],
              [2, 1]])

# Initial vector x_0
x_0 = np.array([1, 1])

# Number of iterations
n = 6

# Generate the sequence
xs = np.array([x_0])
x_t = x_0
for _ in range(1, n):
    x_t = A @ x_t
    xs = np.vstack([xs, x_t])

# Print the sequence
for i, (a_t, b_t) in enumerate(xs):
    print(f"Iter {i}: a_t = {a_t}, b_t = {b_t}")

# Compute eigenvalues and eigenvectors of A
eigenvalues, eigenvectors = np.linalg.eig(A)

print(f'\nEigenvalues:\n{eigenvalues}')
print(f'\nEigenvectors:\n{eigenvectors}')
```

```
Iter 0: a_t = 1, b_t = 1
Iter 1: a_t = 2, b_t = 3
Iter 2: a_t = 5, b_t = 7
Iter 3: a_t = 12, b_t = 17
Iter 4: a_t = 29, b_t = 41
Iter 5: a_t = 70, b_t = 99

Eigenvalues:
[ 2.41421356 -0.41421356]

Eigenvectors:
[[ 0.57735027 -0.57735027]
 [ 0.81649658  0.81649658]]
```

Part VI

Probability and Distributions

EIGHTEEN

DISTRIBUTIONS AND PROBABILITIES

18.1 Outline

In this lecture we give a quick introduction to data and probability distributions using Python.

```
!pip install --upgrade yfinance
```

```python
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import yfinance as yf
import scipy.stats
import seaborn as sns
```

18.2 Common distributions

In this section we recall the definitions of some well-known distributions and explore how to manipulate them with SciPy.

18.2.1 Discrete distributions

Let's start with discrete distributions.

A discrete distribution is defined by a set of numbers $S = \{x_1, \dots, x_n\}$ and a **probability mass function** (PMF) on S, which is a function p from S to $[0, 1]$ with the property

$$\sum_{i=1}^{n} p(x_i) = 1$$

We say that a random variable X **has distribution** p if X takes value x_i with probability $p(x_i)$.

That is,

$$\mathbb{P}\{X = x_i\} = p(x_i) \quad \text{for } i = 1, \dots, n$$

The **mean** or **expected value** of a random variable X with distribution p is

$$\mathbb{E}[X] = \sum_{i=1}^{n} x_i p(x_i)$$

Expectation is also called the *first moment* of the distribution.

We also refer to this number as the mean of the distribution (represented by) p.

The **variance** of X is defined as

$$\mathbb{V}[X] = \sum_{i=1}^{n} (x_i - \mathbb{E}[X])^2 p(x_i)$$

Variance is also called the *second central moment* of the distribution.

The **cumulative distribution function** (CDF) of X is defined by

$$F(x) = \mathbb{P}\{X \leq x\} = \sum_{i=1}^{n} \mathbb{1}\{x_i \leq x\} p(x_i)$$

Here $\mathbb{1}\{\text{statement}\} = 1$ if "statement" is true and zero otherwise.

Hence the second term takes all $x_i \leq x$ and sums their probabilities.

Uniform distribution

One simple example is the **uniform distribution**, where $p(x_i) = 1/n$ for all i.

We can import the uniform distribution on $S = \{1, \dots, n\}$ from SciPy like so:

```
n = 10
u = scipy.stats.randint(1, n+1)
```

Here's the mean and variance:

```
u.mean(), u.var()
```

```
(5.5, 8.25)
```

The formula for the mean is $(n + 1)/2$, and the formula for the variance is $(n^2 - 1)/12$.

Now let's evaluate the PMF:

```
u.pmf(1)
```

```
0.1
```

```
u.pmf(2)
```

```
0.1
```

Here's a plot of the probability mass function:

```
fig, ax = plt.subplots()
S = np.arange(1, n+1)
ax.plot(S, u.pmf(S), linestyle='', marker='o', alpha=0.8, ms=4)
ax.vlines(S, 0, u.pmf(S), lw=0.2)
ax.set_xticks(S)
ax.set_xlabel('S')
ax.set_ylabel('PMF')
plt.show()
```


Here's a plot of the CDF:

```
fig, ax = plt.subplots()
```
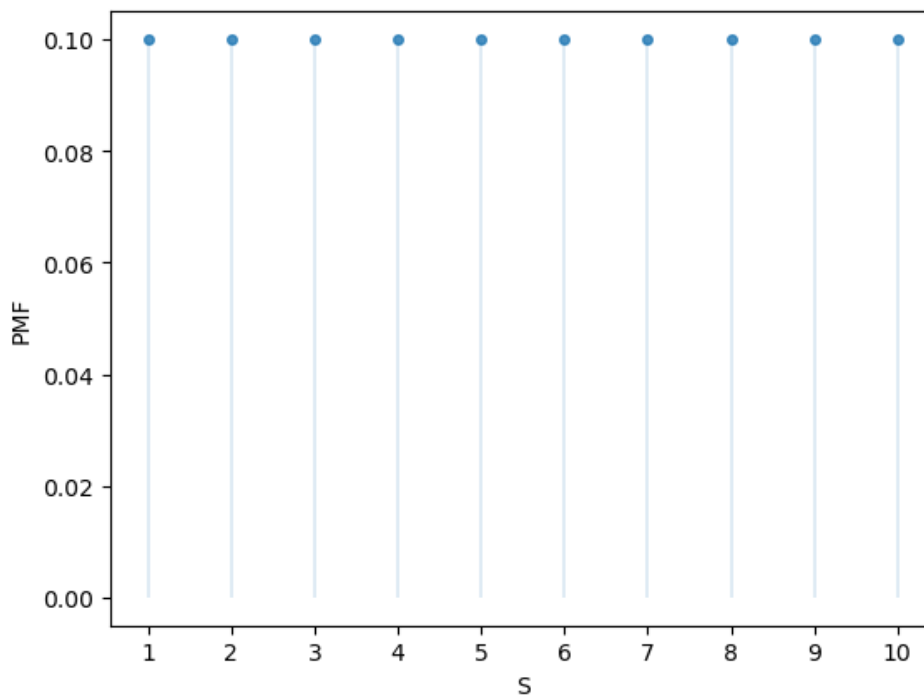
```
S = np.arange(1, n+1)
ax.step(S, u.cdf(S))
ax.vlines(S, 0, u.cdf(S), lw=0.2)
ax.set_xticks(S)
ax.set_xlabel('S')
ax.set_ylabel('CDF')
plt.show()
```
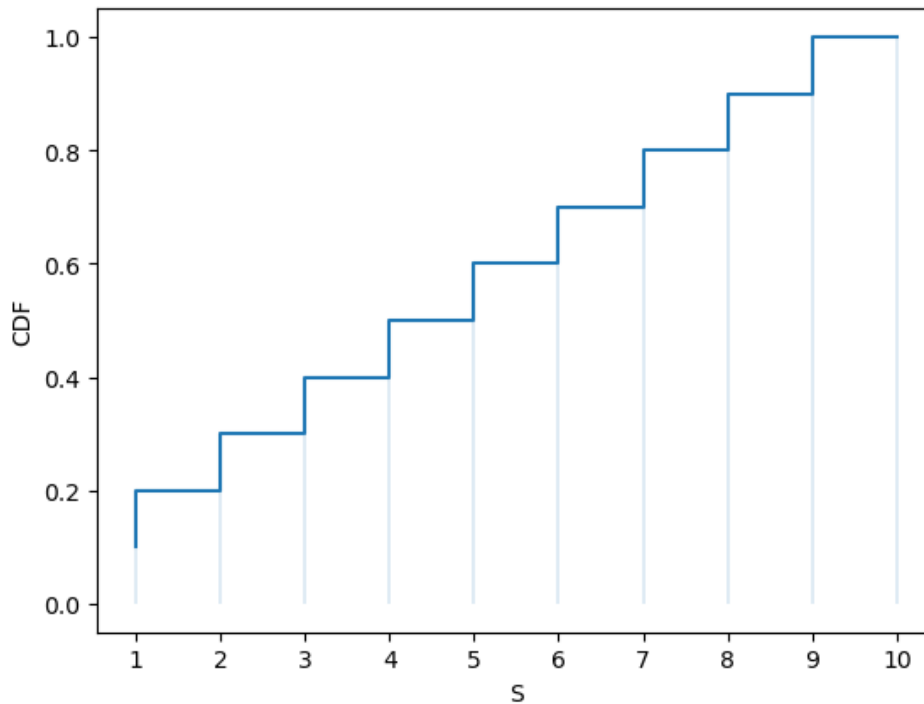

The CDF jumps up by $p(x_i)$ at x_i.

Exercise 18.2.1

Calculate the mean and variance for this parameterization (i.e., $n = 10$) directly from the PMF, using the expressions given above.

Check that your answers agree with `u.mean()` and `u.var()`.

Bernoulli distribution

Another useful distribution is the Bernoulli distribution on $S = \{0, 1\}$, which has PMF:

$$p(i) = \theta^i(1-\theta)^{1-i} \qquad (i = 0, 1)$$

Here $\theta \in [0, 1]$ is a parameter.

We can think of this distribution as modeling probabilities for a random trial with success probability θ.

- $p(1) = \theta$ means that the trial succeeds (takes value 1) with probability θ

- $p(0) = 1 - \theta$ means that the trial fails (takes value 0) with probability $1 - \theta$

The formula for the mean is θ, and the formula for the variance is $\theta(1 - \theta)$.

We can import the Bernoulli distribution on $S = \{0, 1\}$ from SciPy like so:

```
θ = 0.4
u = scipy.stats.bernoulli(θ)
```

Here's the mean and variance at $\theta = 0.4$

```
u.mean(), u.var()
```

```
(0.4, 0.24)
```

We can evaluate the PMF as follows

```
u.pmf(0), u.pmf(1)
```

```
(0.5999999999999998, 0.4)
```

Binomial distribution

Another useful (and more interesting) distribution is the **binomial distribution** on $S = \{0, ..., n\}$, which has PMF:

$$p(i) = \binom{n}{i} \theta^i (1-\theta)^{n-i}$$

Again, $\theta \in [0, 1]$ is a parameter.

The interpretation of $p(i)$ is: the probability of i successes in n independent trials with success probability θ.

For example, if $\theta = 0.5$, then $p(i)$ is the probability of i heads in n flips of a fair coin.

The formula for the mean is $n\theta$ and the formula for the variance is $n\theta(1 - \theta)$.

Let's investigate an example

```
n = 10
θ = 0.5
u = scipy.stats.binom(n, θ)
```

According to our formulas, the mean and variance are

```
n * θ,  n *  θ * (1 - θ)
```

```
(5.0, 2.5)
```

Let's see if SciPy gives us the same results:

```
u.mean(), u.var()
```

```
(5.0, 2.5)
```

Here's the PMF:

```
u.pmf(1)
```

```
0.009765625000000009
```

```
fig, ax = plt.subplots()
S = np.arange(1, n+1)
ax.plot(S, u.pmf(S), linestyle='', marker='o', alpha=0.8, ms=4)
ax.vlines(S, 0, u.pmf(S), lw=0.2)
ax.set_xticks(S)
ax.set_xlabel('S')
ax.set_ylabel('PMF')
plt.show()
```
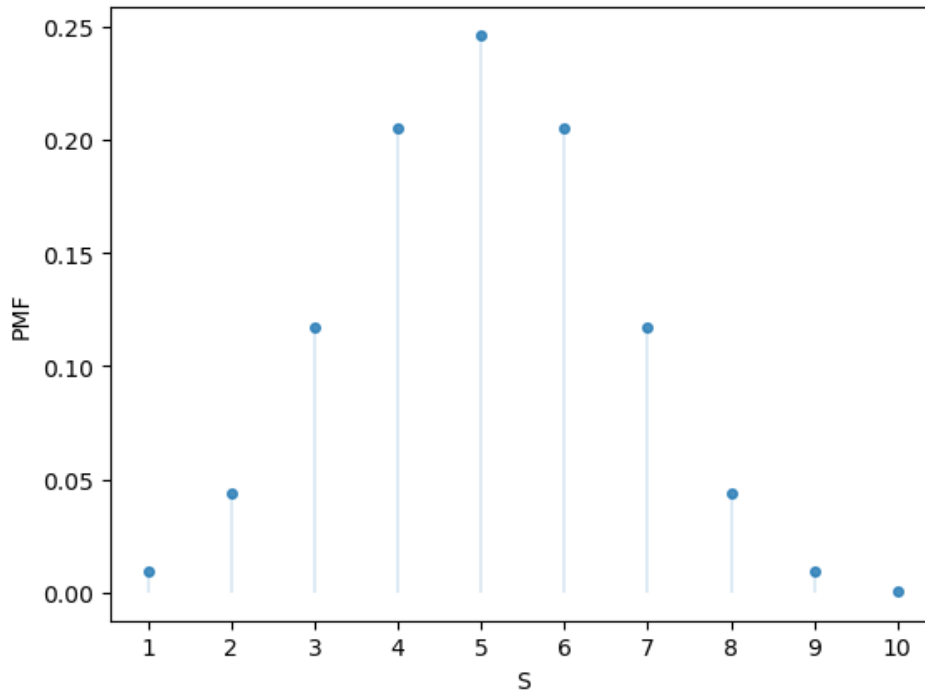
Here's the CDF:

```
fig, ax = plt.subplots()
S = np.arange(1, n+1)
ax.step(S, u.cdf(S))
ax.vlines(S, 0, u.cdf(S), lw=0.2)
ax.set_xticks(S)
ax.set_xlabel('S')
ax.set_ylabel('CDF')
plt.show()
```

Exercise 18.2.2

Using `u.pmf`, check that our definition of the CDF given above calculates the same function as `u.cdf`.

Solution to Exercise 18.2.2

Here is one solution:

```
fig, ax = plt.subplots()
S = np.arange(1, n+1)
u_sum = np.cumsum(u.pmf(S))
ax.step(S, u_sum)
ax.vlines(S, 0, u_sum, lw=0.2)
ax.set_xticks(S)
ax.set_xlabel('S')
ax.set_ylabel('CDF')
plt.show()
```
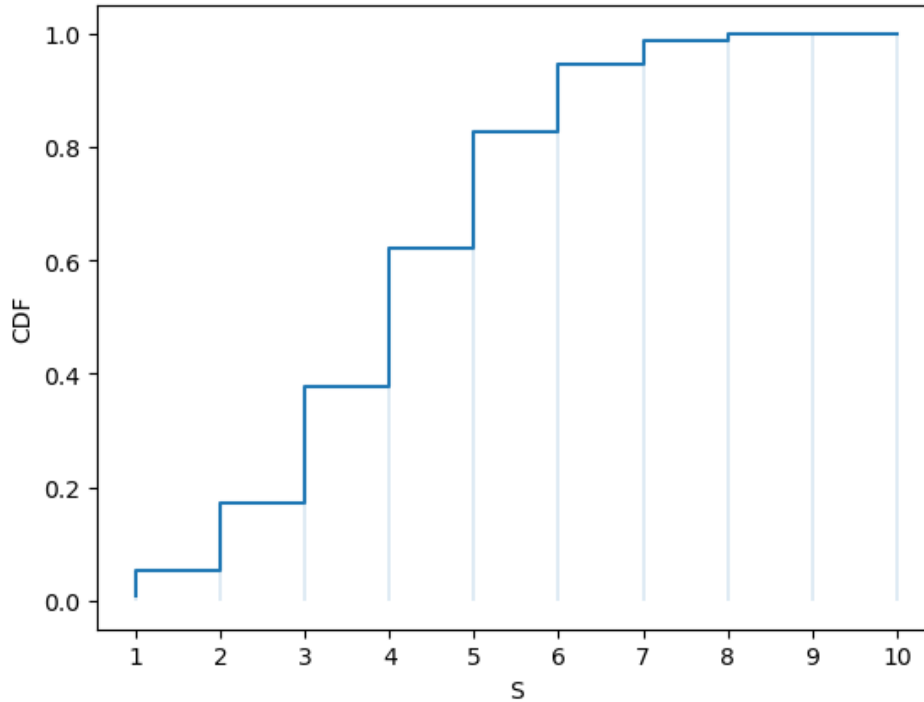
We can see that the output graph is the same as the one above.

Geometric distribution

The geometric distribution has infinite support $S = \{0, 1, 2, ...\}$ and its PMF is given by

$$p(i) = (1 - \theta)^i \theta$$

where $\theta \in [0, 1]$ is a parameter

(A discrete distribution has infinite support if the set of points to which it assigns positive probability is infinite.)

To understand the distribution, think of repeated independent random trials, each with success probability θ.

The interpretation of $p(i)$ is: the probability there are i failures before the first success occurs.

It can be shown that the mean of the distribution is $1/\theta$ and the variance is $(1 - \theta)/\theta$.

Here's an example.

```
θ = 0.1
u = scipy.stats.geom(θ)
u.mean(), u.var()
```

```
(10.0, 90.0)
```

Here's part of the PMF:

```
fig, ax = plt.subplots()
n = 20
S = np.arange(n)
ax.plot(S, u.pmf(S), linestyle='', marker='o', alpha=0.8, ms=4)
ax.vlines(S, 0, u.pmf(S), lw=0.2)
ax.set_xticks(S)
ax.set_xlabel('S')
ax.set_ylabel('PMF')
plt.show()
```
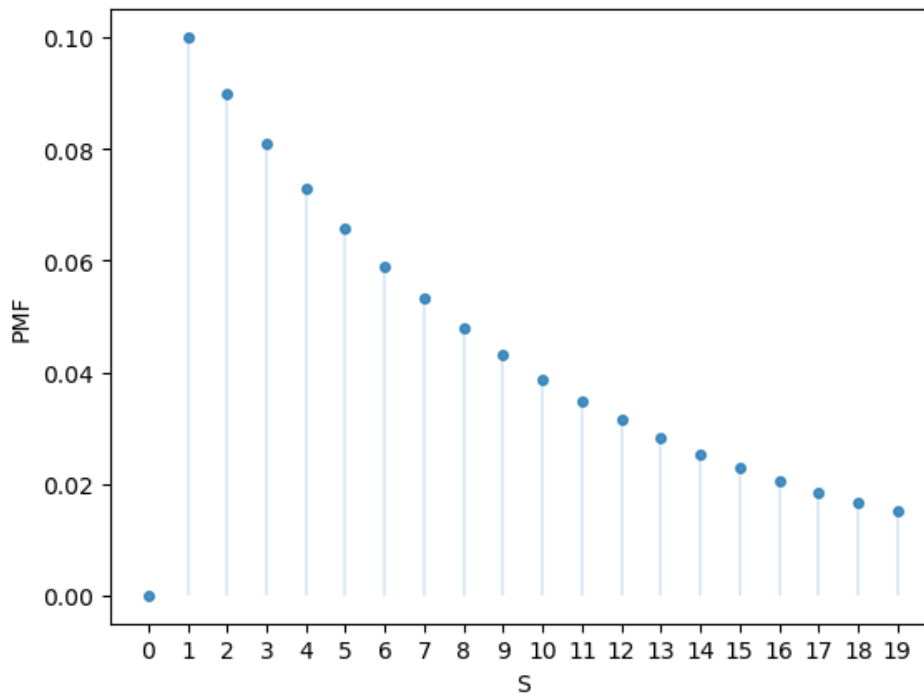

Poisson distribution

The Poisson distribution on $S = \{0, 1, ...\}$ with parameter $\lambda > 0$ has PMF

$$p(i) = \frac{\lambda^i}{i!} e^{-\lambda}$$

The interpretation of $p(i)$ is: the probability of i events in a fixed time interval, where the events occur independently at a constant rate λ.

It can be shown that the mean is λ and the variance is also λ.

Here's an example.

```
λ = 2
u = scipy.stats.poisson(λ)
u.mean(), u.var()
```

```
(2.0, 2.0)
```

Here's the PMF:

```
u.pmf(1)
```

```
0.2706705664732254
```
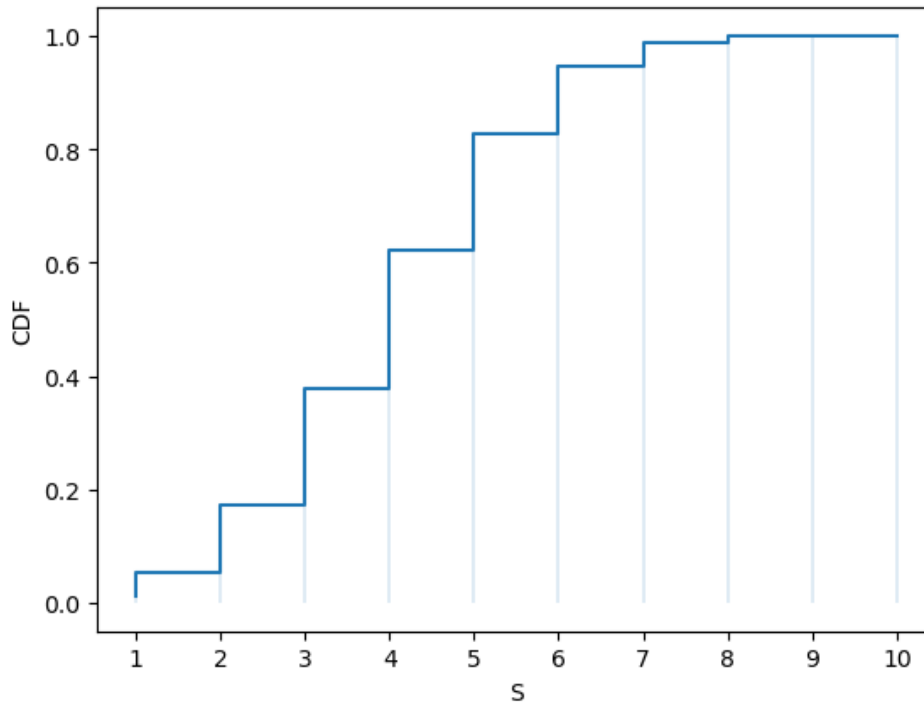
```
fig, ax = plt.subplots()
S = np.arange(1, n+1)
ax.plot(S, u.pmf(S), linestyle='', marker='o', alpha=0.8, ms=4)
ax.vlines(S, 0, u.pmf(S), lw=0.2)
ax.set_xticks(S)
ax.set_xlabel('S')
ax.set_ylabel('PMF')
plt.show()
```
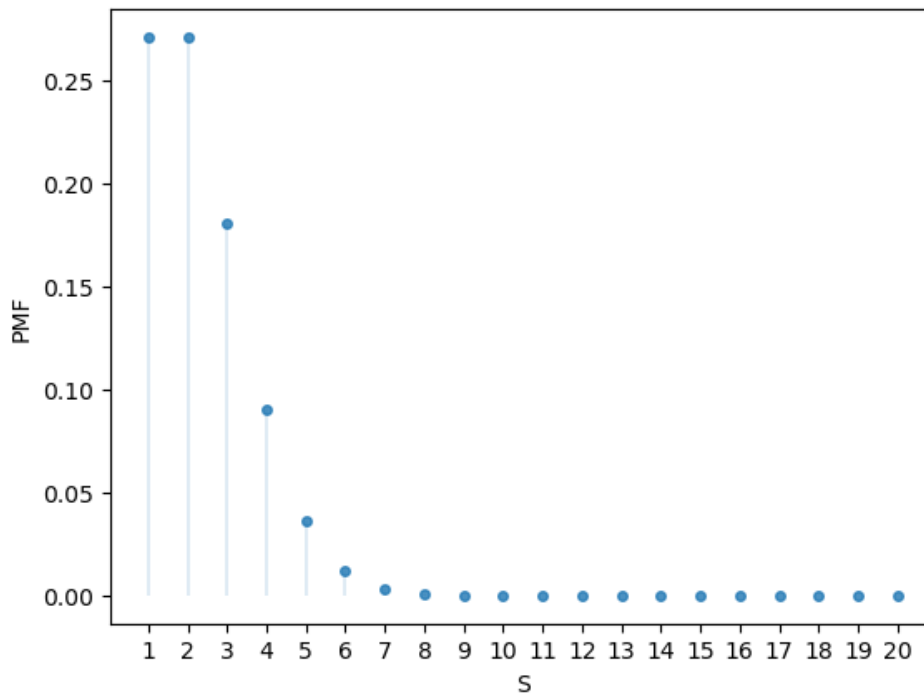
18.2.2 Continuous distributions

A continuous distribution is represented by a **probability density function**, which is a function p over $\mathbb{R}$ (the set of all real numbers) such that $p(x) \geq 0$ for all x and

$$\int_{-\infty}^{\infty} p(x)dx = 1$$

We say that random variable X has distribution p if

$$\mathbb{P}\{a < X < b\} = \int_{a}^{b} p(x)dx$$

for all $a \leq b$.

The definition of the mean and variance of a random variable X with distribution p are the same as the discrete case, after replacing the sum with an integral.

For example, the mean of X is

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} xp(x)dx$$

The **cumulative distribution function** (CDF) of X is defined by

$$F(x) = \mathbb{P}\{X \leq x\} = \int_{-\infty}^{x} p(x)dx$$

Normal distribution

Perhaps the most famous distribution is the **normal distribution**, which has density

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

This distribution has two parameters, $\mu \in \mathbb{R}$ and $\sigma \in (0, \infty)$.

Using calculus, it can be shown that, for this distribution, the mean is μ and the variance is σ^2.

We can obtain the moments, PDF and CDF of the normal density via SciPy as follows:

```
μ, σ = 0.0, 1.0
u = scipy.stats.norm(μ, σ)
```

```
u.mean(), u.var()
```

```
(0.0, 1.0)
```

Here's a plot of the density —the famous "bell-shaped curve":

```
μ_vals = [-1, 0, 1]
σ_vals = [0.4, 1, 1.6]
fig, ax = plt.subplots()
x_grid = np.linspace(-4, 4, 200)

for μ, σ in zip(μ_vals, σ_vals):
    u = scipy.stats.norm(μ, σ)
    ax.plot(x_grid, u.pdf(x_grid),
    alpha=0.5, lw=2,
    label=f'$\mu={μ}, \sigma={σ}$')
ax.set_xlabel('x')
ax.set_ylabel('PDF')
plt.legend()
plt.show()
```


Here's a plot of the CDF:

```
fig, ax = plt.subplots()
for μ, σ in zip(μ_vals, σ_vals):
    u = scipy.stats.norm(μ, σ)
    ax.plot(x_grid, u.cdf(x_grid),
    alpha=0.5, lw=2,
    label=f'$\mu={μ}, \sigma={σ}$')
    ax.set_ylim(0, 1)
```
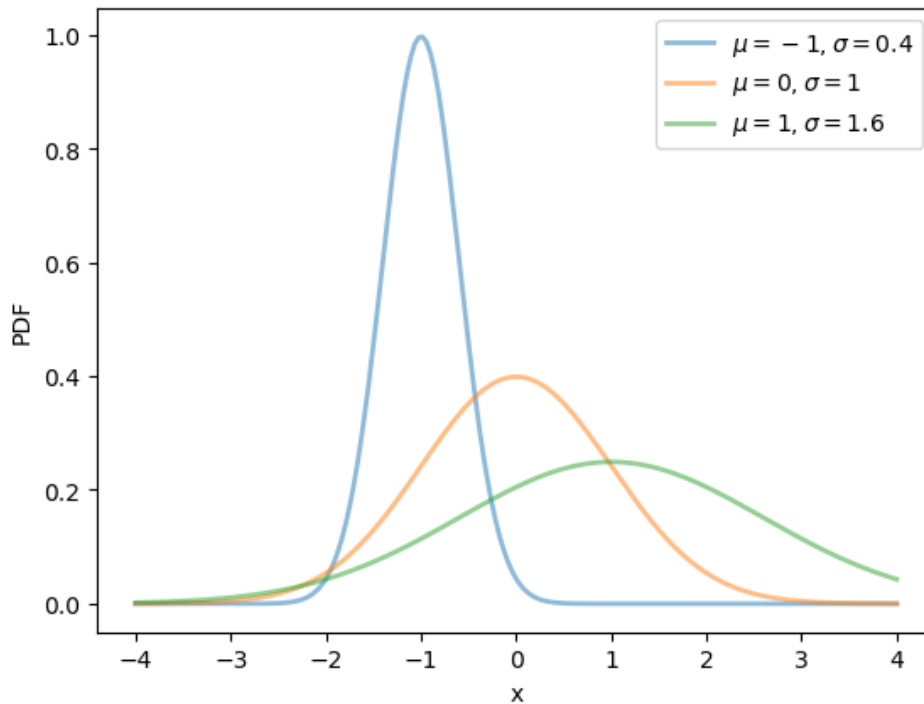
```
ax.set_xlabel('x')
ax.set_ylabel('CDF')
plt.legend()
plt.show()
```
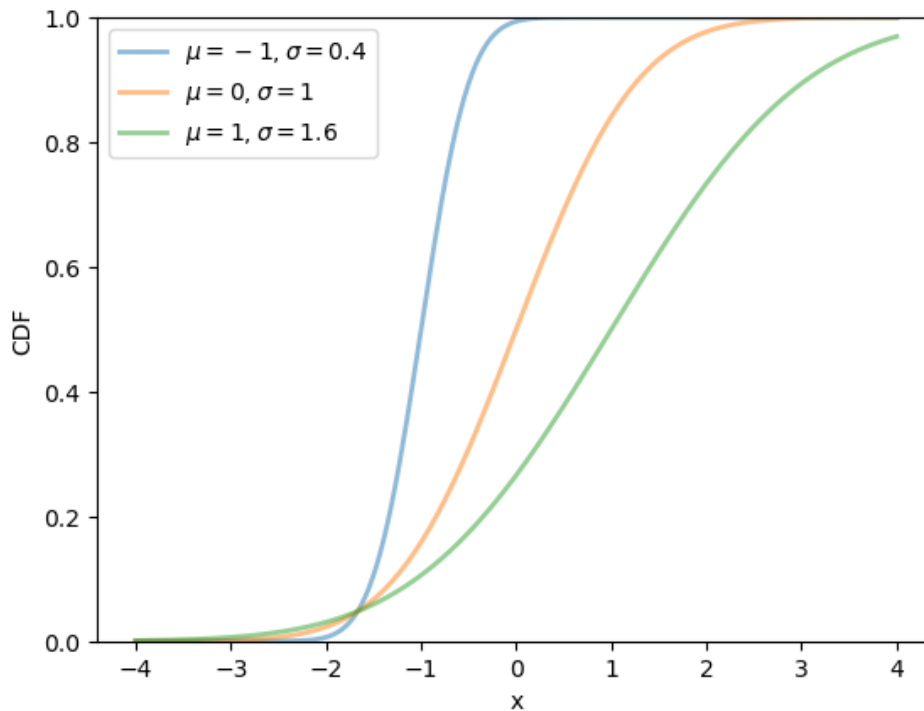

Lognormal distribution

The **lognormal distribution** is a distribution on $(0, \infty)$ with density

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} \exp \left(-\frac{(\log x - \mu)^2}{2\sigma^2} \right)$$

This distribution has two parameters, μ and σ.

It can be shown that, for this distribution, the mean is $\exp \left(\mu + \sigma^2/2 \right)$ and the variance is $\left[\exp \left(\sigma^2 \right) - 1 \right] \exp \left(2\mu + \sigma^2 \right)$.

It can be proved that

- if X is lognormally distributed, then $\log X$ is normally distributed, and

- if X is normally distributed, then $\exp X$ is lognormally distributed.

We can obtain the moments, PDF, and CDF of the lognormal density as follows:

```
μ, σ = 0.0, 1.0
u = scipy.stats.lognorm(s=σ, scale=np.exp(μ))
```

```
u.mean(), u.var()
```

```
(1.6487212707001282, 4.670774270471604)
```

```
μ_vals = [-1, 0, 1]
σ_vals = [0.25, 0.5, 1]
x_grid = np.linspace(0, 3, 200)

fig, ax = plt.subplots()
for μ, σ in zip(μ_vals, σ_vals):
    u = scipy.stats.lognorm(σ, scale=np.exp(μ))
    ax.plot(x_grid, u.pdf(x_grid),
    alpha=0.5, lw=2,
    label=f'$\mu={μ}, \sigma={σ}$')
ax.set_xlabel('x')
ax.set_ylabel('PDF')
plt.legend()
plt.show()
```


18.2. Common distributions
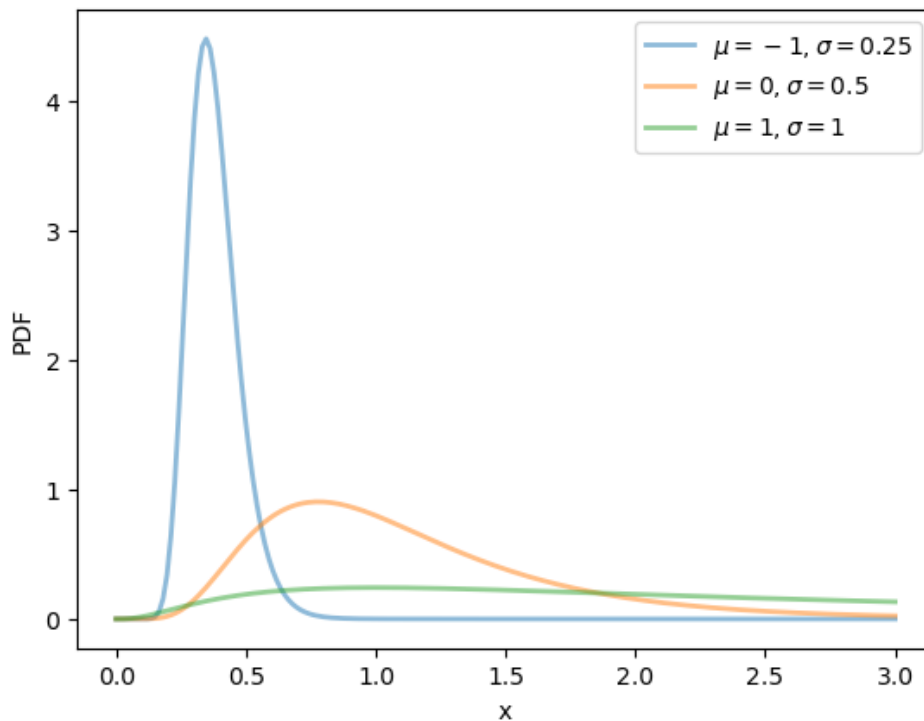
```
fig, ax = plt.subplots()
μ = 1
for σ in σ_vals:
    u = scipy.stats.norm(μ, σ)
    ax.plot(x_grid, u.cdf(x_grid),
    alpha=0.5, lw=2,
    label=f'$\mu={μ}, \sigma={σ}$')
    ax.set_ylim(0, 1)
    ax.set_xlim(0, 3)
ax.set_xlabel('x')
ax.set_ylabel('CDF')
plt.legend()
plt.show()
```
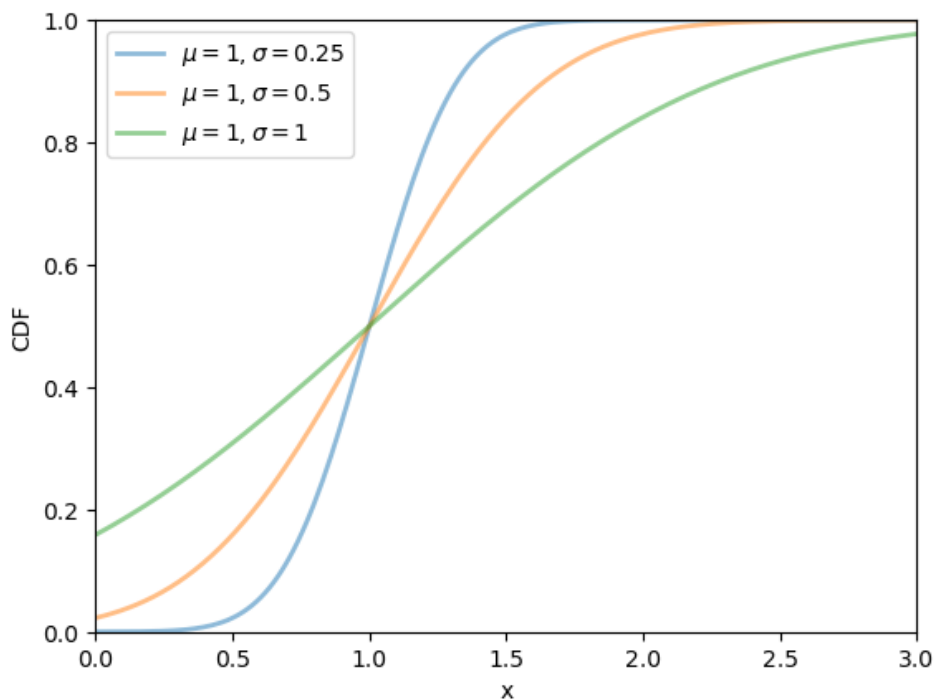

Exponential distribution

The **exponential distribution** is a distribution supported on $(0, \infty)$ with density

$$p(x) = \lambda \exp(-\lambda x) \qquad (x > 0)$$

This distribution has one parameter λ.

The exponential distribution can be thought of as the continuous analog of the geometric distribution.

It can be shown that, for this distribution, the mean is $1/\lambda$ and the variance is $1/\lambda^2$.

We can obtain the moments, PDF, and CDF of the exponential density as follows:

```
λ = 1.0
u = scipy.stats.expon(scale=1/λ)
```

```
u.mean(), u.var()
```

```
(1.0, 1.0)
```

```
fig, ax = plt.subplots()
λ_vals = [0.5, 1, 2]
x_grid = np.linspace(0, 6, 200)

for λ in λ_vals:
    u = scipy.stats.expon(scale=1/λ)
    ax.plot(x_grid, u.pdf(x_grid),
    alpha=0.5, lw=2,
    label=f'$\lambda={λ}$')
ax.set_xlabel('x')
ax.set_ylabel('PDF')
plt.legend()
plt.show()
```
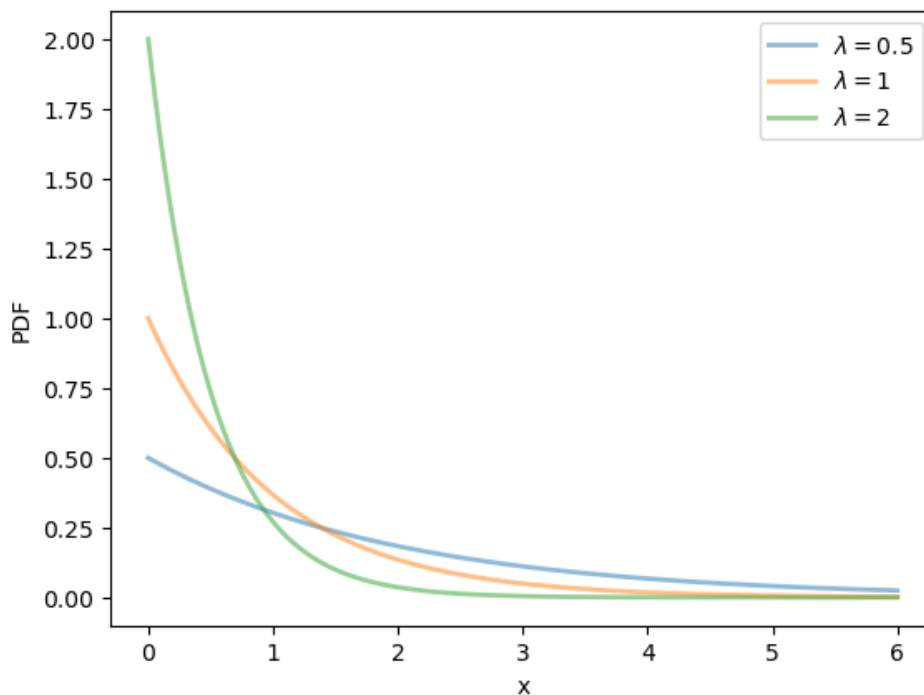
```
fig, ax = plt.subplots()
for λ in λ_vals:
    u = scipy.stats.expon(scale=1/λ)
    ax.plot(x_grid, u.cdf(x_grid),
    alpha=0.5, lw=2,
    label=f'$\lambda={λ}$')
    ax.set_ylim(0, 1)
ax.set_xlabel('x')
ax.set_ylabel('CDF')
plt.legend()
plt.show()
```
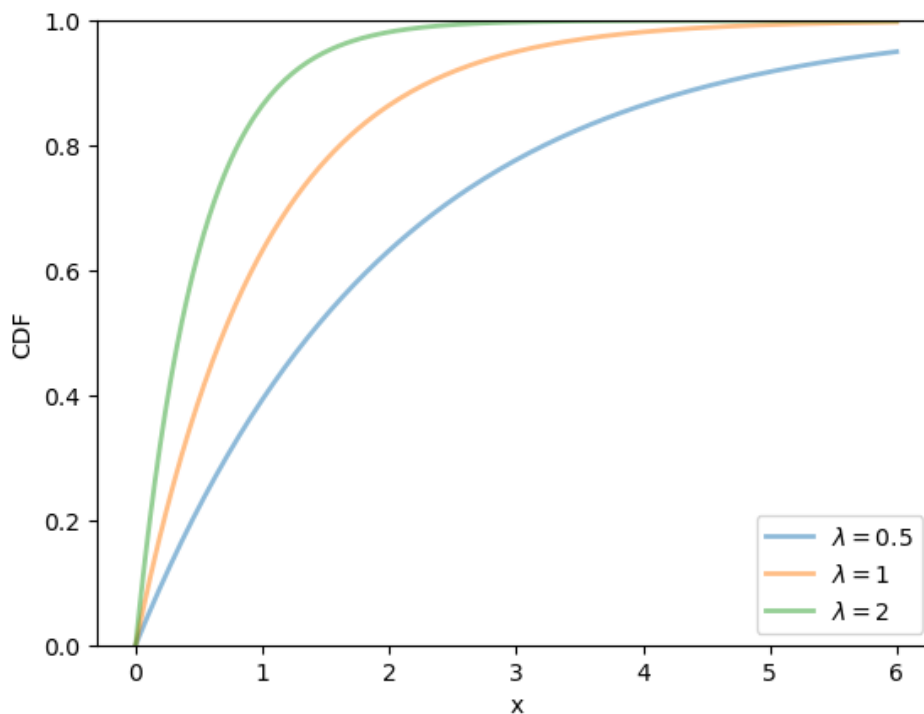

Beta distribution

The **beta distribution** is a distribution on $(0, 1)$ with density

$$p(x) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1 - x)^{\beta-1}$$

where Γ is the gamma function.

(The role of the gamma function is just to normalize the density, so that it integrates to one.)

This distribution has two parameters, $\alpha > 0$ and $\beta > 0$.

It can be shown that, for this distribution, the mean is $\alpha/(\alpha + \beta)$ and the variance is $\alpha\beta/(\alpha + \beta)^2(\alpha + \beta + 1)$.

We can obtain the moments, PDF, and CDF of the Beta density as follows:

```
α, β = 3.0, 1.0
u = scipy.stats.beta(α, β)
```

```
u.mean(), u.var()
```

```
(0.75, 0.0375)
```

```
α_vals = [0.5, 1, 5, 25, 3]
β_vals = [3, 1, 10, 20, 0.5]
x_grid = np.linspace(0, 1, 200)

fig, ax = plt.subplots()
for α, β in zip(α_vals, β_vals):
    u = scipy.stats.beta(α, β)
    ax.plot(x_grid, u.pdf(x_grid),
    alpha=0.5, lw=2,
    label=fr'$\alpha={α}, \beta={β}$')
ax.set_xlabel('x')
ax.set_ylabel('PDF')
plt.legend()
plt.show()
```
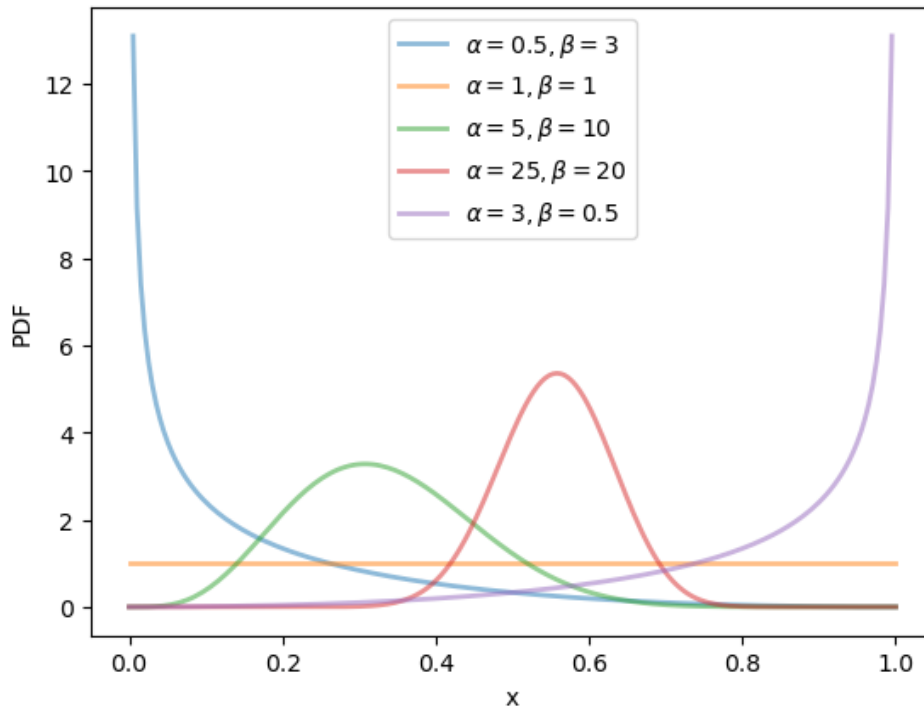
```
fig, ax = plt.subplots()
for α, β in zip(α_vals, β_vals):
    u = scipy.stats.beta(α, β)
    ax.plot(x_grid, u.cdf(x_grid),
    alpha=0.5, lw=2,
    label=fr'$\alpha={α}, \beta={β}$')
    ax.set_ylim(0, 1)
ax.set_xlabel('x')
ax.set_ylabel('CDF')
plt.legend()
plt.show()
```
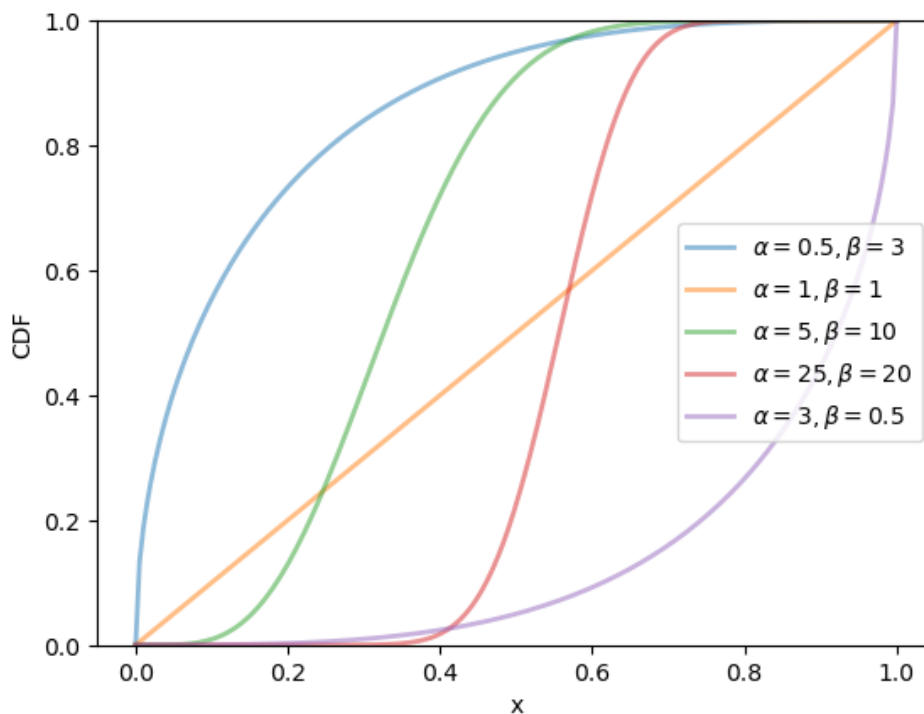

Gamma distribution

The **gamma distribution** is a distribution on $(0, \infty)$ with density

$$p(x) = \frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} \exp(-\beta x)$$

This distribution has two parameters, $\alpha > 0$ and $\beta > 0$.

It can be shown that, for this distribution, the mean is α/β and the variance is α/β^2.

One interpretation is that if X is gamma distributed and α is an integer, then X is the sum of α independent exponentially distributed random variables with mean $1/\beta$.

We can obtain the moments, PDF, and CDF of the Gamma density as follows:

```
α, β = 3.0, 2.0
u = scipy.stats.gamma(α, scale=1/β)
```

```
u.mean(), u.var()
```

```
(1.5, 0.75)
```

```python
α_vals = [1, 3, 5, 10]
β_vals = [3, 5, 3, 3]
x_grid = np.linspace(0, 7, 200)

fig, ax = plt.subplots()
for α, β in zip(α_vals, β_vals):
    u = scipy.stats.gamma(α, scale=1/β)
    ax.plot(x_grid, u.pdf(x_grid),
    alpha=0.5, lw=2,
    label=fr'$\alpha={α}, \beta={β}$')
ax.set_xlabel('x')
ax.set_ylabel('PDF')
plt.legend()
plt.show()
```
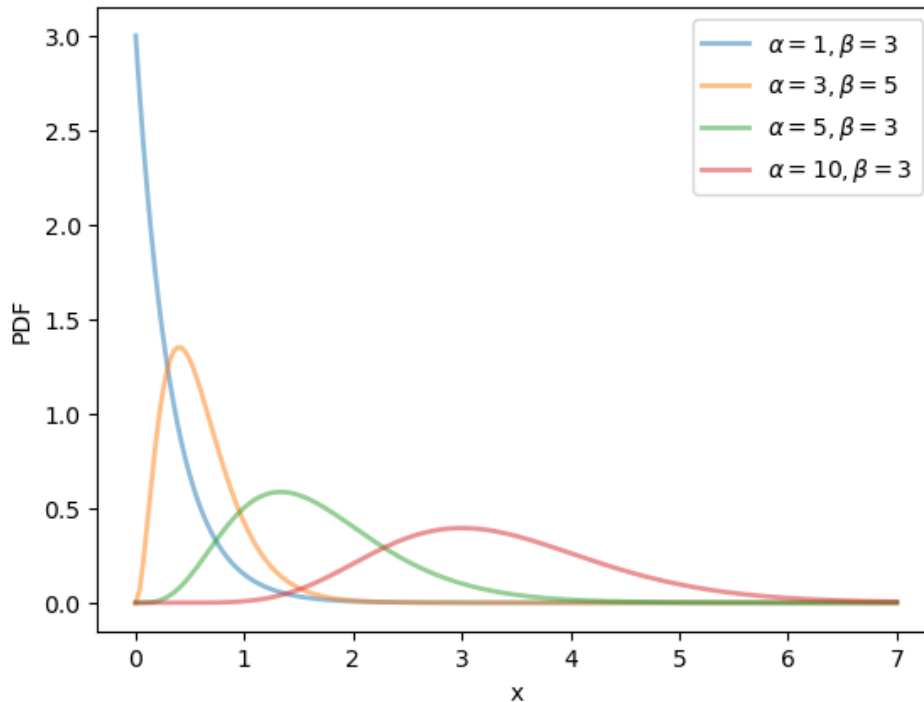
```
fig, ax = plt.subplots()
for α, β in zip(α_vals, β_vals):
    u = scipy.stats.gamma(α, scale=1/β)
    ax.plot(x_grid, u.cdf(x_grid),
    alpha=0.5, lw=2,
    label=fr'$\alpha={α}, \beta={β}$')
    ax.set_ylim(0, 1)
ax.set_xlabel('x')
ax.set_ylabel('CDF')
plt.legend()
plt.show()
```
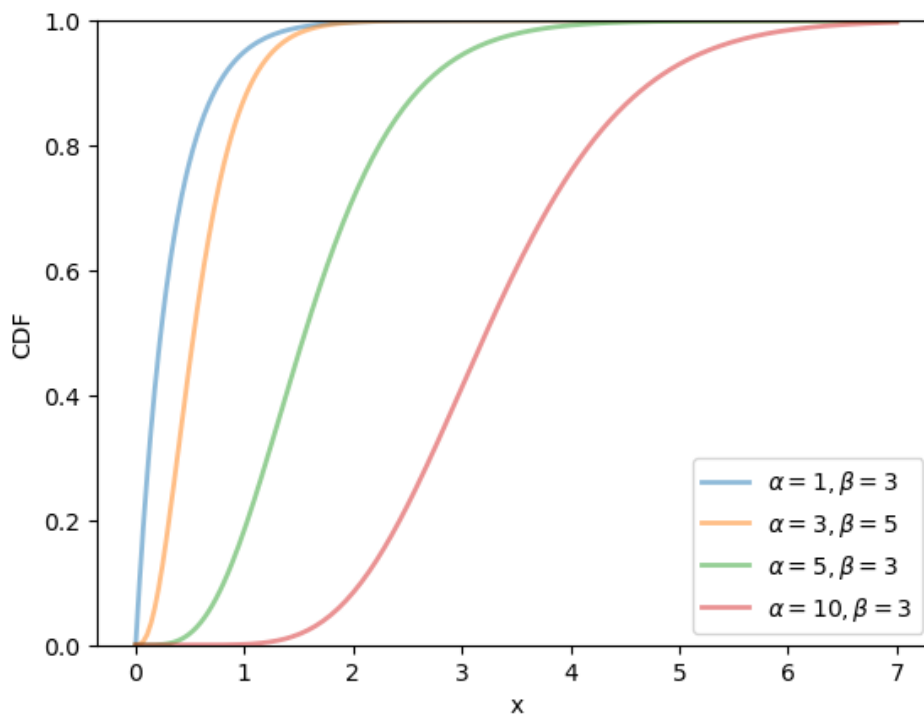

18.3 Observed distributions

Sometimes we refer to observed data or measurements as "distributions".

For example, let's say we observe the income of 10 people over a year:

```
data = [['Hiroshi', 1200],
        ['Ako', 1210],
        ['Emi', 1400],
        ['Daiki', 990],
```

```
        ['Chiyo', 1530],
        ['Taka', 1210],
        ['Katsuhiko', 1240],
        ['Daisuke', 1124],
        ['Yoshi', 1330],
        ['Rie', 1340]]

df = pd.DataFrame(data, columns=['name', 'income'])
df
```

```
         name  income
0      Hiroshi    1200
1          Ako    1210
2          Emi    1400
3        Daiki     990
4        Chiyo    1530
5         Taka    1210
6    Katsuhiko    1240
7      Daisuke    1124
8        Yoshi    1330
9          Rie    1340
```

In this situation, we might refer to the set of their incomes as the "income distribution."

The terminology is confusing because this set is not a probability distribution —it's just a collection of numbers.

However, as we will see, there are connections between observed distributions (i.e., sets of numbers like the income distribution above) and probability distributions.

Below we explore some observed distributions.

18.3.1 Summary statistics

Suppose we have an observed distribution with values $\{x_1, \dots, x_n\}$

The **sample mean** of this distribution is defined as

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

The **sample variance** is defined as

$$\frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2$$

For the income distribution given above, we can calculate these numbers via

```
x = df['income']
x.mean(), x.var()
```

```
(1257.4, 22680.933333333334)
```

Exercise 18.3.1

If you try to check that the formulas given above for the sample mean and sample variance produce the same numbers, you will see that the variance isn't quite right. This is because SciPy uses $1/(n-1)$ instead of $1/n$ as the term at the front of the variance. (Some books define the sample variance this way.) Confirm.

18.3.2 Visualization

Let's look at different ways that we can visualize one or more observed distributions.

We will cover

- histograms

- kernel density estimates and

- violin plots

Histograms

We can histogram the income distribution we just constructed as follows

```
fig, ax = plt.subplots()
ax.hist(x, bins=5, density=True, histtype='bar')
ax.set_xlabel('income')
ax.set_ylabel('density')
plt.show()
```
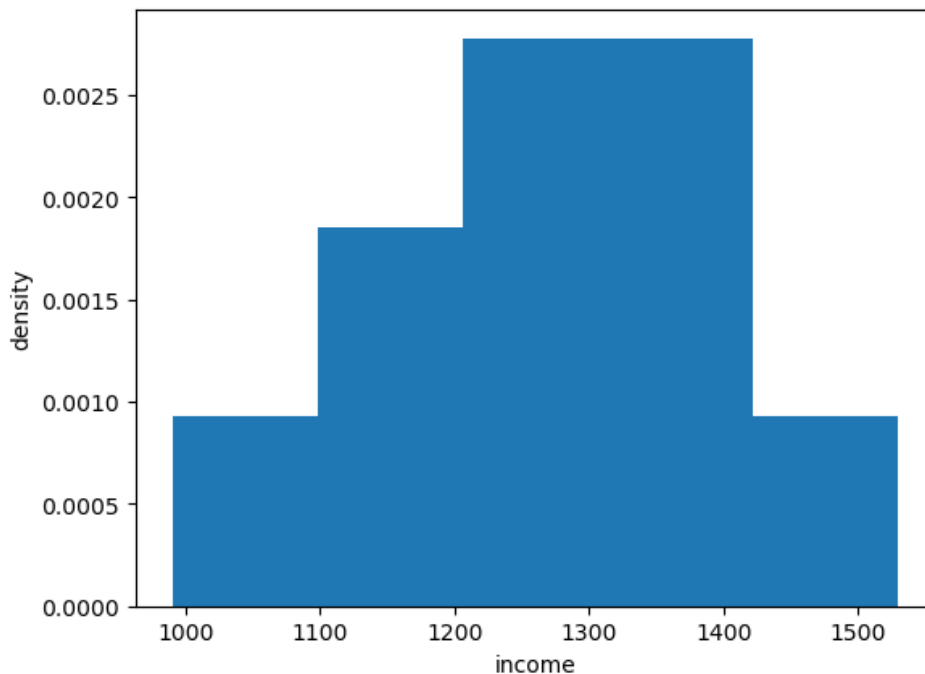
Let's look at a distribution from real data.

In particular, we will look at the monthly return on Amazon shares between 2000/1/1 and 2024/1/1.

The monthly return is calculated as the percent change in the share price over each month.

So we will have one observation for each month.

```
df = yf.download('AMZN', '2000-1-1', '2024-1-1', interval='1mo')
prices = df['Adj Close']
x_amazon = prices.pct_change()[1:] * 100
x_amazon.head()
```

The first observation is the monthly return (percent change) over January 2000, which was

```
x_amazon.iloc[0]
```

```
6.6795679502808625
```

Let's turn the return observations into an array and histogram it.

```
fig, ax = plt.subplots()
ax.hist(x_amazon, bins=20)
ax.set_xlabel('monthly return (percent change)')
ax.set_ylabel('density')
plt.show()
```
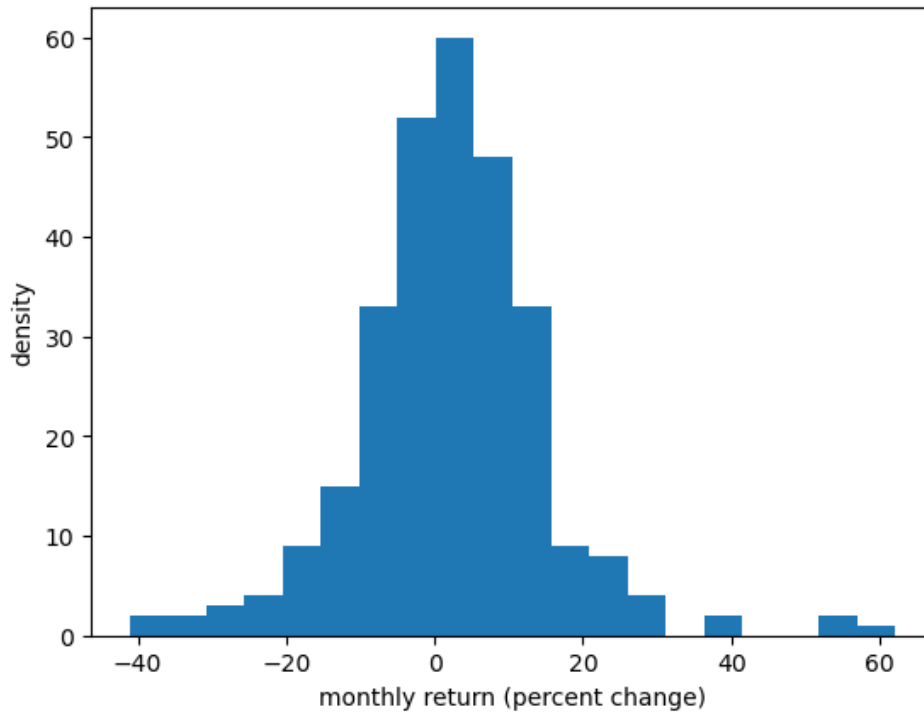
Kernel density estimates

Kernel density estimates (KDE) provide a simple way to estimate and visualize the density of a distribution.

If you are not familiar with KDEs, you can think of them as a smoothed histogram.

Let's have a look at a KDE formed from the Amazon return data.

```
fig, ax = plt.subplots()
sns.kdeplot(x_amazon, ax=ax)
ax.set_xlabel('monthly return (percent change)')
ax.set_ylabel('KDE')
plt.show()
```
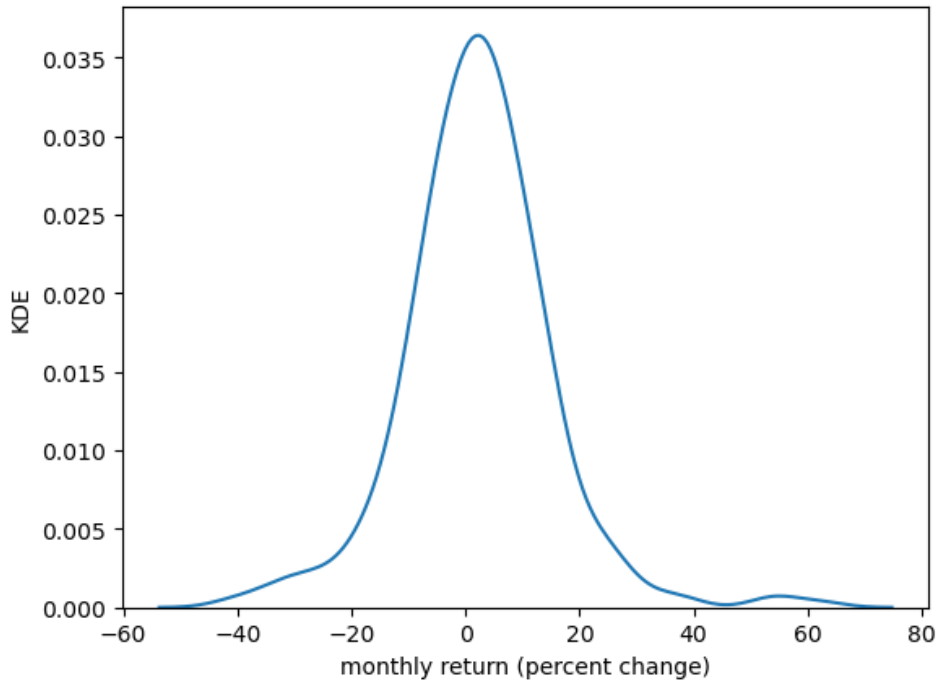
The smoothness of the KDE is dependent on how we choose the bandwidth.

```
fig, ax = plt.subplots()
sns.kdeplot(x_amazon, ax=ax, bw_adjust=0.1, alpha=0.5, label="bw=0.1")
sns.kdeplot(x_amazon, ax=ax, bw_adjust=0.5, alpha=0.5, label="bw=0.5")
sns.kdeplot(x_amazon, ax=ax, bw_adjust=1, alpha=0.5, label="bw=1")
ax.set_xlabel('monthly return (percent change)')
ax.set_ylabel('KDE')
plt.legend()
plt.show()
```
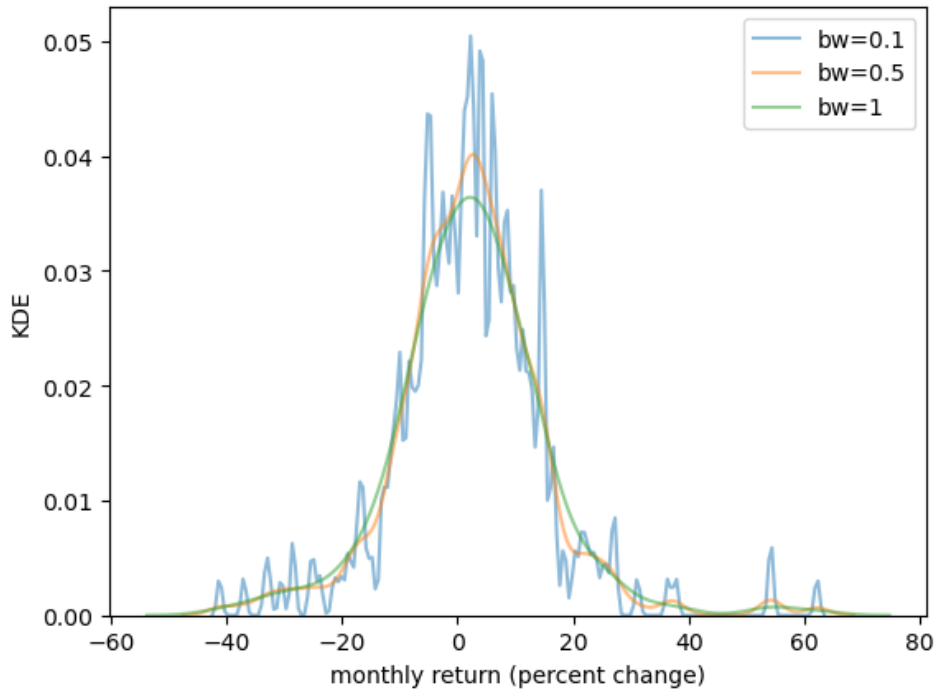
When we use a larger bandwidth, the KDE is smoother.

A suitable bandwidth is not too smooth (underfitting) or too wiggly (overfitting).

Violin plots

Another way to display an observed distribution is via a violin plot.

```
fig, ax = plt.subplots()
ax.violinplot(x_amazon)
ax.set_ylabel('monthly return (percent change)')
ax.set_xlabel('KDE')
plt.show()
```
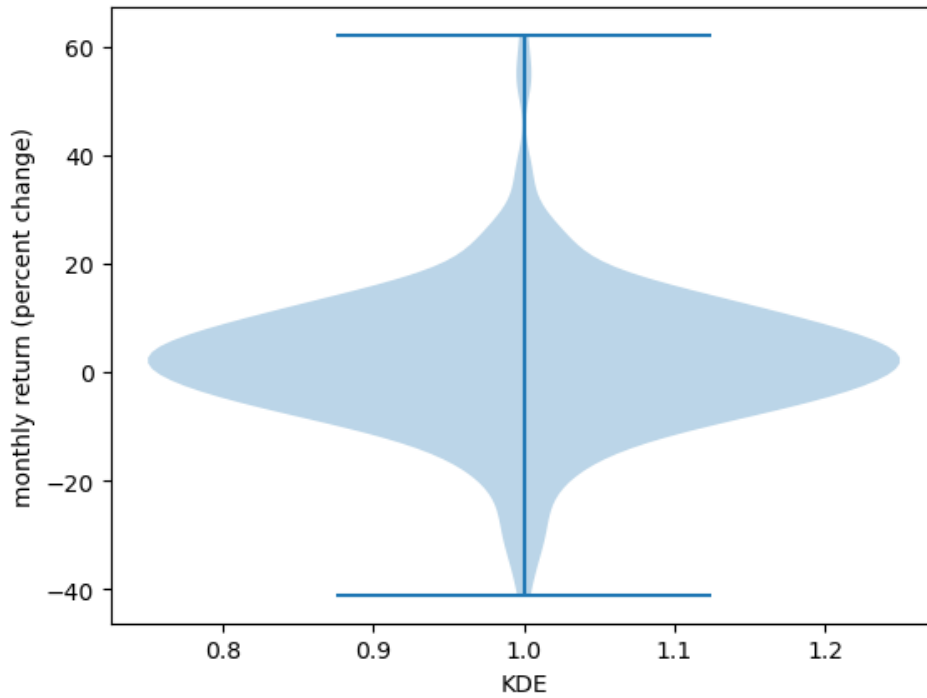
Violin plots are particularly useful when we want to compare different distributions.

For example, let's compare the monthly returns on Amazon shares with the monthly return on Costco shares.

```
df = yf.download('COST', '2000-1-1', '2024-1-1', interval='1mo')
prices = df['Adj Close']
x_costco = prices.pct_change()[1:] * 100
```

```
fig, ax = plt.subplots()
ax.violinplot([x_amazon, x_costco])
ax.set_ylabel('monthly return (percent change)')
ax.set_xlabel('retailers')

ax.set_xticks([1, 2])
ax.set_xticklabels(['Amazon', 'Costco'])
plt.show()
```
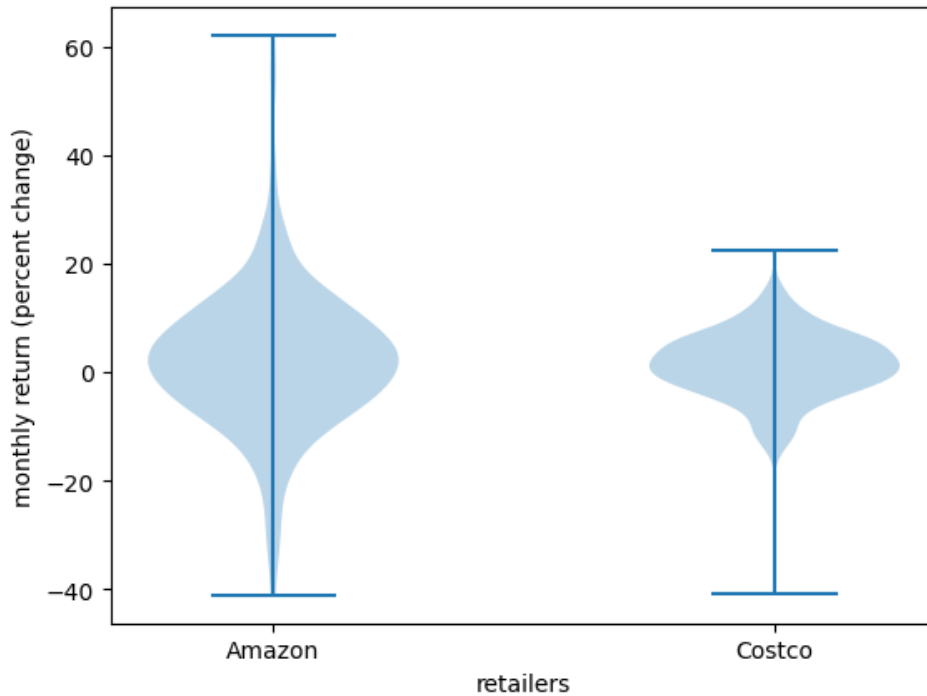
18.3.3 Connection to probability distributions

Let's discuss the connection between observed distributions and probability distributions.

Sometimes it's helpful to imagine that an observed distribution is generated by a particular probability distribution.

For example, we might look at the returns from Amazon above and imagine that they were generated by a normal distribution.

(Even though this is not true, it *might* be a helpful way to think about the data.)

Here we match a normal distribution to the Amazon monthly returns by setting the sample mean to the mean of the normal distribution and the sample variance equal to the variance.

Then we plot the density and the histogram.

```
μ = x_amazon.mean()
σ_squared = x_amazon.var()
σ = np.sqrt(σ_squared)
u = scipy.stats.norm(μ, σ)
```

```
x_grid = np.linspace(-50, 65, 200)
fig, ax = plt.subplots()
ax.plot(x_grid, u.pdf(x_grid))
```

```
ax.hist(x_amazon, density=True, bins=40)
ax.set_xlabel('monthly return (percent change)')
ax.set_ylabel('density')
plt.show()
```
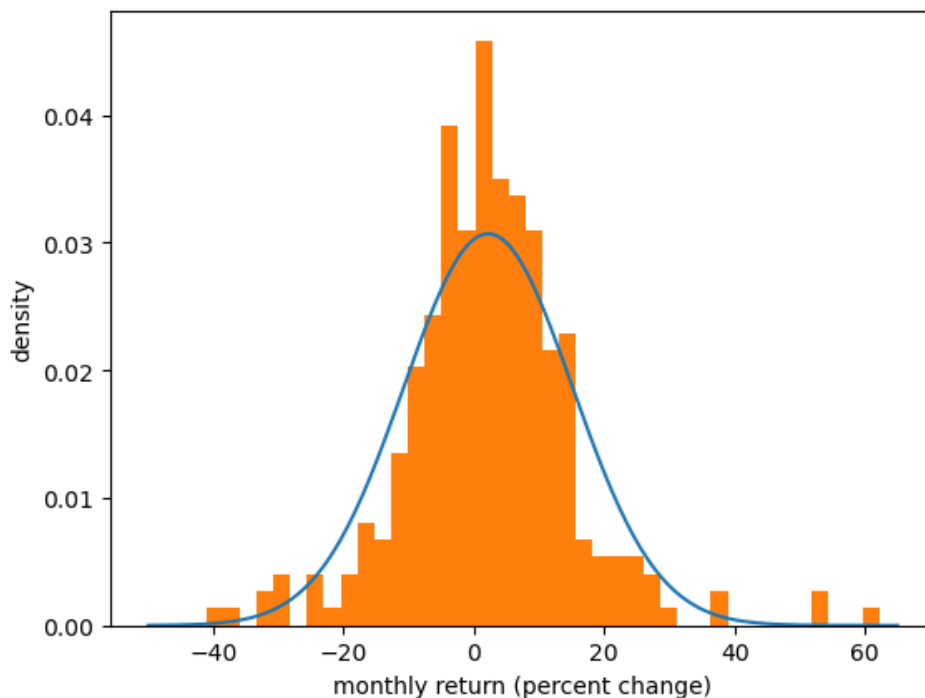

The match between the histogram and the density is not bad but also not very good.

One reason is that the normal distribution is not really a good fit for this observed data —we will discuss this point again when we talk about *heavy tailed distributions*.

Of course, if the data really *is* generated by the normal distribution, then the fit will be better.

Let's see this in action

- first we generate random draws from the normal distribution

- then we histogram them and compare with the density.

```
μ, σ = 0, 1
u = scipy.stats.norm(μ, σ)
N = 2000  # Number of observations
x_draws = u.rvs(N)
x_grid = np.linspace(-4, 4, 200)
fig, ax = plt.subplots()
ax.plot(x_grid, u.pdf(x_grid))
```

18.3. Observed distributions   **377**

```
ax.hist(x_draws, density=True, bins=40)
ax.set_xlabel('x')
ax.set_ylabel('density')
plt.show()
```
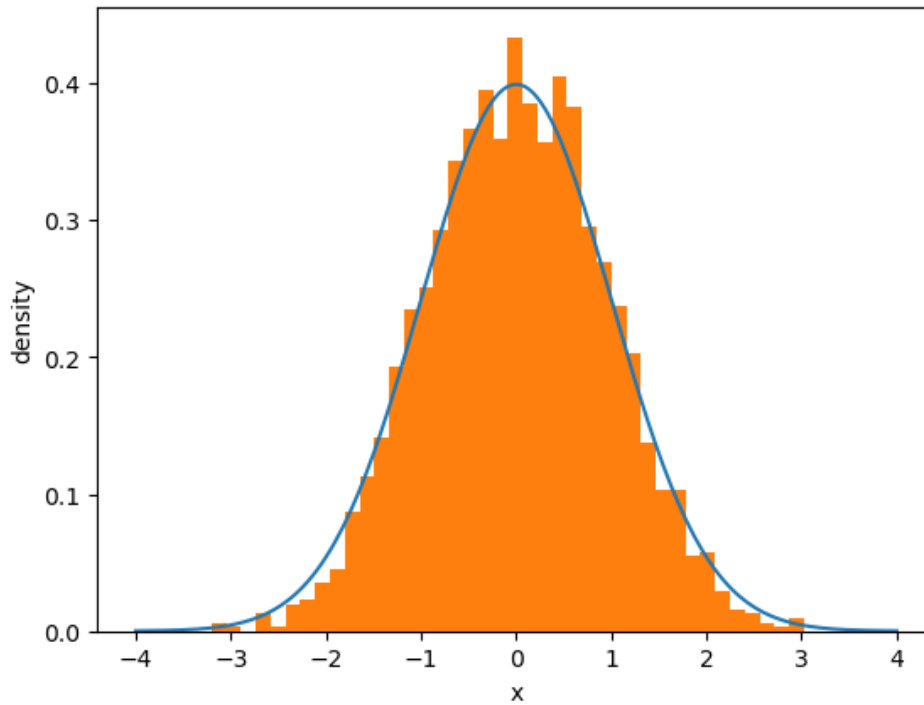

Note that if you keep increasing N, which is the number of observations, the fit will get better and better.

This convergence is a version of the "law of large numbers", which we will discuss *later*.

LLN AND CLT

19.1 Overview

This lecture illustrates two of the most important results in probability and statistics:

1. the law of large numbers (LLN) and

2. the central limit theorem (CLT).

These beautiful theorems lie behind many of the most fundamental results in econometrics and quantitative economic modeling.

The lecture is based around simulations that show the LLN and CLT in action.

We also demonstrate how the LLN and CLT break down when the assumptions they are based on do not hold.

This lecture will focus on the univariate case (the multivariate case is treated in a more advanced lecture).

We'll need the following imports:

```python
import matplotlib.pyplot as plt
import numpy as np
import scipy.stats as st
```

19.2 The law of large numbers

We begin with the law of large numbers, which tells us when sample averages will converge to their population means.

19.2.1 The LLN in action

Let's see an example of the LLN in action before we go further.

Example 19.2.1

Consider a Bernoulli random variable X with parameter p.

This means that X takes values in $\{0, 1\}$ and $\mathbb{P}\{X = 1\} = p$.

We can think of drawing X as tossing a biased coin where

- the coin falls on "heads" with probability p and

- the coin falls on "tails" with probability $1 - p$

We set $X = 1$ if the coin is "heads" and zero otherwise.

The (population) mean of X is

$$\mathbb{E}X = 0 \cdot \mathbb{P}\{X = 0\} + 1 \cdot \mathbb{P}\{X = 1\} = \mathbb{P}\{X = 1\} = p$$

We can generate a draw of X with `scipy.stats` (imported as `st`) as follows:

```
p = 0.8
X = st.bernoulli.rvs(p)
print(X)
```

```
1
```

In this setting, the LLN tells us if we flip the coin many times, the fraction of heads that we see will be close to the mean p.

We use n to represent the number of times the coin is flipped.

Let's check this:

```
n = 1_000_000
X_draws = st.bernoulli.rvs(p, size=n)
print(X_draws.mean()) # count the number of 1's and divide by n
```

```
0.799911
```

If we change p the claim still holds:

```
p = 0.3
X_draws = st.bernoulli.rvs(p, size=n)
print(X_draws.mean())
```

```
0.30052
```

Let's connect this to the discussion above, where we said the sample average converges to the "population mean".

Think of $X_1, \dots, X_n$ as independent flips of the coin.

The population mean is the mean in an infinite sample, which equals the expectation $\mathbb{E}X$.

The sample mean of the draws $X_1, \dots, X_n$ is

$$\bar{X}_n := \frac{1}{n} \sum_{i=1}^n X_i$$

In this case, it is the fraction of draws that equal one (the number of heads divided by n).

Thus, the LLN tells us that for the Bernoulli trials above

$$\bar{X}_n \to \mathbb{E}X = p \qquad (n \to \infty) \tag{19.1}$$

This is exactly what we illustrated in the code.

19.2.2 Statement of the LLN

Let's state the LLN more carefully.

Let $X_1, \dots, X_n$ be random variables, all of which have the same distribution.

These random variables can be continuous or discrete.

For simplicity we will

- assume they are continuous and

- let f denote their common density function

The last statement means that for any i in $\{1, \dots, n\}$ and any numbers a, b,

$$\mathbb{P}\{a \le X_i \le b\} = \int_a^b f(x) dx$$

(For the discrete case, we need to replace densities with probability mass functions and integrals with sums.)

Let μ denote the common mean of this sample.

Thus, for each i,

$$\mu := \mathbb{E}X_i = \int_{-\infty}^{\infty} x f(x) dx$$

The sample mean is

$$\bar{X}_n := \frac{1}{n} \sum_{i=1}^{n} X_i$$

The next theorem is called Kolmogorov's strong law of large numbers.

Theorem 19.2.1

If $X_1, \ldots, X_n$ are IID and $\mathbb{E}|X|$ is finite, then

$$\mathbb{P}\left\{\bar{X}_n \to \mu \text{ as } n \to \infty\right\} = 1 \tag{19.2}$$

Here

- IID means independent and identically distributed and

- $\mathbb{E}|X| = \int_{-\infty}^{\infty} |x| f(x) dx$

19.2.3 Comments on the theorem

What does the probability one statement in the theorem mean?

Let's think about it from a simulation perspective, imagining for a moment that our computer can generate perfect random samples (although this isn't strictly true).

Let's also imagine that we can generate infinite sequences so that the statement $\bar{X}_n \to \mu$ can be evaluated.

In this setting, (19.2) should be interpreted as meaning that the probability of the computer producing a sequence where $\bar{X}_n \to \mu$ fails to occur is zero.

19.2.4 Illustration

Let's illustrate the LLN using simulation.

When we illustrate it, we will use a key idea: the sample mean $\bar{X}_n$ is itself a random variable.

The reason $\bar{X}_n$ is a random variable is that it's a function of the random variables $X_1, \ldots, X_n$.

What we are going to do now is

1. pick some fixed distribution to draw each X_i from

2. set n to some large number

and then repeat the following three instructions.

---

1. generate the draws $X_1, \dots, X_n$

2. calculate the sample mean $\bar{X}_n$ and record its value in an array `sample_means`

3. go to step 1.

We will loop over these three steps m times, where m is some large integer.

The array `sample_means` will now contain m draws of the random variable $\bar{X}_n$.

If we histogram these observations of $\bar{X}_n$, we should see that they are clustered around the population mean $\mathbb{E}X$.

Moreover, if we repeat the exercise with a larger value of n, we should see that the observations are even more tightly clustered around the population mean.

This is, in essence, what the LLN is telling us.

To implement these steps, we will use functions.

Our first function generates a sample mean of size n given a distribution.

```python
def draw_means(X_distribution,   # The distribution of each X_i
              n):                # The size of the sample mean

    # Generate n draws: X_1, ..., X_n
    X_samples = X_distribution.rvs(size=n)

    # Return the sample mean
    return np.mean(X_samples)
```

Now we write a function to generate m sample means and histogram them.

```python
def generate_histogram(X_distribution, n, m):

    # Compute m sample means

    sample_means = np.empty(m)
    for j in range(m):
      sample_means[j] = draw_means(X_distribution, n)

    # Generate a histogram

    fig, ax = plt.subplots()
    ax.hist(sample_means, bins=30, alpha=0.5, density=True)
    μ = X_distribution.mean()  # Get the population mean
    σ = X_distribution.std()   # and the standard deviation
    ax.axvline(x=μ, ls="--", c="k", label=fr"$\mu = {μ}$")
```

(continues on next page)

```
    ax.set_xlim(μ - σ, μ + σ)
    ax.set_xlabel(r'$\bar X_n$', size=12)
    ax.set_ylabel('density', size=12)
    ax.legend()
    plt.show()
```

Now we call the function.

```
# pick a distribution to draw each $X_i$ from
X_distribution = st.norm(loc=5, scale=2)
# Call the function
generate_histogram(X_distribution, n=1_000, m=1000)
```
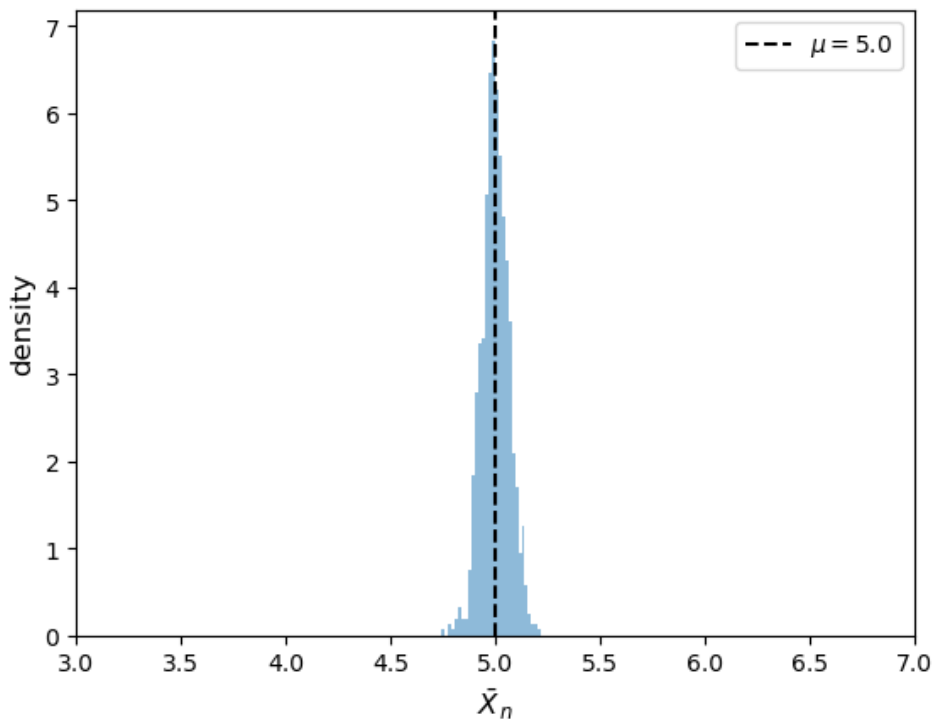

We can see that the distribution of $\bar X$ is clustered around $\mathbb{E} X$ as expected.

Let's vary n to see how the distribution of the sample mean changes.

We will use a violin plot to show the different distributions.

Each distribution in the violin plot represents the distribution of X_n for some n, calculated by simulation.

```
def means_violin_plot(distribution,
                      ns = [1_000, 10_000, 100_000],
```

```
                    m = 10_000):

    data = []
    for n in ns:
        sample_means = [draw_means(distribution, n) for i in range(m)]
        data.append(sample_means)

    fig, ax = plt.subplots()

    ax.violinplot(data)
    μ = distribution.mean()
    ax.axhline(y=μ, ls="--", c="k", label=fr"$\mu = {μ}$")

    labels=[fr'$n = {n}$' for n in ns]

    ax.set_xticks(np.arange(1, len(labels) + 1), labels=labels)
    ax.set_xlim(0.25, len(labels) + 0.75)



    plt.subplots_adjust(bottom=0.15, wspace=0.05)

    ax.set_ylabel('density', size=12)
    ax.legend()
    plt.show()
```

Let's try with a normal distribution.

```
means_violin_plot(st.norm(loc=5, scale=2))
```
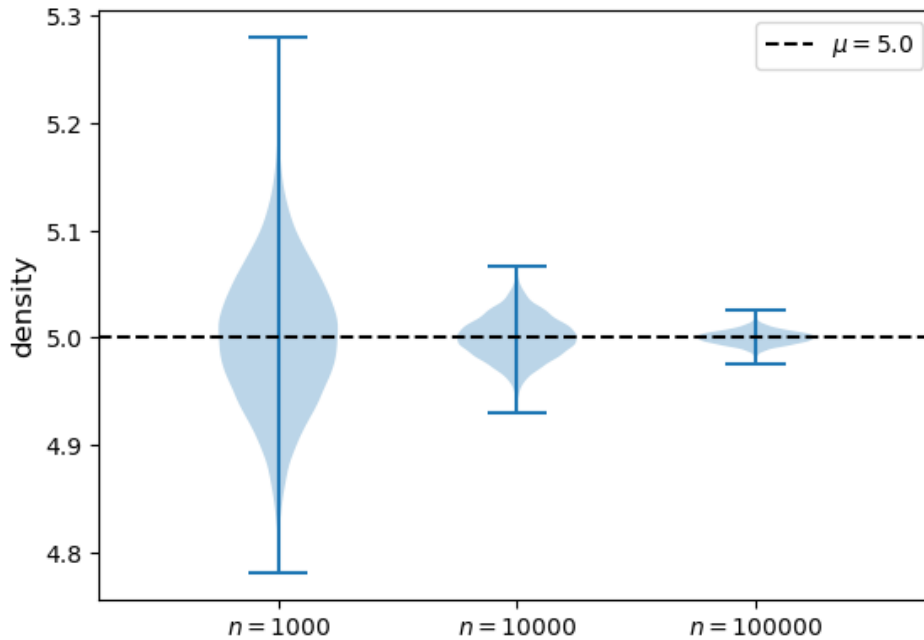
As n gets large, more probability mass clusters around the population mean μ.

Now let's try with a Beta distribution.

```
means_violin_plot(st.beta(6, 6))
```
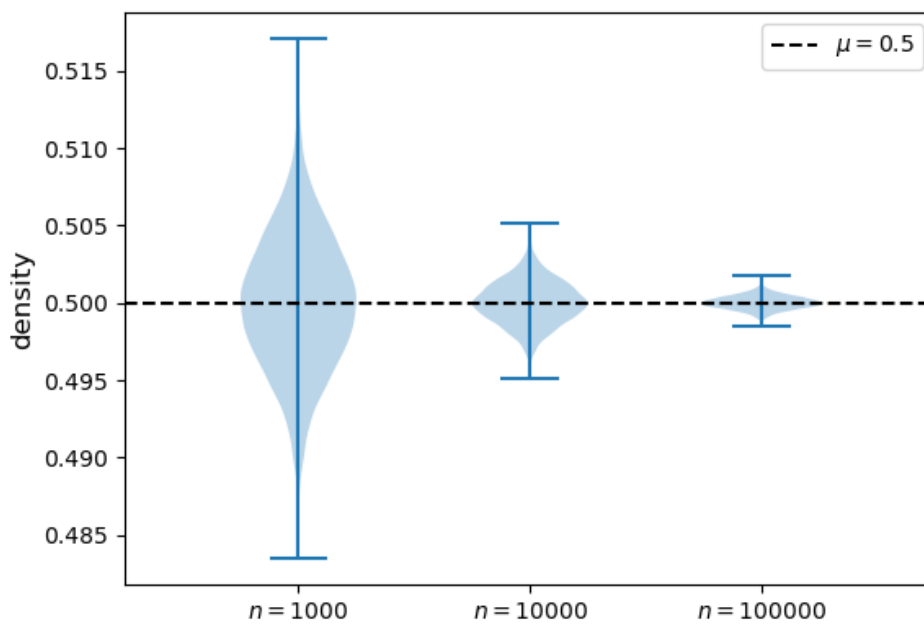

We get a similar result.

19.3 Breaking the LLN

We have to pay attention to the assumptions in the statement of the LLN.

If these assumptions do not hold, then the LLN might fail.

19.3.1 Infinite first moment

As indicated by the theorem, the LLN can break when $\mathbb{E}|X|$ is not finite.

We can demonstrate this using the Cauchy distribution.

The Cauchy distribution has the following property:

If $X_1, \ldots, X_n$ are IID and Cauchy, then so is $\bar{X}_n$.

This means that the distribution of $\bar{X}_n$ does not eventually concentrate on a single number.

Hence the LLN does not hold.

The LLN fails to hold here because the assumption $\mathbb{E}|X| < \infty$ is violated by the Cauchy distribution.

19.3.2 Failure of the IID condition

The LLN can also fail to hold when the IID assumption is violated.

Example 19.3.1

$$X_0 \sim N(0,1) \quad \text{and} \quad X_i = X_{i-1} \quad \text{for} \quad i = 1, \ldots, n$$

In this case,

$$\bar{X}_n = \frac{1}{n} \sum_{i=1}^{n} X_i = X_0 \sim N(0,1)$$

Therefore, the distribution of $\bar{X}_n$ is $N(0,1)$ for all n!

Does this contradict the LLN, which says that the distribution of $\bar{X}_n$ collapses to the single point μ?

No, the LLN is correct —the issue is that its assumptions are not satisfied.

In particular, the sequence $X_1, \ldots, X_n$ is not independent.

Note: Although in this case the violation of IID breaks the LLN, there *are* situations where IID fails but the LLN still holds.

We will show an example in the exercise.

19.4 Central limit theorem

Next, we turn to the central limit theorem (CLT), which tells us about the distribution of the deviation between sample averages and population means.

19.4.1 Statement of the theorem

The central limit theorem is one of the most remarkable results in all of mathematics.

In the IID setting, it tells us the following:

Theorem 19.4.1

If $X_1, \dots, X_n$ is IID with common mean μ and common variance $\sigma^2 \in (0, \infty)$, then

$$\sqrt{n}(\bar{X}_n - \mu) \overset{d}{\to} N(0, \sigma^2) \quad \text{as} \quad n \to \infty \tag{19.3}$$

Here $\overset{d}{\to} N(0, \sigma^2)$ indicates convergence in distribution to a centered (i.e., zero mean) normal with standard deviation σ.

The striking implication of the CLT is that for any distribution with finite second moment, the simple operation of adding independent copies always leads to a Gaussian(Normal) curve.

19.4.2 Simulation 1

Since the CLT seems almost magical, running simulations that verify its implications is one good way to build understanding.

To this end, we now perform the following simulation

1. Choose an arbitrary distribution F for the underlying observations X_i.

2. Generate independent draws of $Y_n := \sqrt{n}(\bar{X}_n - \mu)$.

3. Use these draws to compute some measure of their distribution —such as a histogram.

4. Compare the latter to $N(0, \sigma^2)$.

Here's some code that does exactly this for the exponential distribution $F(x) = 1 - e^{-\lambda x}$.

(Please experiment with other choices of F, but remember that, to conform with the conditions of the CLT, the distribution must have a finite second moment.)

```
# Set parameters
n = 250          # Choice of n
k = 1_000_000          # Number of draws of Y_n
distribution = st.expon(2) # Exponential distribution, λ = 1/2
μ, σ = distribution.mean(), distribution.std()

# Draw underlying RVs. Each row contains a draw of X_1,..,X_n
data = distribution.rvs((k, n))
# Compute mean of each row, producing k draws of \bar X_n
sample_means = data.mean(axis=1)
# Generate observations of Y_n
Y = np.sqrt(n) * (sample_means - μ)

# Plot
fig, ax = plt.subplots(figsize=(10, 6))
xmin, xmax = -3 * σ, 3 * σ
ax.set_xlim(xmin, xmax)
ax.hist(Y, bins=60, alpha=0.4, density=True)
xgrid = np.linspace(xmin, xmax, 200)
ax.plot(xgrid, st.norm.pdf(xgrid, scale=σ),
        'k-', lw=2, label='$N(0, \sigma^2)$')
ax.set_xlabel(r"$Y_n$", size=12)
ax.set_ylabel(r"$density$", size=12)

ax.legend()

plt.show()
```
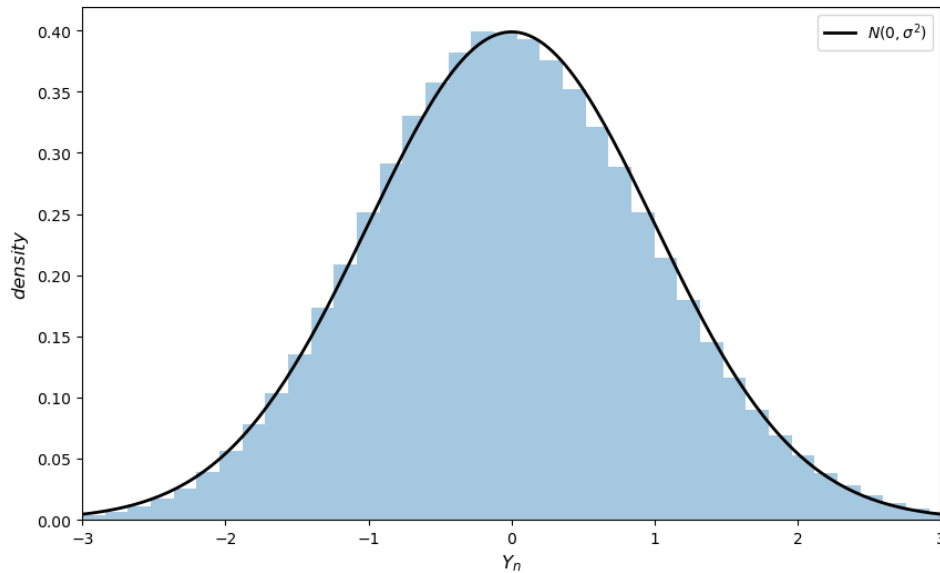
(Notice the absence of for loops —every operation is vectorized, meaning that the major calculations are all shifted to fast C code.)

The fit to the normal density is already tight and can be further improved by increasing n.

19.5 Exercises

Exercise 19.5.1

Repeat the simulation *above* with the Beta distribution.

You can choose any $\alpha > 0$ and $\beta > 0$.

Solution to Exercise 19.5.1

```
# Set parameters
n = 250           # Choice of n
k = 1_000_000         # Number of draws of Y_n
distribution = st.beta(2,2) # We chose Beta(2, 2) as an example
μ, σ = distribution.mean(), distribution.std()

# Draw underlying RVs. Each row contains a draw of X_1,..,X_n
data = distribution.rvs((k, n))
# Compute mean of each row, producing k draws of \bar X_n
sample_means = data.mean(axis=1)
```

(continues on next page)

```python
# Generate observations of Y_n
Y = np.sqrt(n) * (sample_means - μ)

# Plot
fig, ax = plt.subplots(figsize=(10, 6))
xmin, xmax = -3 * σ, 3 * σ
ax.set_xlim(xmin, xmax)
ax.hist(Y, bins=60, alpha=0.4, density=True)
ax.set_xlabel(r"$Y_n$", size=12)
ax.set_ylabel(r"$density$", size=12)
xgrid = np.linspace(xmin, xmax, 200)
ax.plot(xgrid, st.norm.pdf(xgrid, scale=σ), 'k-', lw=2, label='$N(0, \sigma^2)$')
ax.legend()

plt.show()
```
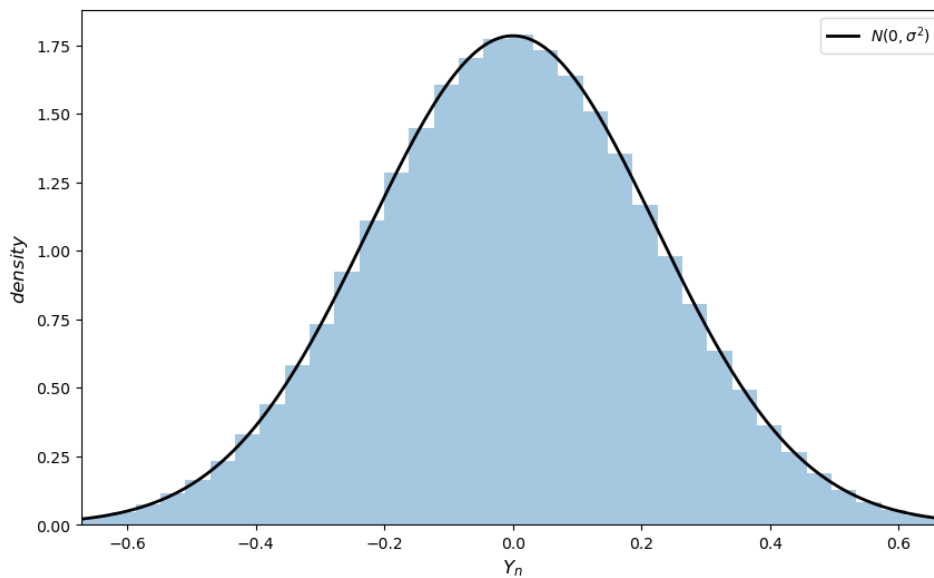

Exercise 19.5.2

At the start of this lecture we discussed Bernoulli random variables.

NumPy doesn't provide a `bernoulli` function that we can sample from.

However, we can generate a draw of Bernoulli X using NumPy via

```python
U = np.random.rand()
X = 1 if U < p else 0
```

```
print(X)
```

Explain why this provides a random variable X with the right distribution.

Solution to Exercise 19.5.2

We can write X as $X = \mathbf{1}\{U < p\}$ where $\mathbf{1}$ is the indicator function (i.e., 1 if the statement is true and zero otherwise).

Here we generated a uniform draw U on $[0, 1]$ and then used the fact that

$$\mathbb{P}\{0 \leq U < p\} = p - 0 = p$$

This means that $X = \mathbf{1}\{U < p\}$ has the right distribution.

Exercise 19.5.3

We mentioned above that LLN can still hold sometimes when IID is violated.

Let's investigate this claim further.

Consider the AR(1) process

$$X_{t+1} = \alpha + \beta X_t + \sigma \epsilon_{t+1}$$

where α, β, σ are constants and $\epsilon_1, \epsilon_2, \ldots$ are IID and standard normal.

Suppose that

$$X_0 \sim N\left(\frac{\alpha}{1 - \beta}, \frac{\sigma^2}{1 - \beta^2}\right)$$

This process violates the independence assumption of the LLN (since X_{t+1} depends on the value of X_t).

However, the next exercise teaches us that LLN type convergence of the sample mean to the population mean still occurs.

1. Prove that the sequence $X_1, X_2, \ldots$ is identically distributed.

2. Show that LLN convergence holds using simulations with $\alpha = 0.8, \beta = 0.2$.

Solution to Exercise 19.5.3

Q1 Solution

Regarding part 1, we claim that X_t has the same distribution as X_0 for all t.

To construct a proof, we suppose that the claim is true for X_t.

Now we claim it is also true for X_{t+1}.

Observe that we have the correct mean:

$$\mathbb{E}X_{t+1} = \alpha + \beta\mathbb{E}X_t$$
$$= \alpha + \beta\frac{\alpha}{1-\beta}$$
$$= \frac{\alpha}{1-\beta}$$

We also have the correct variance:

$$\text{Var}(X_{t+1}) = \beta^2\text{Var}(X_t) + \sigma^2$$
$$= \frac{\beta^2\sigma^2}{1-\beta^2} + \sigma^2$$
$$= \frac{\sigma^2}{1-\beta^2}$$

Finally, since both X_t and ϵ_0 are normally distributed and independent from each other, any linear combination of these two variables is also normally distributed.

We have now shown that

$$X_{t+1} \sim N\left(\frac{\alpha}{1-\beta}, \frac{\sigma^2}{1-\beta^2}\right)$$

We can conclude this AR(1) process violates the independence assumption but is identically distributed.

Q2 Solution

```
σ = 10
α = 0.8
β = 0.2
n = 100_000


fig, ax = plt.subplots(figsize=(10, 6))
x = np.ones(n)
x[0] = st.norm.rvs(α/(1-β), α**2/(1-β**2))
ε = st.norm.rvs(size=n+1)
means = np.ones(n)
means[0] = x[0]
for t in range(n-1):
    x[t+1] = α + β * x[t] + σ * ε[t+1]
    means[t+1] = np.mean(x[:t+1])



ax.scatter(range(100, n), means[100:n], s=10, alpha=0.5)
```

```
ax.set_xlabel(r"$n$", size=12)
ax.set_ylabel(r"$\bar X_n$", size=12)
yabs_max = max(ax.get_ylim(), key=abs)
ax.axhline(y=α/(1-β), ls="--", lw=3,
           label=r"$\mu = \frac{\alpha}{1-\beta}$",
           color = 'black')

plt.legend()
plt.show()
```
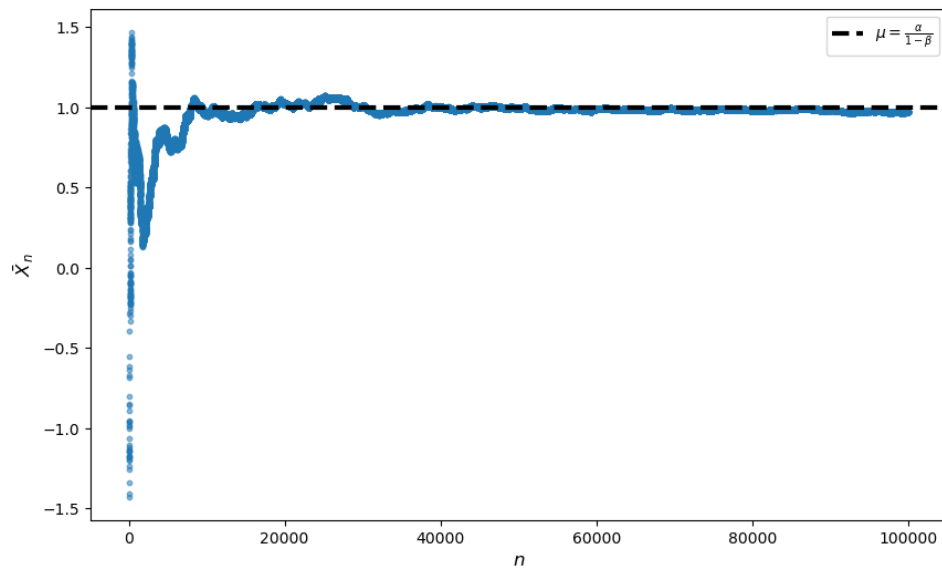

We see the convergence of $\bar x$ around μ even when the independence assumption is violated.

MONTE CARLO AND OPTION PRICING

20.1 Overview

Simple probability calculations can be done either

- with pencil and paper, or

- by looking up facts about well known probability distributions, or

- in our heads.

For example, we can easily work out

- the probability of three heads in five flips of a fair coin

- the expected value of a random variable that equals -10 with probability $1/2$ and 100 with probability $1/2$.

But some probability calculations are very complex.

Complex calculations concerning probabilities and expectations occur in many economic and financial problems.

Perhaps the most important tool for handling complicated probability calculations is Monte Carlo methods.

In this lecture we introduce Monte Carlo methods for computing expectations, with some applications in finance.

We will use the following imports.

```python
import numpy as np
import matplotlib.pyplot as plt
from numpy.random import randn
```

20.2 An introduction to Monte Carlo

In this section we describe how Monte Carlo can be used to compute expectations.

20.2.1 Share price with known distribution

Suppose that we are considering buying a share in some company.

Our plan is either to

1. buy the share now, hold it for one year and then sell it, or

2. do something else with our money.

We start by thinking of the share price in one year as a random variable S.

Before deciding whether or not to buy the share, we need to know some features of the distribution of S.

For example, suppose the mean of S is high relative to the price of buying the share.

This suggests we have a good chance of selling at a relatively high price.

Suppose, however, that the variance of S is also high.

This suggests that buying the share is risky, so perhaps we should refrain.

Either way, this discussion shows the importance of understanding the distribution of S.

Suppose that, after analyzing the data, we guess that S is well represented by a lognormal distribution with parameters μ, σ.

- S has the same distribution as $\exp(\mu + \sigma Z)$ where Z is standard normal.

- We write this statement as $S \sim LN(\mu, \sigma)$.

Any good reference on statistics (such as Wikipedia) will tell us that the mean and variance are

$$\mathbb{E} S = \exp\left(\mu + \frac{\sigma^2}{2}\right)$$

and

$$\operatorname{Var} S = [\exp(\sigma^2) - 1]\exp(2\mu + \sigma^2)$$

So far we have no need for a computer.

20.2.2 Share price with unknown distribution

But now suppose that we study the distribution of S more carefully.

We decide that the share price depends on three variables, X_1, X_2, and X_3 (e.g., sales, inflation, and interest rates).

In particular, our study suggests that

$$S = (X_1 + X_2 + X_3)^p$$

where

- p is a positive number, which is known to us (i.e., has been estimated),

- $X_i \sim LN(\mu_i, \sigma_i)$ for $i = 1, 2, 3$,

- the values μ_i, σ_i are also known, and

- the random variables X_1, X_2 and X_3 are independent.

How should we compute the mean of S?

To do this with pencil and paper is hard (unless, say, $p = 1$).

But fortunately there's an easy way to do this, at least approximately.

This is the Monte Carlo method, which runs as follows:

1. Generate n independent draws of X_1, X_2 and X_3 on a computer,

2. use these draws to generate n independent draws of S, and

3. take the average value of these draws of S.

This average will be close to the true mean when n is large.

This is due to the law of large numbers, which we discussed in *LLN and CLT*.

We use the following values for p and each μ_i and σ_i.

```
n = 1_000_000
p = 0.5
μ_1, μ_2, μ_3 = 0.2, 0.8, 0.4
σ_1, σ_2, σ_3 = 0.1, 0.05, 0.2
```

A routine using loops in python

Here's a routine using native Python loops to calculate the desired mean

$$\frac{1}{n}\sum_{i=1}^{n} S_i \approx \mathbb{E}S$$

```
%%time

S = 0.0
for i in range(n):
    X_1 = np.exp(μ_1 + σ_1 * randn())
    X_2 = np.exp(μ_2 + σ_2 * randn())
    X_3 = np.exp(μ_3 + σ_3 * randn())
    S += (X_1 + X_2 + X_3)**p
S / n
```

```
CPU times: user 1.96 s, sys: 7.71 ms, total: 1.97 s
Wall time: 1.87 s
```

```
2.2296568890358475
```

We can also construct a function that contains these operations:

```
def compute_mean(n=1_000_000):
    S = 0.0
    for i in range(n):
        X_1 = np.exp(μ_1 + σ_1 * randn())
        X_2 = np.exp(μ_2 + σ_2 * randn())
        X_3 = np.exp(μ_3 + σ_3 * randn())
        S += (X_1 + X_2 + X_3)**p
    return (S / n)
```

Now let's call it.

```
compute_mean()
```

```
2.22969720532873
```

20.2.3 A vectorized routine

If we want a more accurate estimate we should increase n.

But the code above runs quite slowly.

To make it faster, let's implement a vectorized routine using NumPy.

```python
def compute_mean_vectorized(n=1_000_000):
    X_1 = np.exp(μ_1 + σ_1 * randn(n))
    X_2 = np.exp(μ_2 + σ_2 * randn(n))
    X_3 = np.exp(μ_3 + σ_3 * randn(n))
    S = (X_1 + X_2 + X_3)**p
    return S.mean()
```

```python
%%time

compute_mean_vectorized()
```

```
CPU times: user 54.4 ms, sys: 5.64 ms, total: 60.1 ms
Wall time: 60.9 ms
```

```
2.2296599818220524
```

Notice that this routine is much faster.

We can increase n to get more accuracy and still have reasonable speed:

```python
%%time

compute_mean_vectorized(n=10_000_000)
```

```
CPU times: user 542 ms, sys: 50.9 ms, total: 593 ms
Wall time: 592 ms
```

```
2.2297332265797922
```

20.3 Pricing a European call option under risk neutrality

Next we are going to price a European call option under risk neutrality.

Let's first discuss risk neutrality and then consider European options.

20.3.1 Risk-neutral pricing

When we use risk-neutral pricing, we determine the price of a given asset according to its expected payoff:

$$\text{cost} = \text{expected benefit}$$

For example, suppose someone promises to pay you

- 1,000,000 dollars if "heads" is the outcome of a fair coin flip

- 0 dollars if "tails" is the outcome

Let's denote the payoff as G, so that

$$\mathbb{P}\left\{G = 10^6\right\} = \mathbb{P}\{G = 0\} = \frac{1}{2}$$

Suppose in addition that you can sell this promise to anyone who wants it.

- First they pay you P, the price at which you sell it

- Then they get G, which could be either 1,000,000 or 0.

What's a fair price for this asset (this promise)?

The definition of "fair" is ambiguous, but we can say that the **risk-neutral price** is 500,000 dollars.

This is because the risk-neutral price is just the expected payoff of the asset, which is

$$\mathbb{E}G = \frac{1}{2} \times 10^6 + \frac{1}{2} \times 0 = 5 \times 10^5$$

20.3.2 A comment on risk

As suggested by the name, the risk-neutral price ignores risk.

To understand this, consider whether you would pay 500,000 dollars for such a promise.

Would you prefer to receive 500,000 for sure or 1,000,000 dollars with 50% probability and nothing with 50% probability?

At least some readers will strictly prefer the first option —although some might prefer the second.

Thinking about this makes us realize that 500,000 is not necessarily the "right" price —or the price that we would see if there was a market for these promises.

Nonetheless, the risk-neutral price is an important benchmark, which economists and financial market participants try to calculate every day.

20.3.3 Discounting

Another thing we ignored in the previous discussion was time.

In general, receiving x dollars now is preferable to receiving x dollars in n periods (e.g., 10 years).

After all, if we receive x dollars now, we could put it in the bank at interest rate $r > 0$ and receive $(1+r)^n x$ in n periods.

Hence future payments need to be discounted when we consider their present value.

We will implement discounting by

- multiplying a payment in one period by $\beta < 1$

- multiplying a payment in n periods by β^n, etc.

The same adjustment needs to be applied to our risk-neutral price for the promise described above.

Thus, if G is realized in n periods, then the risk-neutral price is

$$P = \beta^n \mathbb{E} G = \beta^n 5 \times 10^5$$

20.3.4 European call options

Now let's price a European call option.

The option is described by three things:

2. n, the **expiry date**,

3. K, the **strike price**, and

4. S_n, the price of the **underlying** asset at date n.

For example, suppose that the underlying is one share in Amazon.

The owner of this option has the right to buy one share in Amazon at price K after n days.

If $S_n > K$, then the owner will exercise the option, buy at K, sell at S_n, and make profit $S_n - K$.

If $S_n \leq K$, then the owner will not exercise the option and the payoff is zero.

Thus, the payoff is $\max\{S_n - K, 0\}$.

Under the assumption of risk neutrality, the price of the option is the expected discounted payoff:

$$P = \beta^n \mathbb{E} \max\{S_n - K, 0\}$$

Now all we need to do is specify the distribution of S_n, so the expectation can be calculated.

Suppose we know that $S_n \sim LN(\mu, \sigma)$ and μ and σ are known.

If $S_n^1, \ldots, S_n^M$ are independent draws from this lognormal distribution then, by the law of large numbers,

$$\mathbb{E}\max\{S_n - K, 0\} \approx \frac{1}{M}\sum_{m=1}^{M}\max\{S_n^m - K, 0\}$$

We suppose that

```
μ = 1.0
σ = 0.1
K = 1
n = 10
β = 0.95
```

We set the simulation size to

```
M = 10_000_000
```

Here is our code

```
S = np.exp(μ + σ * np.random.randn(M))
return_draws = np.maximum(S - K, 0)
P = β**n * np.mean(return_draws)
print(f"The Monte Carlo option price is approximately {P:3f}")
```

```
The Monte Carlo option price is approximately 1.037013
```

20.4 Pricing via a dynamic model

In this exercise we investigate a more realistic model for the share price S_n.

This comes from specifying the underlying dynamics of the share price.

First we specify the dynamics.

Then we'll compute the price of the option using Monte Carlo.

20.4.1 Simple dynamics

One simple model for $\{S_t\}$ is

$$\ln \frac{S_{t+1}}{S_t} = \mu + \sigma \xi_{t+1}$$

where

- S_0 is lognormally distributed and

- $\{\xi_t\}$ is IID and standard normal.

Under the stated assumptions, S_n is lognormally distributed.

To see why, observe that, with $s_t := \ln S_t$, the price dynamics become

$$s_{t+1} = s_t + \mu + \sigma \xi_{t+1} \tag{20.1}$$

Since s_0 is normal and ξ_1 is normal and IID, we see that s_1 is normally distributed.

Continuing in this way shows that s_n is normally distributed.

Hence $S_n = \exp(s_n)$ is lognormal.

20.4.2 Problems with simple dynamics

The simple dynamic model we studied above is convenient, since we can work out the distribution of S_n.

However, its predictions are counterfactual because, in the real world, volatility (measured by σ) is not stationary.

Instead it rather changes over time, sometimes high (like during the GFC) and sometimes low.

In terms of our model above, this means that σ should not be constant.

20.4.3 More realistic dynamics

This leads us to study the improved version:

$$\ln \frac{S_{t+1}}{S_t} = \mu + \sigma_t \xi_{t+1}$$

where

$$\sigma_t = \exp(h_t), \quad h_{t+1} = \rho h_t + \nu \eta_{t+1}$$

Here $\{\eta_t\}$ is also IID and standard normal.

20.4.4 Default parameters

For the dynamic model, we adopt the following parameter values.

```
default_μ  = 0.0001
default_ρ  = 0.1
default_v  = 0.001
default_S0 = 10
default_h0 = 0
```

(Here `default_S0` is S_0 and `default_h0` is h_0.)

For the option we use the following defaults.

```
default_K = 100
default_n = 10
default_β = 0.95
```

20.4.5 Visualizations

With $s_t := \ln S_t$, the price dynamics become

$$s_{t+1} = s_t + \mu + \exp(h_t)\xi_{t+1}$$

Here is a function to simulate a path using this equation:

```python
def simulate_asset_price_path(μ=default_μ, S0=default_S0, h0=default_h0, n=default_n,
 ↪ρ=default_ρ, v=default_v):
    s = np.empty(n+1)
    s[0] = np.log(S0)

    h = h0
    for t in range(n):
        s[t+1] = s[t] + μ + np.exp(h) * randn()
        h = ρ * h + v * randn()

    return np.exp(s)
```

Here we plot the paths and the log of the paths.

```python
fig, axes = plt.subplots(2, 1)

titles = 'log paths', 'paths'
```

```python
transforms = np.log, lambda x: x
for ax, transform, title in zip(axes, transforms, titles):
    for i in range(50):
        path = simulate_asset_price_path()
        ax.plot(transform(path))
    ax.set_title(title)

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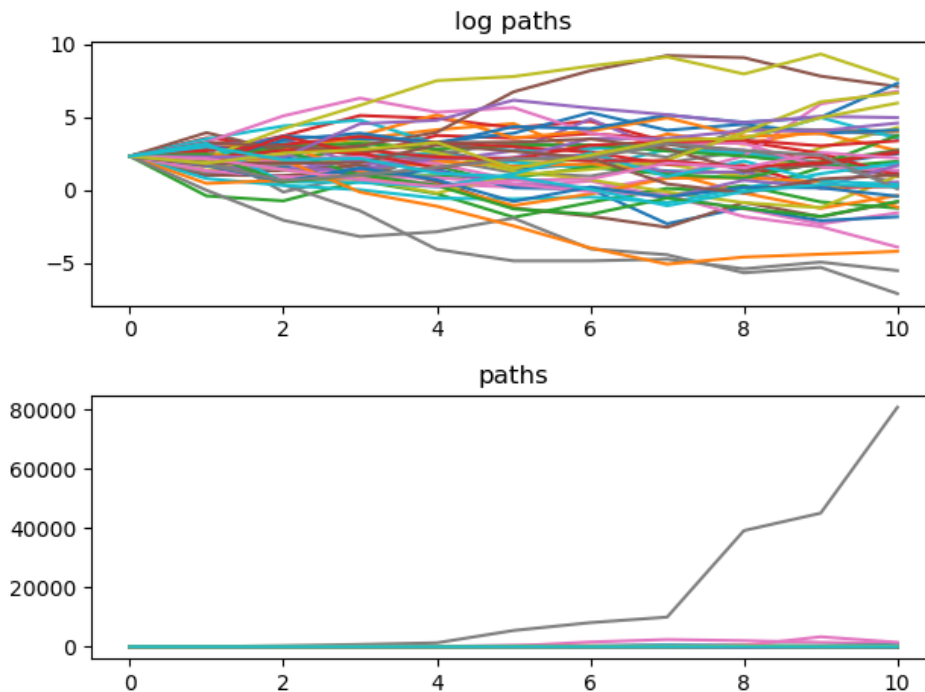

20.4.6 Computing the price

Now that our model is more complicated, we cannot easily determine the distribution of S_n.

So to compute the price P of the option, we use Monte Carlo.

We average over realizations $S_n^1, \ldots, S_n^M$ of S_n and appealing to the law of large numbers:

$$\mathbb{E}\max\{S_n - K, 0\} \approx \frac{1}{M}\sum_{m=1}^{M}\max\{S_n^m - K, 0\}$$

Here's a version using Python loops.

```python
def compute_call_price(β=default_β,
                       µ=default_µ,
                       S0=default_S0,
                       h0=default_h0,
                       K=default_K,
                       n=default_n,
                       ρ=default_ρ,
                       ν=default_ν,
                       M=10_000):
    current_sum = 0.0
    # For each sample path
    for m in range(M):
        s = np.log(S0)
        h = h0
        # Simulate forward in time
        for t in range(n):
            s = s + µ + np.exp(h) * randn()
            h = ρ * h + ν * randn()
        # And add the value max{S_n - K, 0} to current_sum
        current_sum += np.maximum(np.exp(s) - K, 0)

    return β**n * current_sum / M
```

```python
%%time
compute_call_price()
```

```
CPU times: user 308 ms, sys: 2.49 ms, total: 310 ms
Wall time: 104 ms
```

```
820.4218484959187
```

20.5 Exercises

Exercise 20.5.1

We would like to increase M in the code above to make the calculation more accurate.

But this is problematic because Python loops are slow.

Your task is to write a faster version of this code using NumPy.

Solution to Exercise 20.5.1

```python
def compute_call_price_vector(β=default_β,
                              μ=default_μ,
                              S0=default_S0,
                              h0=default_h0,
                              K=default_K,
                              n=default_n,
                              ρ=default_ρ,
                              ν=default_ν,
                              M=10_000):

    s = np.full(M, np.log(S0))
    h = np.full(M, h0)
    for t in range(n):
        Z = np.random.randn(2, M)
        s = s + μ + np.exp(h) * Z[0, :]
        h = ρ * h + ν * Z[1, :]
    expectation = np.mean(np.maximum(np.exp(s) - K, 0))

    return β**n * expectation
```

```python
%%time
compute_call_price_vector()
```

```
CPU times: user 8.25 ms, sys: 365 µs, total: 8.61 ms
Wall time: 3.77 ms
```

```
1064.3965434226907
```

Notice that this version is faster than the one using a Python loop.

Now let's try with larger M to get a more accurate calculation.

```python
%%time
compute_call_price(M=10_000_000)
```

```
CPU times: user 1min 42s, sys: 457 ms, total: 1min 42s
Wall time: 1min 43s
```

```
918.1790138714035
```

Exercise 20.5.2

Consider that a European call option may be written on an underlying with spot price of $100 and a knockout barrier of $120.

This option behaves in every way like a vanilla European call, except if the spot price ever moves above $120, the option "knocks out" and the contract is null and void.

Note that the option does not reactivate if the spot price falls below $120 again.

Use the dynamics defined in (20.1) to price the European call option.

Solution to Exercise 20.5.2

```
default_μ  = 0.0001
default_ρ  = 0.1
default_ν  = 0.001
default_S0 = 10
default_h0 = 0
default_K = 100
default_n = 10
default_β = 0.95
default_bp = 120
```

```
def compute_call_price_with_barrier(β=default_β,
                                    μ=default_μ,
                                    S0=default_S0,
                                    h0=default_h0,
                                    K=default_K,
                                    n=default_n,
                                    ρ=default_ρ,
                                    ν=default_ν,
                                    bp=default_bp,
                                    M=50_000):
    current_sum = 0.0
    # For each sample path
    for m in range(M):
        s = np.log(S0)
```

(continues on next page)

```
        h = h0
        payoff = 0
        option_is_null = False
        # Simulate forward in time
        for t in range(n):
            s = s + μ + np.exp(h) * randn()
            h = ρ * h + ν * randn()
            if np.exp(s) > bp:
                payoff = 0
                option_is_null = True
                break

        if not option_is_null:
            payoff = np.maximum(np.exp(s) - K, 0)
        # And add the payoff to current_sum
        current_sum += payoff

    return β**n * current_sum / M
```

```
%time compute_call_price_with_barrier()
```

```
CPU times: user 596 ms, sys: 3.15 ms, total: 600 ms
Wall time: 601 ms
```

```
0.03798374019879899
```

Let's look at the vectorized version which is faster than using Python loops.

```
def compute_call_price_with_barrier_vector(β=default_β,
                                            μ=default_μ,
                                            S0=default_S0,
                                            h0=default_h0,
                                            K=default_K,
                                            n=default_n,
                                            ρ=default_ρ,
                                            ν=default_ν,
                                            bp=default_bp,
                                            M=50_000):
    s = np.full(M, np.log(S0))
    h = np.full(M, h0)
    option_is_null = np.full(M, False)
```

```python
    for t in range(n):
        Z = np.random.randn(2, M)
        s = s + μ + np.exp(h) * Z[0, :]
        h = ρ * h + ν * Z[1, :]
        # Mark all the options null where S_n > barrier price
        option_is_null = np.where(np.exp(s) > bp, True, option_is_null)

    # mark payoff as 0 in the indices where options are null
    payoff = np.where(option_is_null, 0, np.maximum(np.exp(s) - K, 0))
    expectation = np.mean(payoff)
    return β**n * expectation
```

```python
%time compute_call_price_with_barrier_vector()
```

```
CPU times: user 21.2 ms, sys: 1.5 ms, total: 22.7 ms
Wall time: 22.9 ms
```

```
0.03487550480503958
```

HEAVY-TAILED DISTRIBUTIONS

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

```
!pip install --upgrade yfinance pandas_datareader
```

We use the following imports.

```python
import matplotlib.pyplot as plt
import numpy as np
import yfinance as yf
import pandas as pd
import statsmodels.api as sm


from pandas_datareader import wb
from scipy.stats import norm, cauchy
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
```

21.1 Overview

Heavy-tailed distributions are a class of distributions that generate "extreme" outcomes.

In the natural sciences (and in more traditional economics courses), heavy-tailed distributions are seen as quite exotic and non-standard.

However, it turns out that heavy-tailed distributions play a crucial role in economics.

In fact many – if not most – of the important distributions in economics are heavy-tailed.

In this lecture we explain what heavy tails are and why they are – or at least why they should be – central to economic analysis.

21.1.1 Introduction: light tails

Most *commonly used probability distributions* in classical statistics and the natural sciences have "light tails."

To explain this concept, let's look first at examples.

Example 21.1.1

The classic example is the normal distribution, which has density

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \qquad (-\infty < x < \infty)$$

The two parameters μ and σ are the mean and standard deviation respectively.

As x deviates from μ, the value of $f(x)$ goes to zero extremely quickly.

We can see this when we plot the density and show a histogram of observations, as with the following code (which assumes $\mu = 0$ and $\sigma = 1$).

```
fig, ax = plt.subplots()
X = norm.rvs(size=1_000_000)
ax.hist(X, bins=40, alpha=0.4, label='histogram', density=True)
x_grid = np.linspace(-4, 4, 400)
ax.plot(x_grid, norm.pdf(x_grid), label='density')
ax.legend()
plt.show()
```

Notice how

- the density's tails converge quickly to zero in both directions and

- even with 1,000,000 draws, we get no very large or very small observations.

We can see the last point more clearly by executing

```
X.min(), X.max()
```

```
(-4.897207259233048, 4.940638539225094)
```

Here's another view of draws from the same distribution:
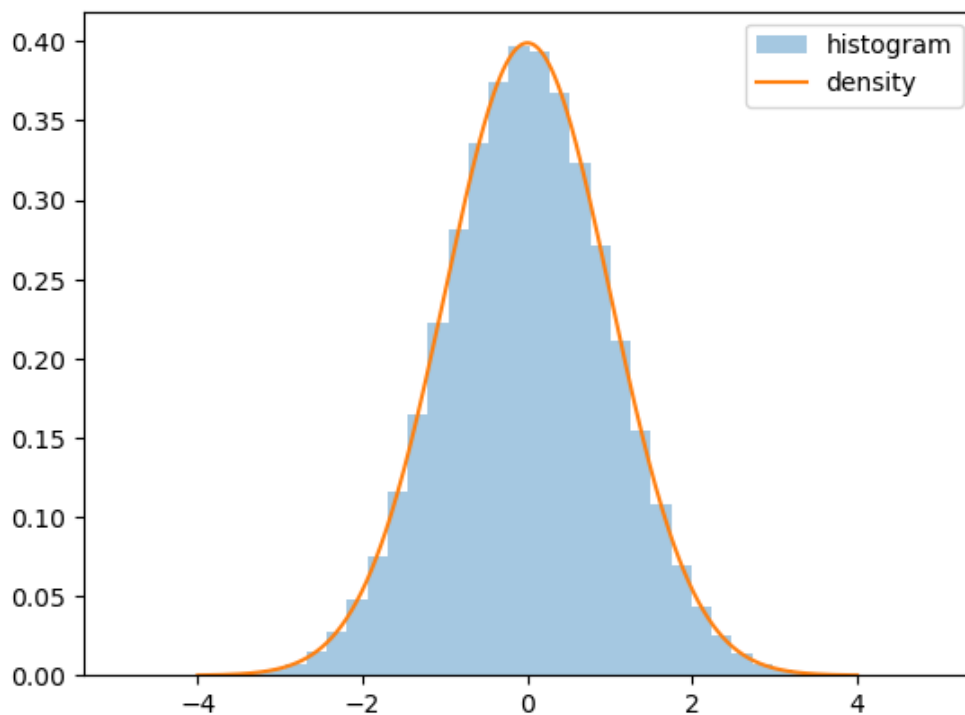
Fig. 21.1: Histogram of observations

```
n = 2000
fig, ax = plt.subplots()
data = norm.rvs(size=n)
ax.plot(list(range(n)), data, linestyle='', marker='o', alpha=0.5, ms=4)
ax.vlines(list(range(n)), 0, data, lw=0.2)
ax.set_ylim(-15, 15)
ax.set_xlabel('$i$')
ax.set_ylabel('$X_i$', rotation=0)
plt.show()
```

We have plotted each individual draw X_i against i.

None are very large or very small.

In other words, extreme observations are rare and draws tend not to deviate too much from the mean.

Putting this another way, light-tailed distributions are those that rarely generate extreme values.

(A more formal definition is given *below*.)

Many statisticians and econometricians use rules of thumb such as "outcomes more than four or five standard deviations from the mean can safely be ignored."
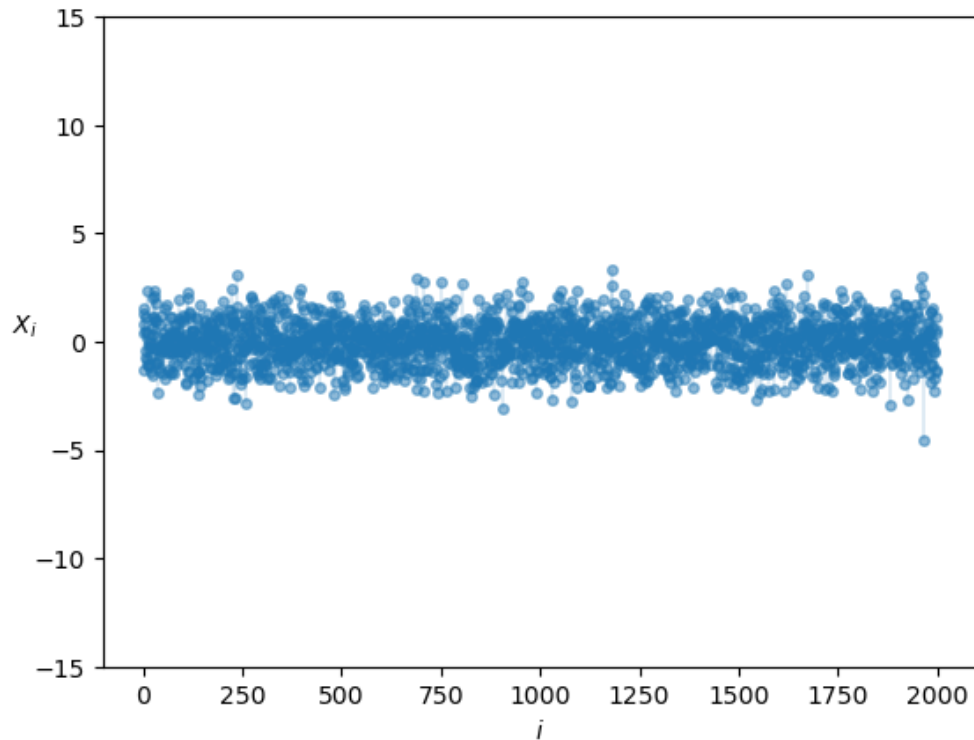
Fig. 21.2: Histogram of observations

But this is only true when distributions have light tails.

21.1.2 When are light tails valid?

In probability theory and in the real world, many distributions are light-tailed.

For example, human height is light-tailed.

Yes, it's true that we see some very tall people.

- For example, basketballer Sun Mingming is 2.32 meters tall

But have you ever heard of someone who is 20 meters tall? Or 200? Or 2000?

Have you ever wondered why not?

After all, there are 8 billion people in the world!

In essence, the reason we don't see such draws is that the distribution of human height has very light tails.

In fact the distribution of human height obeys a bell-shaped curve similar to the normal distribution.

21.1.3 Returns on assets

But what about economic data?

Let's look at some financial data first.

Our aim is to plot the daily change in the price of Amazon (AMZN) stock for the period from 1st January 2015 to 1st July 2022.

This equates to daily returns if we set dividends aside.

The code below produces the desired plot using Yahoo financial data via the `yfinance` library.

```
data = yf.download('AMZN', '2015-1-1', '2022-7-1')
```

```
s = data['Adj Close']
r = s.pct_change()

fig, ax = plt.subplots()

ax.plot(r, linestyle='', marker='o', alpha=0.5, ms=4)
ax.vlines(r.index, 0, r.values, lw=0.2)
ax.set_ylabel('returns', fontsize=12)
ax.set_xlabel('date', fontsize=12)

plt.show()
```

This data looks different to the draws from the normal distribution we saw above.

Several of observations are quite extreme.

We get a similar picture if we look at other assets, such as Bitcoin

```
data = yf.download('BTC-USD', '2015-1-1', '2022-7-1')
```

```
s = data['Adj Close']
r = s.pct_change()

fig, ax = plt.subplots()

ax.plot(r, linestyle='', marker='o', alpha=0.5, ms=4)
ax.vlines(r.index, 0, r.values, lw=0.2)
ax.set_ylabel('returns', fontsize=12)
```

(continues on next page)
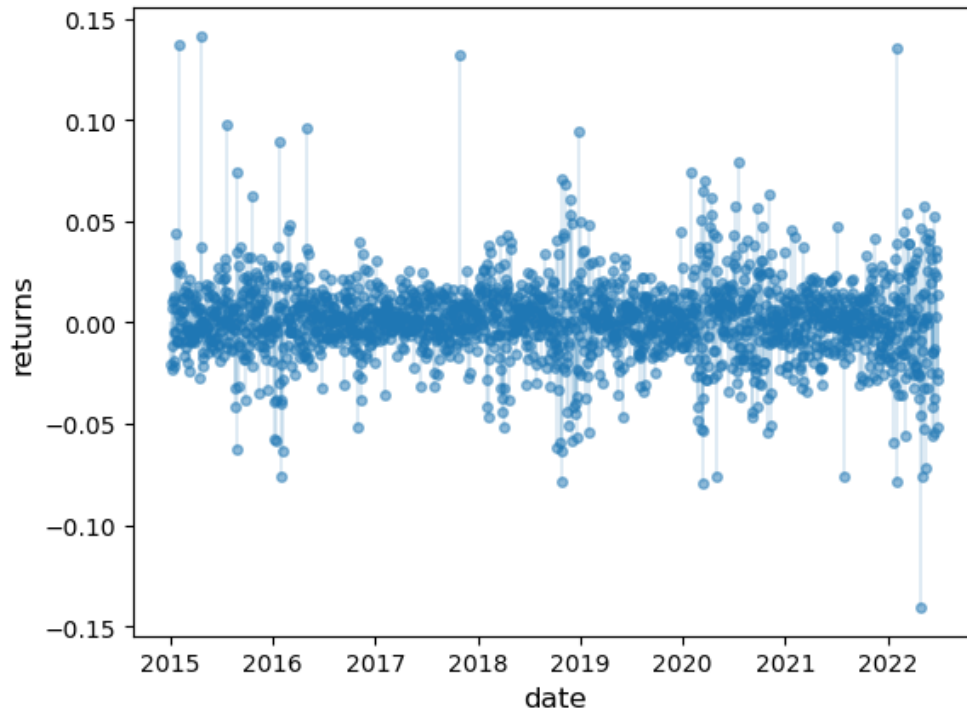
Fig. 21.3: Daily Amazon returns

```
ax.set_xlabel('date', fontsize=12)

plt.show()
```

The histogram also looks different to the histogram of the normal distribution:

```
r = np.random.standard_t(df=5, size=1000)

fig, ax = plt.subplots()
ax.hist(r, bins=60, alpha=0.4, label='bitcoin returns', density=True)

xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, 100)
p = norm.pdf(x, np.mean(r), np.std(r))
ax.plot(x, p, linewidth=2, label='normal distribution')

ax.set_xlabel('returns', fontsize=12)
ax.legend()
```
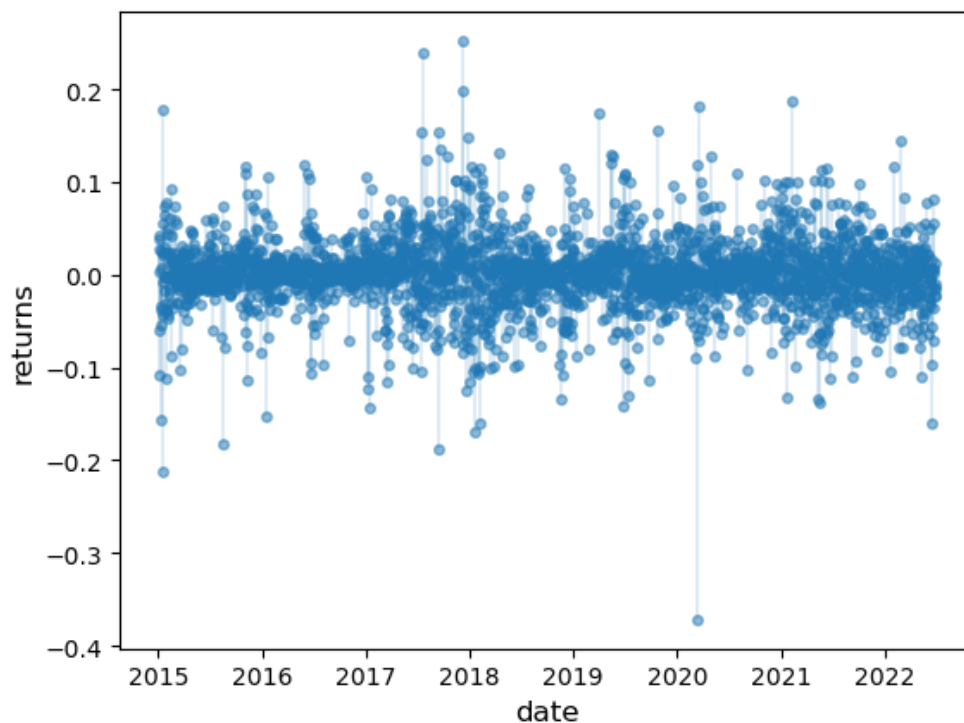
Fig. 21.4: Daily Bitcoin returns

```
plt.show()
```

If we look at higher frequency returns data (e.g., tick-by-tick), we often see even more extreme observations.

See, for example, [Mandelbrot, 1963] or [Rachev, 2003].

21.1.4 Other data

The data we have just seen is said to be "heavy-tailed".

With heavy-tailed distributions, extreme outcomes occur relatively frequently.

Example 21.1.2

Importantly, there are many examples of heavy-tailed distributions observed in economic and financial settings!

For example, the income and the wealth distributions are heavy-tailed

- You can imagine this: most people have low or modest wealth but some people are extremely rich.
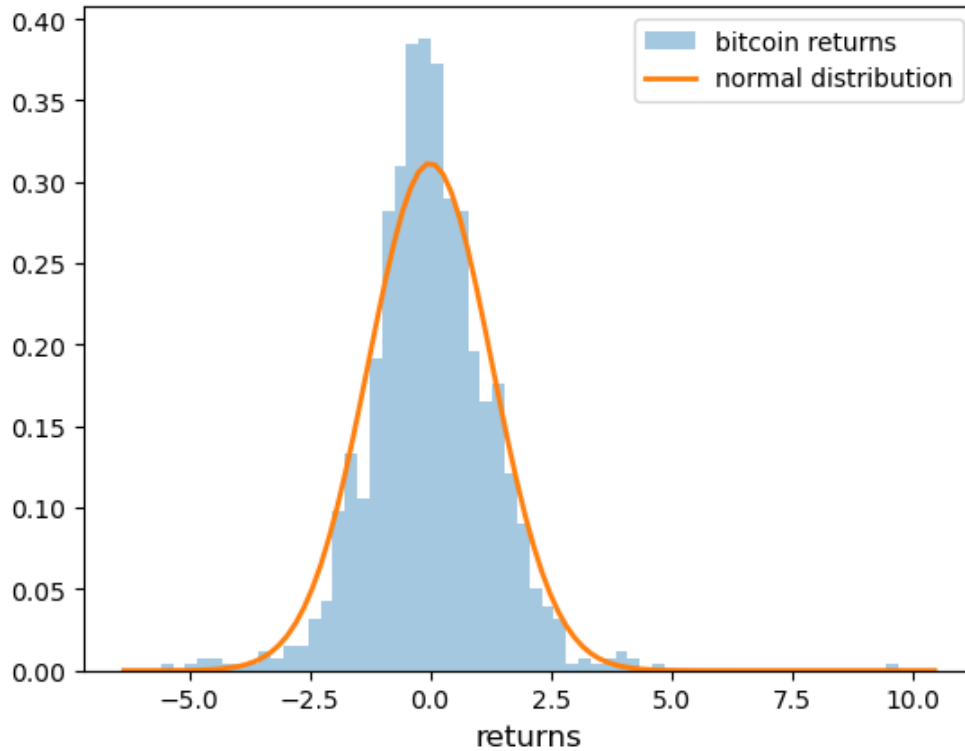
Fig. 21.5: Histogram (normal vs bitcoin returns)

The firm size distribution is also heavy-tailed

- You can imagine this too: most firms are small but some firms are enormous.

The distribution of town and city sizes is heavy-tailed

- Most towns and cities are small but some are very large.

Later in this lecture, we examine heavy tails in these distributions.

21.1.5 Why should we care?

Heavy tails are common in economic data but does that mean they are important?

The answer to this question is affirmative!

When distributions are heavy-tailed, we need to think carefully about issues like

- diversification and risk

- forecasting

- taxation (across a heavy-tailed income distribution), etc.

We return to these points *below*.

21.2 Visual comparisons

In this section, we will introduce important concepts such as the Pareto distribution, Counter CDFs, and Power laws, which aid in recognizing heavy-tailed distributions.

Later we will provide a mathematical definition of the difference between light and heavy tails.

But for now let's do some visual comparisons to help us build intuition on the difference between these two types of distributions.

21.2.1 Simulations

The figure below shows a simulation.

The top two subfigures each show 120 independent draws from the normal distribution, which is light-tailed.

The bottom subfigure shows 120 independent draws from the Cauchy distribution, which is heavy-tailed.

```
n = 120
np.random.seed(11)

fig, axes = plt.subplots(3, 1, figsize=(6, 12))

for ax in axes:
    ax.set_ylim((-120, 120))

s_vals = 2, 12

for ax, s in zip(axes[:2], s_vals):
    data = np.random.randn(n) * s
    ax.plot(list(range(n)), data, linestyle='', marker='o', alpha=0.5, ms=4)
    ax.vlines(list(range(n)), 0, data, lw=0.2)
    ax.set_title(f"draws from $N(0, \sigma^2)$ with $\sigma = {s}$", fontsize=11)

ax = axes[2]
distribution = cauchy()
data = distribution.rvs(n)
ax.plot(list(range(n)), data, linestyle='', marker='o', alpha=0.5, ms=4)
ax.vlines(list(range(n)), 0, data, lw=0.2)
ax.set_title(f"draws from the Cauchy distribution", fontsize=11)
```

```
plt.subplots_adjust(hspace=0.25)


plt.show()
```

In the top subfigure, the standard deviation of the normal distribution is 2, and the draws are clustered around the mean.

In the middle subfigure, the standard deviation is increased to 12 and, as expected, the amount of dispersion rises.

The bottom subfigure, with the Cauchy draws, shows a different pattern: tight clustering around the mean for the great majority of observations, combined with a few sudden large deviations from the mean.

This is typical of a heavy-tailed distribution.

21.2.2 Nonnegative distributions

Let's compare some distributions that only take nonnegative values.

One is the exponential distribution, which we discussed in *our lecture on probability and distributions*.

The exponential distribution is a light-tailed distribution.

Here are some draws from the exponential distribution.

```
n = 120
np.random.seed(11)


fig, ax = plt.subplots()
ax.set_ylim((0, 50))

data = np.random.exponential(size=n)
ax.plot(list(range(n)), data, linestyle='', marker='o', alpha=0.5, ms=4)
ax.vlines(list(range(n)), 0, data, lw=0.2)


plt.show()
```

Another nonnegative distribution is the Pareto distribution.

If X has the Pareto distribution, then there are positive constants $\bar{x}$ and α such that

$$\mathbb{P}\{X > x\} = \begin{cases} (\bar{x}/x)^\alpha & \text{if } x \geq \bar{x} \\ 1 & \text{if } x < \bar{x} \end{cases} \tag{21.1}$$
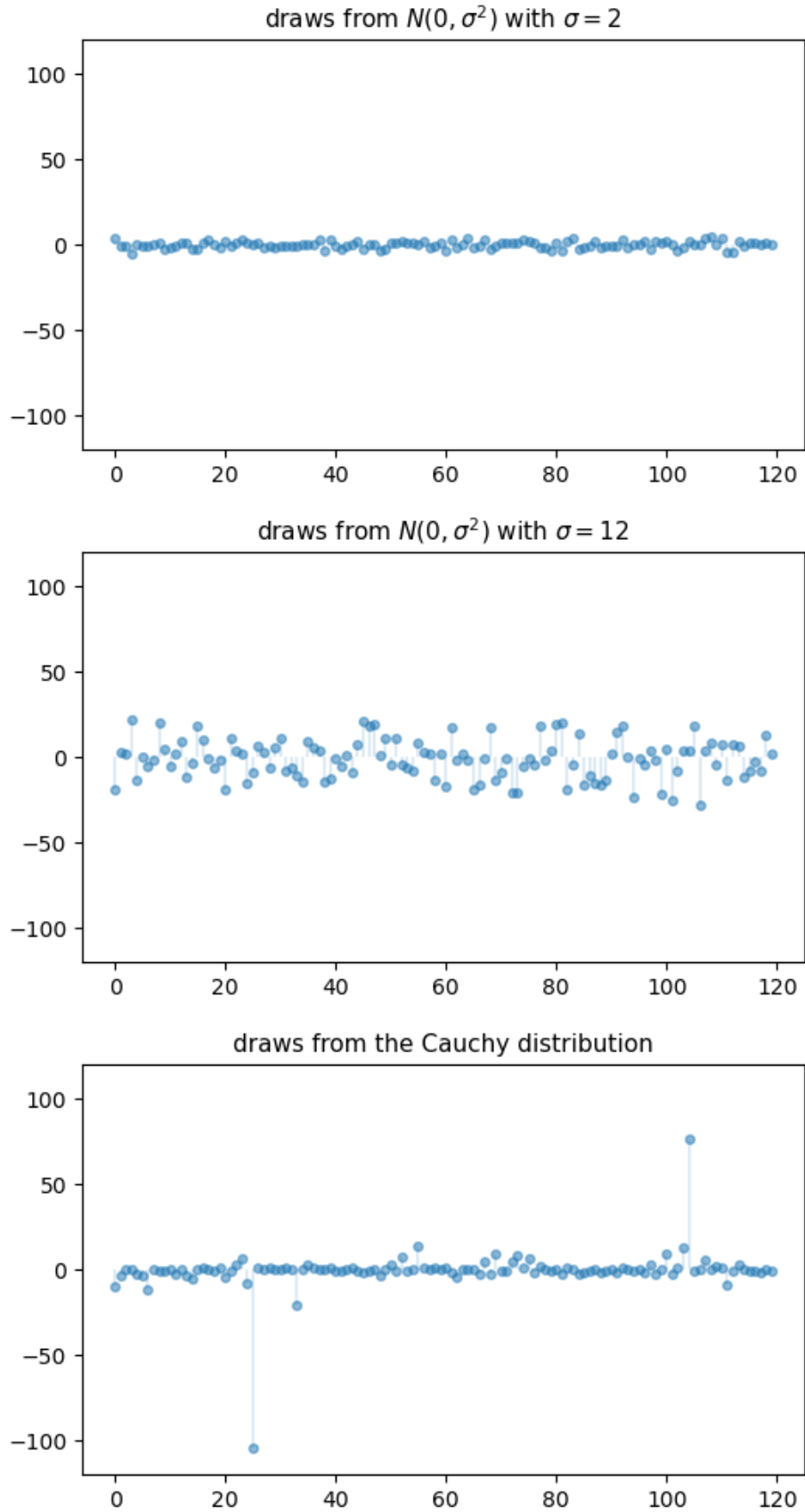
Fig. 21.6: Draws from normal and Cauchy distributions
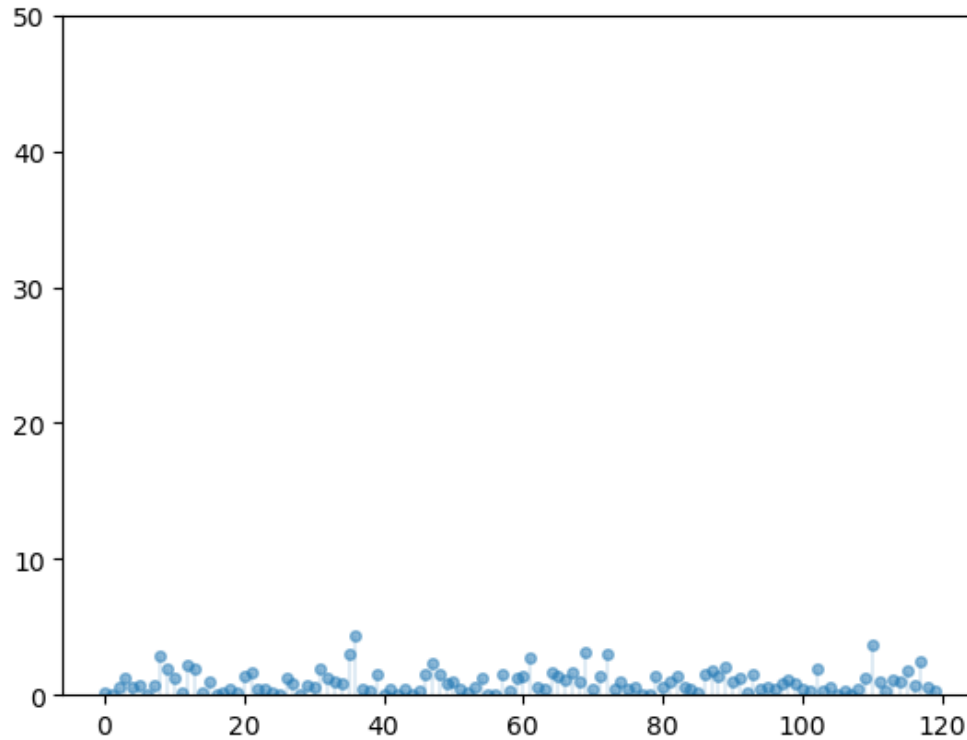
Fig. 21.7: Draws of exponential distribution

The parameter α is called the **tail index** and $\bar{x}$ is called the **minimum**.

The Pareto distribution is a heavy-tailed distribution.

One way that the Pareto distribution arises is as the exponential of an exponential random variable.

In particular, if X is exponentially distributed with rate parameter α, then

$$Y = \bar{x} \exp(X)$$

is Pareto-distributed with minimum $\bar{x}$ and tail index α.

Here are some draws from the Pareto distribution with tail index 1 and minimum 1.

```
n = 120
np.random.seed(11)

fig, ax = plt.subplots()
ax.set_ylim((0, 80))
exponential_data = np.random.exponential(size=n)
pareto_data = np.exp(exponential_data)
ax.plot(list(range(n)), pareto_data, linestyle='', marker='o', alpha=0.5, ms=4)
ax.vlines(list(range(n)), 0, pareto_data, lw=0.2)
```

```
plt.show()
```
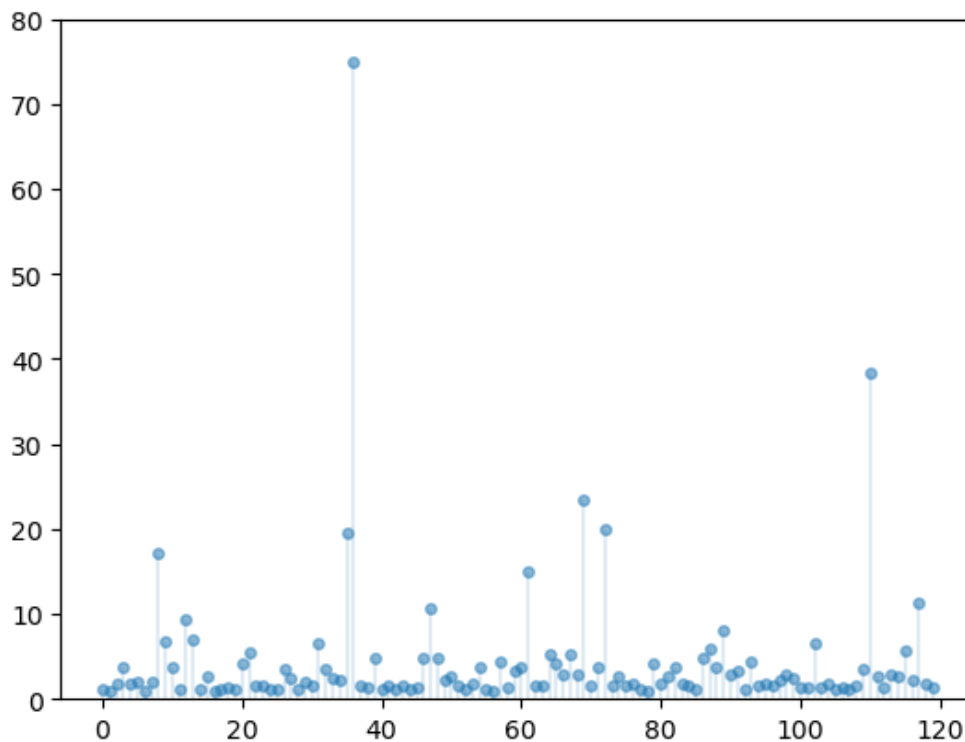

Fig. 21.8: Draws from Pareto distribution

Notice how extreme outcomes are more common.

21.2.3 Counter CDFs

For nonnegative random variables, one way to visualize the difference between light and heavy tails is to look at the **counter CDF** (CCDF).

For a random variable X with CDF F, the CCDF is the function

$$G(x) := 1 - F(x) = \mathbb{P}\{X > x\}$$

(Some authors call G the "survival" function.)

The CCDF shows how fast the upper tail goes to zero as $x \to \infty$.

If X is exponentially distributed with rate parameter α, then the CCDF is

$$G_E(x) = \exp(-\alpha x)$$

This function goes to zero relatively quickly as x gets large.

The standard Pareto distribution, where $\bar{x} = 1$, has CCDF

$$G_P(x) = x^{-\alpha}$$

This function goes to zero as $x \to \infty$, but much slower than G_E.

Exercise 21.2.1

Show how the CCDF of the standard Pareto distribution can be derived from the CCDF of the exponential distribution.

Solution to Exercise 21.2.1

Letting G_E and G_P be defined as above, letting X be exponentially distributed with rate parameter α, and letting $Y = \exp(X)$, we have

$$
\begin{aligned}
G_P(y) &= \mathbb{P}\{Y > y\} \\
&= \mathbb{P}\{\exp(X) > y\} \\
&= \mathbb{P}\{X > \ln y\} \\
&= G_E(\ln y) \\
&= \exp(-\alpha \ln y) \\
&= y^{-\alpha}
\end{aligned}
$$

Here's a plot that illustrates how G_E goes to zero faster than G_P.

```
x = np.linspace(1.5, 100, 1000)
fig, ax = plt.subplots()
alpha = 1.0
ax.plot(x, np.exp(- alpha * x), label='exponential', alpha=0.8)
ax.plot(x, x**(- alpha), label='Pareto', alpha=0.8)
ax.set_xlabel('X value')
ax.set_ylabel('CCDF')
ax.legend()
plt.show()
```

Here's a log-log plot of the same functions, which makes visual comparison easier.
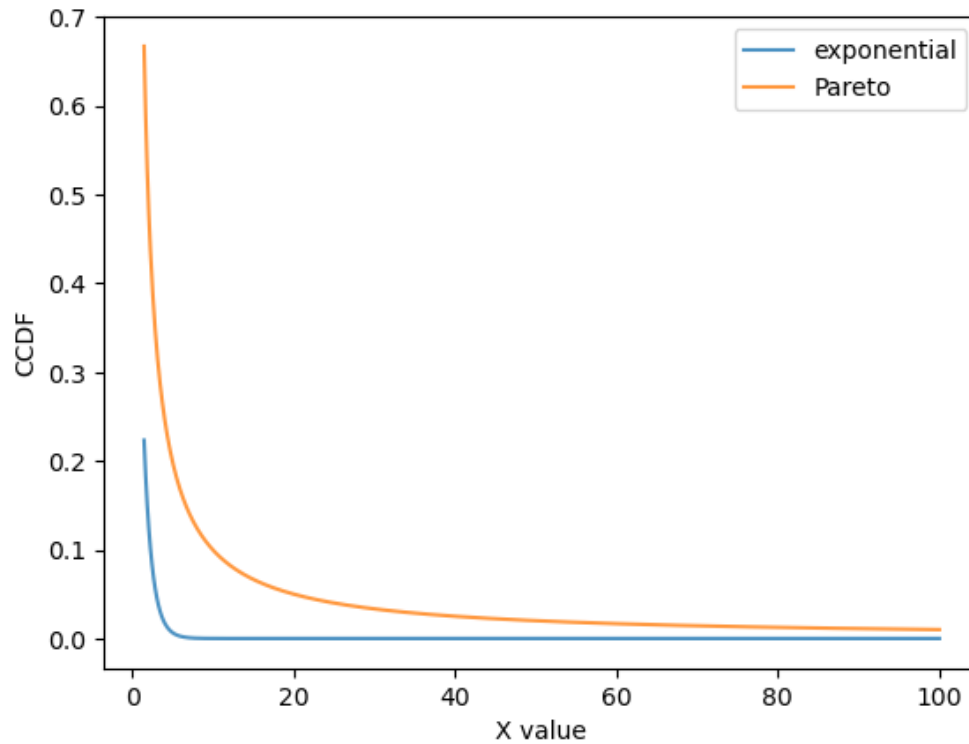
Fig. 21.9: Pareto and exponential distribution comparison

```
fig, ax = plt.subplots()
alpha = 1.0
ax.loglog(x, np.exp(- alpha * x), label='exponential', alpha=0.8)
ax.loglog(x, x**(- alpha), label='Pareto', alpha=0.8)
ax.set_xlabel('log value')
ax.set_ylabel('log prob')
ax.legend()
plt.show()
```

In the log-log plot, the Pareto CCDF is linear, while the exponential one is concave.

This idea is often used to separate light- and heavy-tailed distributions in visualisations —we return to this point below.
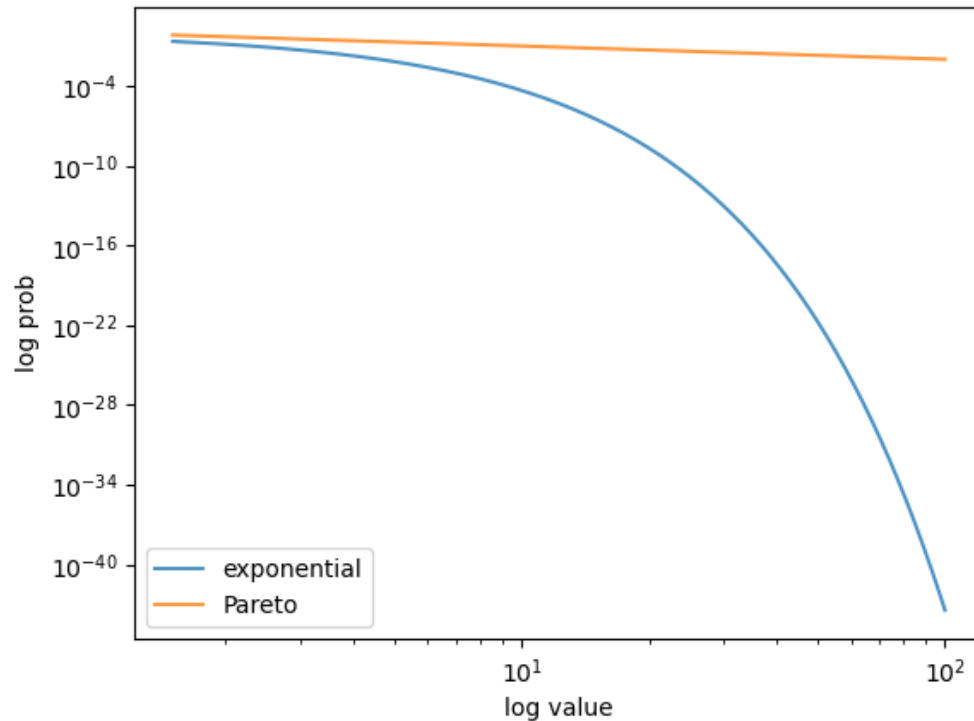
Fig. 21.10: Pareto and exponential distribution comparison (log-log)

21.2.4 Empirical CCDFs

The sample counterpart of the CCDF function is the **empirical CCDF**.

Given a sample $x_1, \dots, x_n$, the empirical CCDF is given by

$$\hat{G}(x) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}\{x_i > x\}$$

Thus, $\hat{G}(x)$ shows the fraction of the sample that exceeds x.

```python
def eccdf(x, data):
    "Simple empirical CCDF function."
    return np.mean(data > x)
```

Here's a figure containing some empirical CCDFs from simulated data.

```python
# Parameters and grid
x_grid = np.linspace(1, 1000, 1000)
sample_size = 1000
np.random.seed(13)
z = np.random.randn(sample_size)
```

```python
# Draws
data_exp = np.random.exponential(size=sample_size)
data_logn = np.exp(z)
data_pareto = np.exp(np.random.exponential(size=sample_size))


data_list = [data_exp, data_logn, data_pareto]


# Build figure
fig, axes = plt.subplots(3, 1, figsize=(6, 8))
axes = axes.flatten()
labels = ['exponential', 'lognormal', 'Pareto']


for data, label, ax in zip(data_list, labels, axes):

    ax.loglog(x_grid, [eccdf(x, data) for x in x_grid],
        'o', markersize=3.0, alpha=0.5, label=label)
    ax.set_xlabel("log value")
    ax.set_ylabel("log prob")


    ax.legend()



fig.subplots_adjust(hspace=0.4)


plt.show()
```

As with the CCDF, the empirical CCDF from the Pareto distributions is approximately linear in a log-log plot.

We will use this idea below when we look at real data.

Q-Q Plots

We can also use a qq plot to do a visual comparison between two probability distributions.

The statsmodels package provides a convenient qqplot function that, by default, compares sample data to the quintiles of the normal distribution.

If the data is drawn from a normal distribution, the plot would look like:
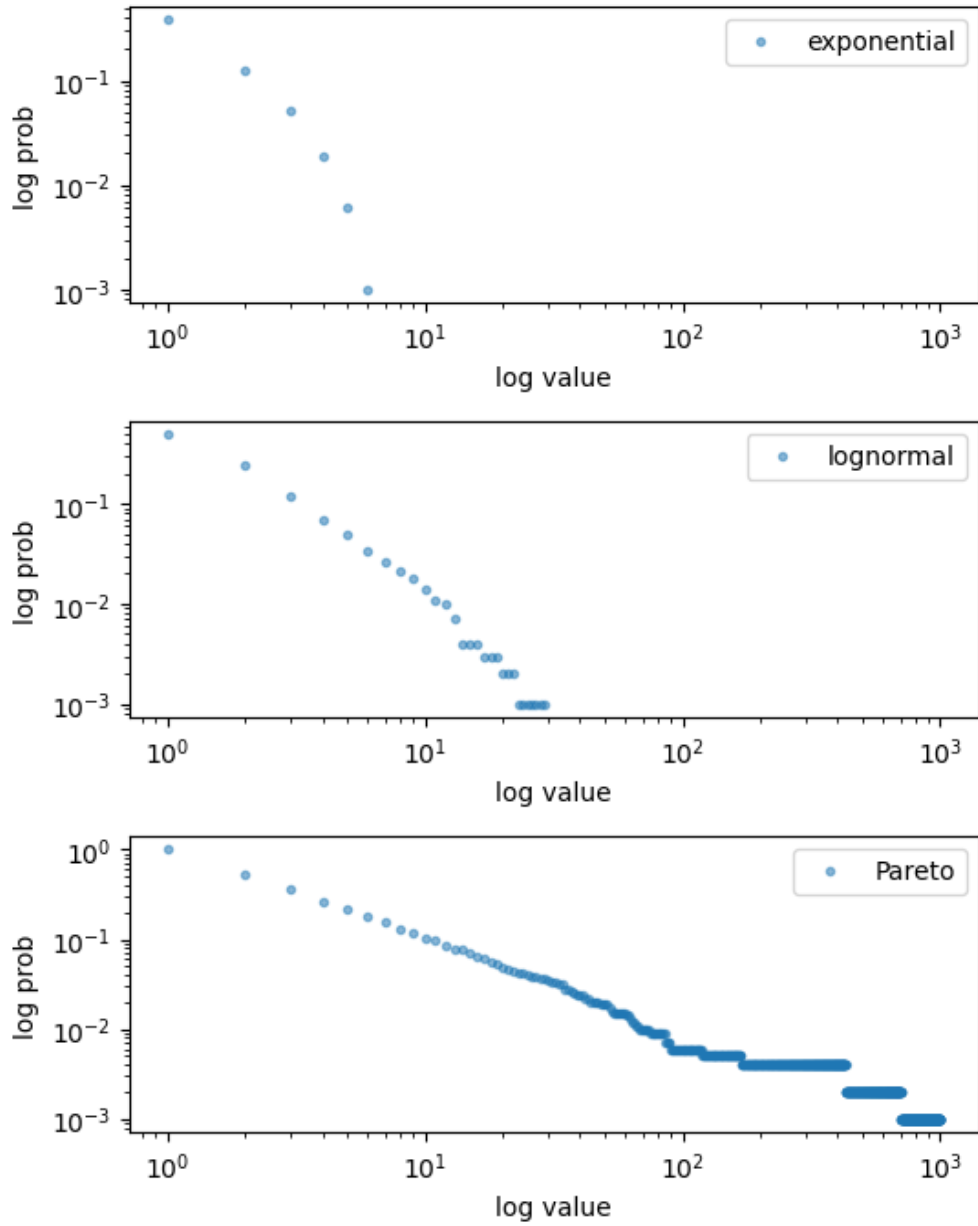
Fig. 21.11: Empirical CCDFs

```
data_normal = np.random.normal(size=sample_size)
sm.qqplot(data_normal, line='45')
plt.show()
```
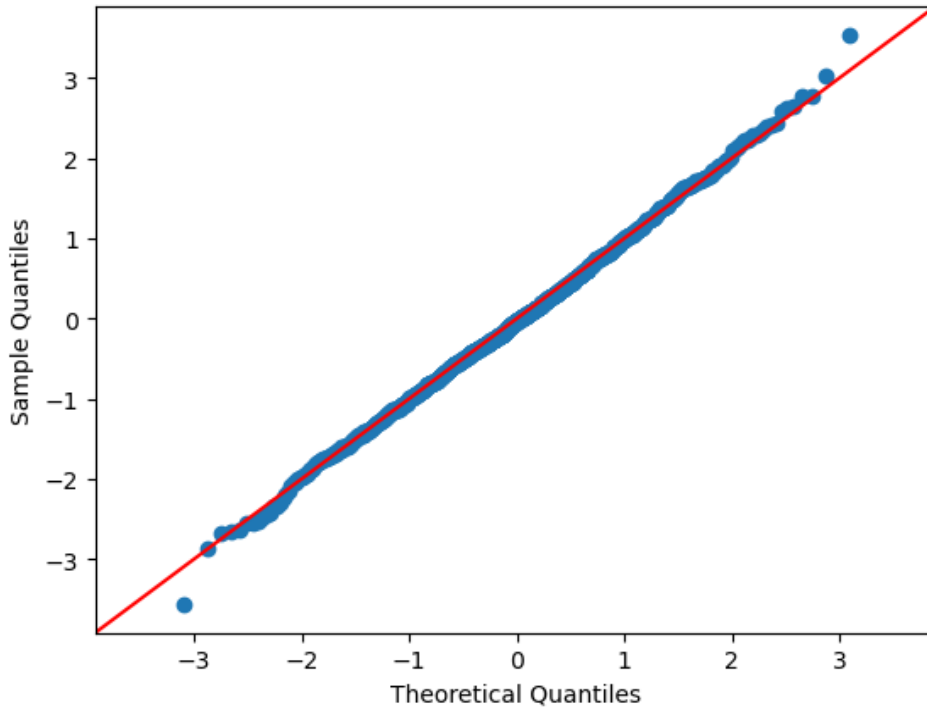

We can now compare this with the exponential, log-normal, and Pareto distributions

```
# Build figure
fig, axes = plt.subplots(1, 3, figsize=(12, 4))
axes = axes.flatten()
labels = ['exponential', 'lognormal', 'Pareto']
for data, label, ax in zip(data_list, labels, axes):
    sm.qqplot(data, line='45', ax=ax, )
    ax.set_title(label)
plt.tight_layout()
plt.show()
```
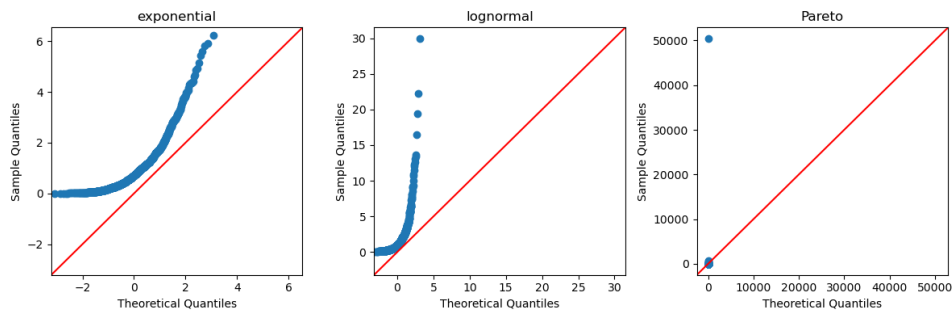
21.2.5 Power laws

One specific class of heavy-tailed distributions has been found repeatedly in economic and social phenomena: the class of so-called power laws.

A random variable X is said to have a **power law** if, for some $\alpha > 0$,

$$\mathbb{P}\{X > x\} \approx x^{-\alpha} \quad \text{when } x \text{ is large}$$

We can write this more mathematically as

$$\lim_{x \to \infty} x^{\alpha} \, \mathbb{P}\{X > x\} = c \quad \text{for some } c > 0 \tag{21.2}$$

It is also common to say that a random variable X with this property has a **Pareto tail** with **tail index** α.

Notice that every Pareto distribution with tail index α has a **Pareto tail** with **tail index** α.

We can think of power laws as a generalization of Pareto distributions.

They are distributions that resemble Pareto distributions in their upper right tail.

Another way to think of power laws is a set of distributions with a specific kind of (very) heavy tail.

21.3 Heavy tails in economic cross-sections

As mentioned above, heavy tails are pervasive in economic data.

In fact power laws seem to be very common as well.

We now illustrate this by showing the empirical CCDF of heavy tails.

All plots are in log-log, so that a power law shows up as a linear log-log plot, at least in the upper tail.

We hide the code that generates the figures, which is somewhat complex, but readers are of course welcome to explore the code (perhaps after examining the figures).

21.3.1 Firm size

Here is a plot of the firm size distribution for the largest 500 firms in 2020 taken from Forbes Global 2000.
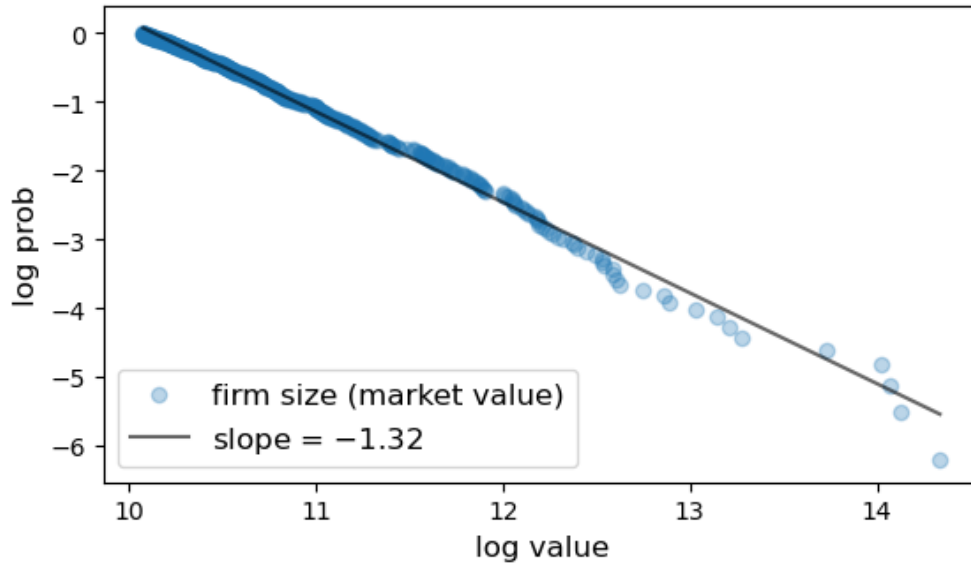
Fig. 21.12: Firm size distribution

21.3.2 City size

Here are plots of the city size distribution for the US and Brazil in 2023 from the World Population Review.

The size is measured by population.
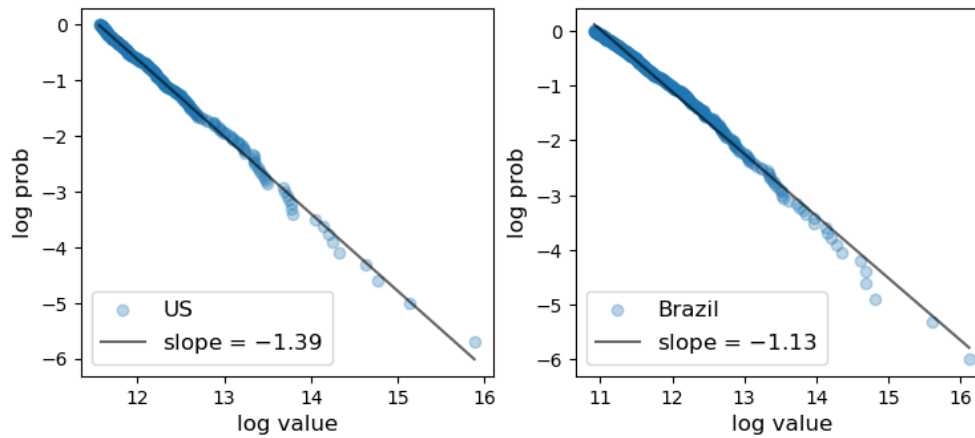
Fig. 21.13: City size distribution

21.3.3 Wealth

Here is a plot of the upper tail (top 500) of the wealth distribution.

The data is from the Forbes Billionaires list in 2020.
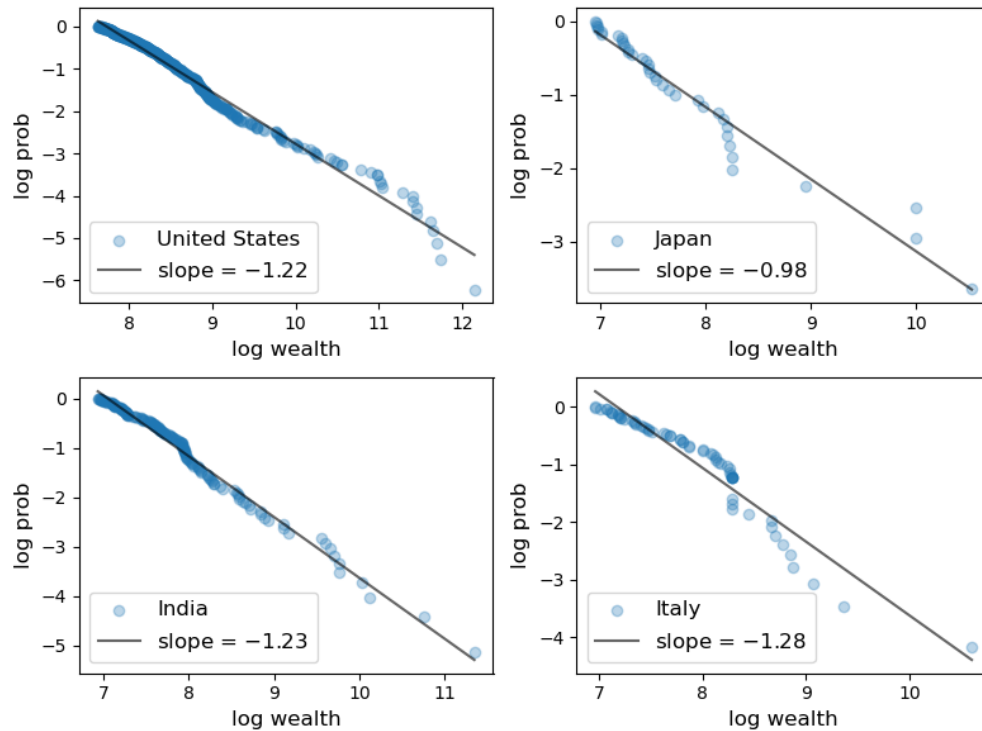
Fig. 21.14: Wealth distribution (Forbes billionaires in 2020)

21.3.4 GDP

Of course, not all cross-sectional distributions are heavy-tailed.

Here we show cross-country per capita GDP.

The plot is concave rather than linear, so the distribution has light tails.

One reason is that this is data on an aggregate variable, which involves some averaging in its definition.

Averaging tends to eliminate extreme outcomes.
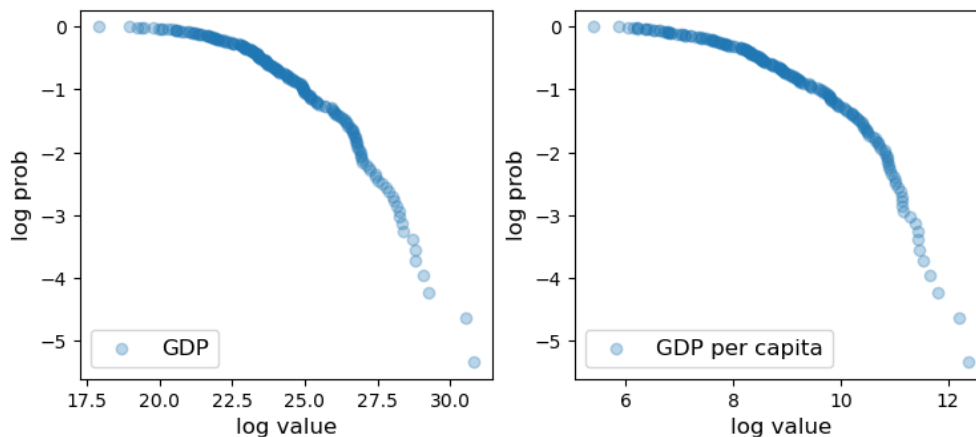
Fig. 21.15: GDP per capita distribution

21.4 Failure of the LLN

One impact of heavy tails is that sample averages can be poor estimators of the underlying mean of the distribution.

To understand this point better, recall *our earlier discussion* of the law of large numbers, which considered IID $X_1, \dots, X_n$ with common distribution F

If $\mathbb{E}|X_i|$ is finite, then the sample mean $\bar{X}_n := \frac{1}{n} \sum_{i=1}^{n} X_i$ satisfies

$$\mathbb{P}\left\{\bar{X}_n \to \mu \text{ as } n \to \infty\right\} = 1 \tag{21.3}$$

where $\mu := \mathbb{E}X_i = \int x F(dx)$ is the common mean of the sample.

The condition $\mathbb{E}|X_i| = \int |x| F(dx) < \infty$ holds in most cases but can fail if the distribution F is very heavy-tailed.

For example, it fails for the Cauchy distribution.

Let's have a look at the behavior of the sample mean in this case, and see whether or not the LLN is still valid.

```python
from scipy.stats import cauchy

np.random.seed(1234)
N = 1_000

distribution = cauchy()

fig, ax = plt.subplots()
data = distribution.rvs(N)

# Compute sample mean at each n
sample_mean = np.empty(N)
```

(continues on next page)

```
for n in range(1, N):
    sample_mean[n] = np.mean(data[:n])

# Plot
ax.plot(range(N), sample_mean, alpha=0.6, label='$\\bar{X}_n$')
ax.plot(range(N), np.zeros(N), 'k--', lw=0.5)
ax.set_xlabel(r"$n$")
ax.legend()

plt.show()
```
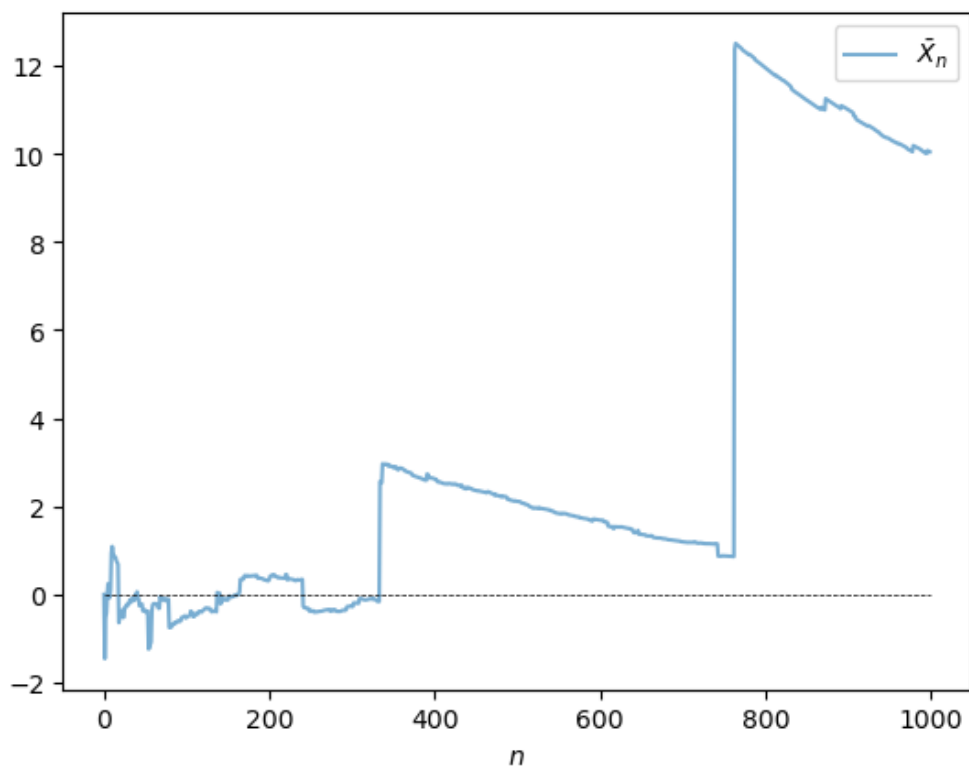

Fig. 21.16: LLN failure

The sequence shows no sign of converging.

We return to this point in the exercises.

21.5 Why do heavy tails matter?

We have now seen that

1. heavy tails are frequent in economics and

2. the law of large numbers fails when tails are very heavy.

But what about in the real world? Do heavy tails matter?

Let's briefly discuss why they do.

21.5.1 Diversification

One of the most important ideas in investing is using diversification to reduce risk.

This is a very old idea —consider, for example, the expression "don't put all your eggs in one basket".

To illustrate, consider an investor with one dollar of wealth and a choice over n assets with payoffs $X_1, \ldots, X_n$.

Suppose that returns on distinct assets are independent and each return has mean μ and variance σ^2.

If the investor puts all wealth in one asset, say, then the expected payoff of the portfolio is μ and the variance is σ^2.

If instead the investor puts share $1/n$ of her wealth in each asset, then the portfolio payoff is

$$Y_n = \sum_{i=1}^{n} \frac{X_i}{n} = \frac{1}{n} \sum_{i=1}^{n} X_i.$$

Try computing the mean and variance.

You will find that

- The mean is unchanged at μ, while

- the variance of the portfolio has fallen to σ^2/n.

Diversification reduces risk, as expected.

But there is a hidden assumption here: the variance of returns is finite.

If the distribution is heavy-tailed and the variance is infinite, then this logic is incorrect.

For example, we saw above that if every X_i is Cauchy, then so is Y_n.

This means that diversification doesn't help at all!

21.5.2 Fiscal policy

The heaviness of the tail in the wealth distribution matters for taxation and redistribution policies.

The same is true for the income distribution.

For example, the heaviness of the tail of the income distribution helps determine *how much revenue a given tax policy will raise*.

21.6 Classifying tail properties

Up until now we have discussed light and heavy tails without any mathematical definitions.

Let's now rectify this.

We will focus our attention on the right hand tails of nonnegative random variables and their distributions.

The definitions for left hand tails are very similar and we omit them to simplify the exposition.

21.6.1 Light and heavy tails

A distribution F with density f on $\mathbb{R}_+$ is called heavy-tailed if

$$\int_0^\infty \exp(tx)f(x)dx = \infty \;\; \text{for all } t > 0. \tag{21.4}$$

We say that a nonnegative random variable X is **heavy-tailed** if its density is heavy-tailed.

This is equivalent to stating that its **moment generating function** $m(t) := \mathbb{E}\exp(tX)$ is infinite for all $t > 0$.

For example, the log-normal distribution is heavy-tailed because its moment generating function is infinite everywhere on $(0, \infty)$.

The Pareto distribution is also heavy-tailed.

Less formally, a heavy-tailed distribution is one that is not exponentially bounded (i.e. the tails are heavier than the exponential distribution).

A distribution F on $\mathbb{R}_+$ is called **light-tailed** if it is not heavy-tailed.

A nonnegative random variable X is **light-tailed** if its distribution F is light-tailed.

For example, every random variable with bounded support is light-tailed. (Why?)

As another example, if X has the exponential distribution, with cdf $F(x) = 1 - \exp(-\lambda x)$ for some $\lambda > 0$, then its moment generating function is

$$m(t) = \frac{\lambda}{\lambda - t} \quad \text{when } t < \lambda$$

In particular, $m(t)$ is finite whenever $t < \lambda$, so X is light-tailed.

One can show that if X is light-tailed, then all of its moments are finite.

Conversely, if some moment is infinite, then X is heavy-tailed.

The latter condition is not necessary, however.

For example, the lognormal distribution is heavy-tailed but every moment is finite.

21.7 Further reading

For more on heavy tails in the wealth distribution, see e.g., [Vilfredo, 1896] and [Benhabib and Bisin, 2018].

For more on heavy tails in the firm size distribution, see e.g., [Axtell, 2001], [Gabaix, 2016].

For more on heavy tails in the city size distribution, see e.g., [Rozenfeld *et al.*, 2011], [Gabaix, 2016].

There are other important implications of heavy tails, aside from those discussed above.

For example, heavy tails in income and wealth affect productivity growth, business cycles, and political economy.

For further reading, see, for example, [Acemoglu and Robinson, 2002], [Glaeser *et al.*, 2003], [Bhandari *et al.*, 2018] or [Ahn *et al.*, 2018].

21.8 Exercises

Exercise 21.8.1

Prove: If X has a Pareto tail with tail index α, then $\mathbb{E}[X^r] = \infty$ for all $r \geq \alpha$.

Solution to Exercise 21.8.1

Let X have a Pareto tail with tail index α and let F be its cdf.

Fix $r \geq \alpha$.

In view of (21.2), we can take positive constants b and $\bar{x}$ such that

$$\mathbb{P}\{X > x\} \geq bx^{-\alpha} \text{ whenever } x \geq \bar{x}$$

But then

$$\mathbb{E}X^r = r \int_0^\infty x^{r-1}\mathbb{P}\{X > x\}dx \geq r \int_0^{\bar{x}} x^{r-1}\mathbb{P}\{X > x\}dx + r \int_{\bar{x}}^\infty x^{r-1}bx^{-\alpha}dx.$$

We know that $\int_{x}^{\infty} x^{r-\alpha-1} dx = \infty$ whenever $r - \alpha - 1 \geq -1$.

Since $r \geq \alpha$, we have $\mathbb{E}X^r = \infty$.

Exercise 21.8.2

Repeat exercise 1, but replace the three distributions (two normal, one Cauchy) with three Pareto distributions using different choices of α.

For α, try 1.15, 1.5 and 1.75.

Use `np.random.seed(11)` to set the seed.

Solution to Exercise 21.8.2

```python
from scipy.stats import pareto

np.random.seed(11)

n = 120
alphas = [1.15, 1.50, 1.75]

fig, axes = plt.subplots(3, 1, figsize=(6, 8))

for (a, ax) in zip(alphas, axes):
    ax.set_ylim((-5, 50))
    data = pareto.rvs(size=n, scale=1, b=a)
    ax.plot(list(range(n)), data, linestyle='', marker='o', alpha=0.5, ms=4)
    ax.vlines(list(range(n)), 0, data, lw=0.2)
    ax.set_title(f"Pareto draws with $\\alpha = {a}$", fontsize=11)

plt.subplots_adjust(hspace=0.4)

plt.show()
```

Pareto draws with $\alpha = 1.15$

Pareto draws with $\alpha = 1.5$

Pareto draws with $\alpha = 1.75$
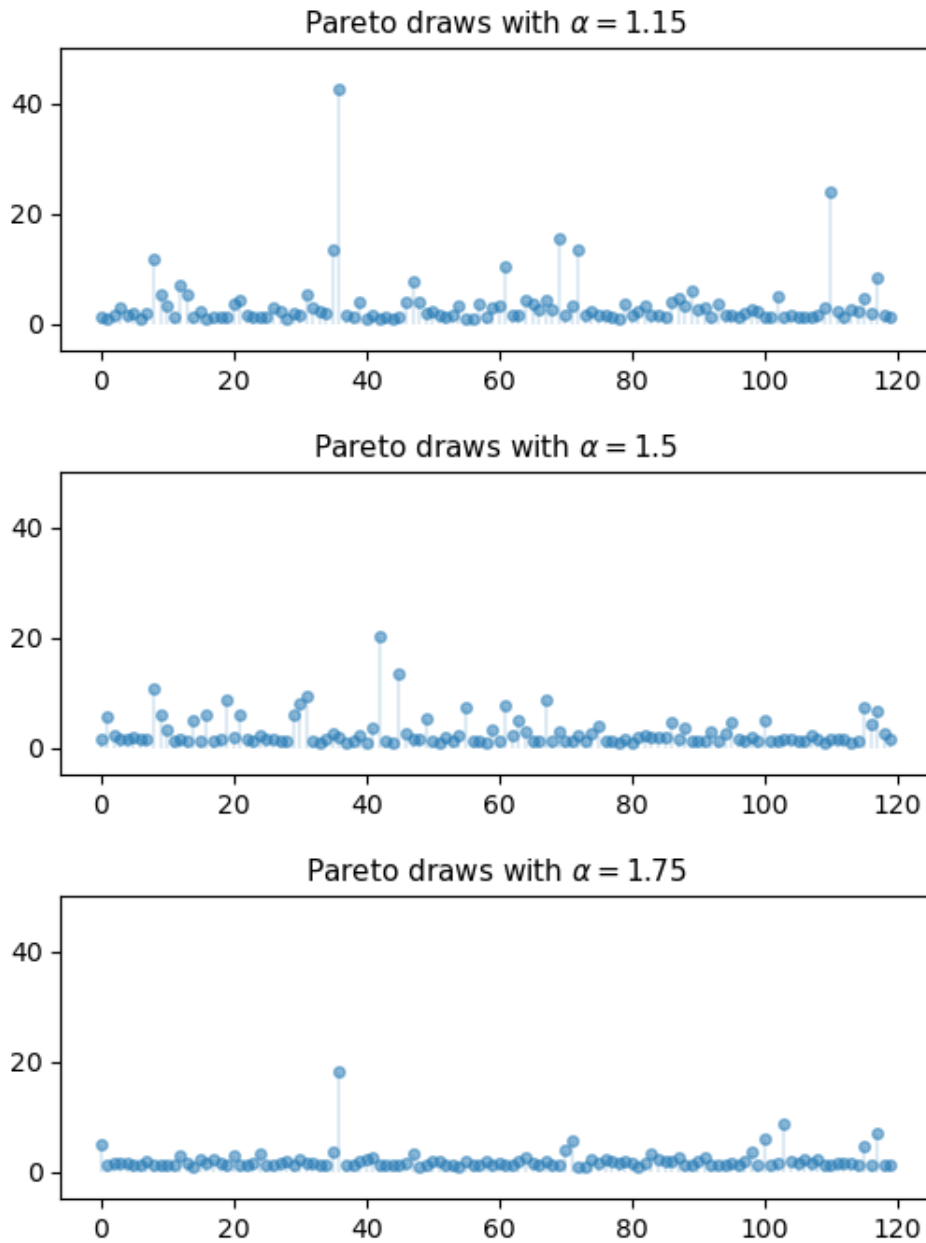
Exercise 21.8.3

There is an ongoing argument about whether the firm size distribution should be modeled as a Pareto distribution or a lognormal distribution (see, e.g., [Fujiwara *et al.*, 2004], [Kondo *et al.*, 2018] or [Schluter and Trede, 2019]).

This sounds esoteric but has real implications for a variety of economic phenomena.

To illustrate this fact in a simple way, let us consider an economy with 100,000 firms, an interest rate of `r = 0.05` and a corporate tax rate of 15%.

Your task is to estimate the present discounted value of projected corporate tax revenue over the next 10 years.

Because we are forecasting, we need a model.

We will suppose that

1. the number of firms and the firm size distribution (measured in profits) remain fixed and

2. the firm size distribution is either lognormal or Pareto.

Present discounted value of tax revenue will be estimated by

1. generating 100,000 draws of firm profit from the firm size distribution,

2. multiplying by the tax rate, and

3. summing the results with discounting to obtain present value.

The Pareto distribution is assumed to take the form (21.1) with $\bar{x} = 1$ and $\alpha = 1.05$.

(The value of the tail index α is plausible given the data [Gabaix, 2016].)

To make the lognormal option as similar as possible to the Pareto option, choose its parameters such that the mean and median of both distributions are the same.

Note that, for each distribution, your estimate of tax revenue will be random because it is based on a finite number of draws.

To take this into account, generate 100 replications (evaluations of tax revenue) for each of the two distributions and compare the two samples by

- producing a violin plot visualizing the two samples side-by-side and

- printing the mean and standard deviation of both samples.

For the seed use `np.random.seed(1234)`.

What differences do you observe?

(Note: a better approach to this problem would be to model firm dynamics and try to track individual firms given the current distribution. We will discuss firm dynamics in later lectures.)

Solution to Exercise 21.8.3

To do the exercise, we need to choose the parameters μ and σ of the lognormal distribution to match the mean and median of the Pareto distribution.

Here we understand the lognormal distribution as that of the random variable $\exp(\mu + \sigma Z)$ when Z is standard normal.

The mean and median of the Pareto distribution (21.1) with $\bar{x} = 1$ are

$$\text{mean} = \frac{\alpha}{\alpha - 1} \quad \text{and} \quad \text{median} = 2^{1/\alpha}$$

Using the corresponding expressions for the lognormal distribution leads us to the equations

$$\frac{\alpha}{\alpha - 1} = \exp(\mu + \sigma^2/2) \quad \text{and} \quad 2^{1/\alpha} = \exp(\mu)$$

which we solve for μ and σ given $\alpha = 1.05$.

Here is the code that generates the two samples, produces the violin plot and prints the mean and standard deviation of the two samples.

```python
num_firms = 100_000
num_years = 10
tax_rate = 0.15
r = 0.05

β = 1 / (1 + r)     # discount factor

x_bar = 1.0
α = 1.05

def pareto_rvs(n):
    "Uses a standard method to generate Pareto draws."
    u = np.random.uniform(size=n)
    y = x_bar / (u**(1/α))
    return y
```

Let's compute the lognormal parameters:

```python
μ = np.log(2) / α
σ_sq = 2 * (np.log(α/(α - 1)) - np.log(2)/α)
σ = np.sqrt(σ_sq)
```

Here's a function to compute a single estimate of tax revenue for a particular choice of distribution `dist`.

```python
def tax_rev(dist):
    tax_raised = 0
    for t in range(num_years):
        if dist == 'pareto':
            π = pareto_rvs(num_firms)
        else:
            π = np.exp(μ + σ * np.random.randn(num_firms))
        tax_raised += β**t * np.sum(π * tax_rate)
    return tax_raised
```

Now let's generate the violin plot.

```
num_reps = 100
np.random.seed(1234)

tax_rev_lognorm = np.empty(num_reps)
tax_rev_pareto = np.empty(num_reps)

for i in range(num_reps):
    tax_rev_pareto[i] = tax_rev('pareto')
    tax_rev_lognorm[i] = tax_rev('lognorm')

fig, ax = plt.subplots()

data = tax_rev_pareto, tax_rev_lognorm

ax.violinplot(data)

plt.show()
```
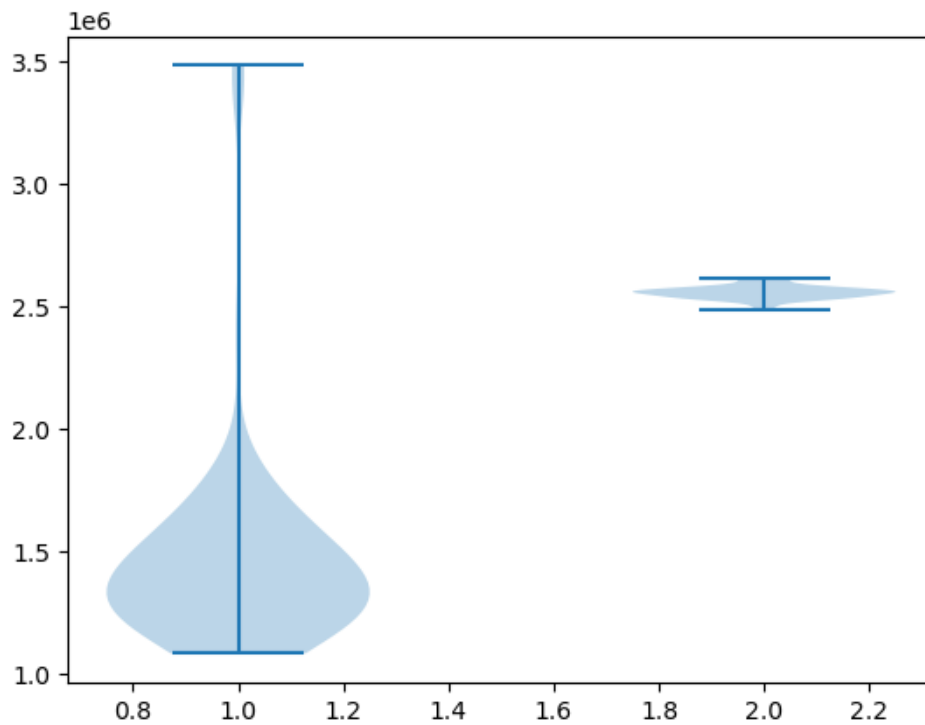

Finally, let's print the means and standard deviations.

```
tax_rev_pareto.mean(), tax_rev_pareto.std()
```

```
(1458729.0546623734, 406089.3613661567)
```

```
tax_rev_lognorm.mean(), tax_rev_lognorm.std()
```

```
(2556174.8615230713, 25586.44456513965)
```

Looking at the output of the code, our main conclusion is that the Pareto assumption leads to a lower mean and greater dispersion.

Exercise 21.8.4

The characteristic function of the Cauchy distribution is

$$\phi(t) = \mathbb{E}e^{itX} = \int e^{itx} f(x)dx = e^{-|t|} \tag{21.5}$$

Prove that the sample mean $\bar{X}_n$ of n independent draws $X_1, \dots, X_n$ from the Cauchy distribution has the same characteristic function as X_1.

(This means that the sample mean never converges.)

Solution to Exercise 21.8.4

By independence, the characteristic function of the sample mean becomes

$$
\begin{aligned}
\mathbb{E}e^{it\bar{X}_n} &= \mathbb{E}\exp\left\{i\frac{t}{n}\sum_{j=1}^{n} X_j\right\} \\
&= \mathbb{E}\prod_{j=1}^{n} \exp\left\{i\frac{t}{n}X_j\right\} \\
&= \prod_{j=1}^{n} \mathbb{E}\exp\left\{i\frac{t}{n}X_j\right\} = [\phi(t/n)]^n
\end{aligned}
$$

In view of (**??**), this is just $e^{-|t|}$.

Thus, in the case of the Cauchy distribution, the sample mean itself has the very same Cauchy distribution, regardless of n!

RACIAL SEGREGATION

22.1 Outline

In 1969, Thomas C. Schelling developed a simple but striking model of racial segregation [Schelling, 1969].

His model studies the dynamics of racially mixed neighborhoods.

Like much of Schelling's work, the model shows how local interactions can lead to surprising aggregate outcomes.

It studies a setting where agents (think of households) have relatively mild preference for neighbors of the same race.

For example, these agents might be comfortable with a mixed race neighborhood but uncomfortable when they feel "surrounded" by people from a different race.

Schelling illustrated the follow surprising result: in such a setting, mixed race neighborhoods are likely to be unstable, tending to collapse over time.

In fact the model predicts strongly divided neighborhoods, with high levels of segregation.

In other words, extreme segregation outcomes arise even though people's preferences are not particularly extreme.

These extreme outcomes happen because of *interactions* between agents in the model (e.g., households in a city) that drive self-reinforcing dynamics in the model.

These ideas will become clearer as the lecture unfolds.

In recognition of his work on segregation and other research, Schelling was awarded the 2005 Nobel Prize in Economic Sciences (joint with Robert Aumann).

Let's start with some imports:

```
import matplotlib.pyplot as plt
from random import uniform, seed
from math import sqrt
import numpy as np
```

22.2 The model

In this section we will build a version of Schelling's model.

22.2.1 Set-Up

We will cover a variation of Schelling's model that is different from the original but also easy to program and, at the same time, captures his main idea.

Suppose we have two types of people: orange people and green people.

Assume there are n of each type.

These agents all live on a single unit square.

Thus, the location (e.g, address) of an agent is just a point (x, y), where $0 < x, y < 1$.

- The set of all points (x, y) satisfying $0 < x, y < 1$ is called the **unit square**

- Below we denote the unit square by S

22.2.2 Preferences

We will say that an agent is *happy* if 5 or more of her 10 nearest neighbors are of the same type.

An agent who is not happy is called *unhappy*.

For example,

- if an agent is orange and 5 of her 10 nearest neighbors are orange, then she is happy.

- if an agent is green and 8 of her 10 nearest neighbors are orange, then she is unhappy.

'Nearest' is in terms of Euclidean distance.

An important point to note is that agents are **not** averse to living in mixed areas.

They are perfectly happy if half of their neighbors are of the other color.

22.2.3 Behavior

Initially, agents are mixed together (integrated).

In particular, we assume that the initial location of each agent is an independent draw from a bivariate uniform distribution on the unit square S.

- First their x coordinate is drawn from a uniform distribution on $(0, 1)$

- Then, independently, their y coordinate is drawn from the same distribution.

Now, cycling through the set of all agents, each agent is now given the chance to stay or move.

Each agent stays if they are happy and moves if they are unhappy.

The algorithm for moving is as follows

Algorithm 22.2.1 (Jump Chain Algorithm)

1. Draw a random location in S

2. If happy at new location, move there

3. Otherwise, go to step 1

We cycle continuously through the agents, each time allowing an unhappy agent to move.

We continue to cycle until no one wishes to move.

22.3 Results

Let's now implement and run this simulation.

In what follows, agents are modeled as objects.

Here's an indication of their structure:

```
* Data:

    * type (green or orange)
    * location

* Methods:

    * determine whether happy or not given locations of other agents
```

```
   * If not happy, move
       * find a new location where happy
```

Let's build them.

```python
class Agent:

    def __init__(self, type):
        self.type = type
        self.draw_location()

    def draw_location(self):
        self.location = uniform(0, 1), uniform(0, 1)

    def get_distance(self, other):
        "Computes the euclidean distance between self and other agent."
        a = (self.location[0] - other.location[0])**2
        b = (self.location[1] - other.location[1])**2
        return sqrt(a + b)

    def happy(self,
                agents,                 # List of other agents
                num_neighbors=10,       # No. of agents viewed as neighbors
                require_same_type=5):   # How many neighbors must be same type
        """
            True if a sufficient number of nearest neighbors are of the same
            type.
        """

        distances = []

        # Distances is a list of pairs (d, agent), where d is distance from
        # agent to self
        for agent in agents:
            if self != agent:
                distance = self.get_distance(agent)
                distances.append((distance, agent))

        # Sort from smallest to largest, according to distance
        distances.sort()

        # Extract the neighboring agents
        neighbors = [agent for d, agent in distances[:num_neighbors]]
```

```python
        # Count how many neighbors have the same type as self
        num_same_type = sum(self.type == agent.type for agent in neighbors)
        return num_same_type >= require_same_type


    def update(self, agents):
        "If not happy, then randomly choose new locations until happy."
        while not self.happy(agents):
            self.draw_location()
```

Here's some code that takes a list of agents and produces a plot showing their locations on the unit square.

Orange agents are represented by orange dots and green ones are represented by green dots.

```python
def plot_distribution(agents, cycle_num):
    "Plot the distribution of agents after cycle_num rounds of the loop."
    x_values_0, y_values_0 = [], []
    x_values_1, y_values_1 = [], []
    # == Obtain locations of each type == #
    for agent in agents:
        x, y = agent.location
        if agent.type == 0:
            x_values_0.append(x)
            y_values_0.append(y)
        else:
            x_values_1.append(x)
            y_values_1.append(y)
    fig, ax = plt.subplots()
    plot_args = {'markersize': 8, 'alpha': 0.8}
    ax.set_facecolor('azure')
    ax.plot(x_values_0, y_values_0,
        'o', markerfacecolor='orange', **plot_args)
    ax.plot(x_values_1, y_values_1,
        'o', markerfacecolor='green', **plot_args)
    ax.set_title(f'Cycle {cycle_num-1}')
    plt.show()
```

And here's some pseudocode for the main loop, where we cycle through the agents until no one wishes to move.

The pseudocode is

```
plot the distribution
while agents are still moving
```

```
    for agent in agents
        give agent the opportunity to move
plot the distribution
```

The real code is below

```python
def run_simulation(num_of_type_0=600,
                   num_of_type_1=600,
                   max_iter=100_000,        # Maximum number of iterations
                   set_seed=1234):

    # Set the seed for reproducibility
    seed(set_seed)

    # Create a list of agents of type 0
    agents = [Agent(0) for i in range(num_of_type_0)]
    # Append a list of agents of type 1
    agents.extend(Agent(1) for i in range(num_of_type_1))

    # Initialize a counter
    count = 1

    # Plot the initial distribution
    plot_distribution(agents, count)

    # Loop until no agent wishes to move
    while count < max_iter:
        print('Entering loop ', count)
        count += 1
        no_one_moved = True
        for agent in agents:
            old_location = agent.location
            agent.update(agents)
            if agent.location != old_location:
                no_one_moved = False
        if no_one_moved:
            break

    # Plot final distribution
    plot_distribution(agents, count)

    if count < max_iter:
        print(f'Converged after {count} iterations.')
```

```
    else:
        print('Hit iteration bound and terminated.')
```

Let's have a look at the results.

```
run_simulation()
```
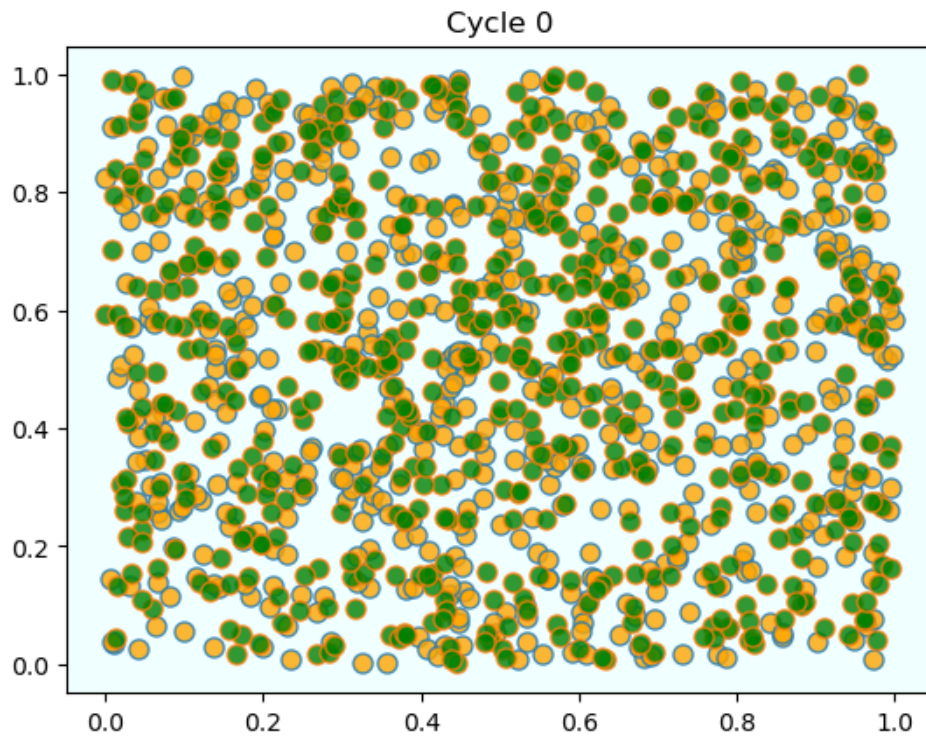

Cycle 0

```
Entering loop  1
```

```
Entering loop  2
```

```
Entering loop  3
```

```
Entering loop  4
```

```
Entering loop  5
```

```
Entering loop  6
```

```
Entering loop  7
```
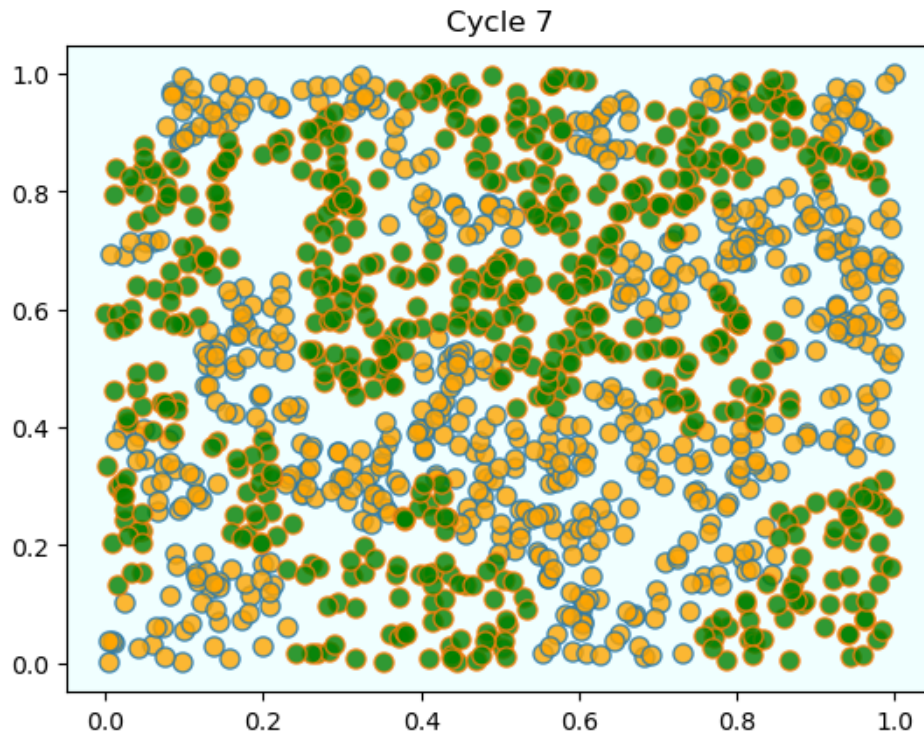

Cycle 7

```
Converged after 8 iterations.
```

As discussed above, agents are initially mixed randomly together.

But after several cycles, they become segregated into distinct regions.

In this instance, the program terminated after a small number of cycles through the set of agents, indicating that all agents had reached a state of happiness.

What is striking about the pictures is how rapidly racial integration breaks down.

This is despite the fact that people in the model don't actually mind living mixed with the other type.

Even with these preferences, the outcome is a high degree of segregation.

22.4 Exercises

Exercise 22.4.1

The object oriented style that we used for coding above is neat but harder to optimize than procedural code (i.e., code based around functions rather than objects and methods).

Try writing a new version of the model that stores

- the locations of all agents as a 2D NumPy array of floats.

- the types of all agents as a flat NumPy array of integers.

Write functions that act on this data to update the model using the logic similar to that described above.

However, implement the following two changes:

1. Agents are offered a move at random (i.e., selected randomly and given the opportunity to move).

2. After an agent has moved, flip their type with probability 0.01

The second change introduces extra randomness into the model.

(We can imagine that, every so often, an agent moves to a different city and, with small probability, is replaced by an agent of the other type.)

Solution to Exercise 22.4.1

solution here

```python
from numpy.random import uniform, randint

n = 1000                    # number of agents (agents = 0, ..., n-1)
k = 10                      # number of agents regarded as neighbors
require_same_type = 5       # want >= require_same_type neighbors of the same type


def initialize_state():
    locations = uniform(size=(n, 2))
    types = randint(0, high=2, size=n)    # label zero or one
    return locations, types


def compute_distances_from_loc(loc, locations):
    """ Compute distance from location loc to all other points. """
    return np.linalg.norm(loc - locations, axis=1)
```

```python
def get_neighbors(loc, locations):
    " Get all neighbors of a given location. "
    all_distances = compute_distances_from_loc(loc, locations)
    indices = np.argsort(all_distances)   # sort agents by distance to loc
    neighbors = indices[:k]               # keep the k closest ones
    return neighbors

def is_happy(i, locations, types):
    happy = True
    agent_loc = locations[i, :]
    agent_type = types[i]
    neighbors = get_neighbors(agent_loc, locations)
    neighbor_types = types[neighbors]
    if sum(neighbor_types == agent_type) < require_same_type:
        happy = False
    return happy

def count_happy(locations, types):
    " Count the number of happy agents. "
    happy_sum = 0
    for i in range(n):
        happy_sum += is_happy(i, locations, types)
    return happy_sum

def update_agent(i, locations, types):
    " Move agent if unhappy. "
    moved = False
    while not is_happy(i, locations, types):
        moved = True
        locations[i, :] = uniform(), uniform()
    return moved

def plot_distribution(locations, types, title, savepdf=False):
    " Plot the distribution of agents after cycle_num rounds of the loop."
    fig, ax = plt.subplots()
    colors = 'orange', 'green'
    for agent_type, color in zip((0, 1), colors):
        idx = (types == agent_type)
        ax.plot(locations[idx, 0],
                locations[idx, 1],
                'o',
                markersize=8,
```

```
                markerfacecolor=color,
                alpha=0.8)
    ax.set_title(title)
    plt.show()


def sim_random_select(max_iter=100_000, flip_prob=0.01, test_freq=10_000):
    """
    Simulate by randomly selecting one household at each update.

    Flip the color of the household with probability `flip_prob`.

    """

    locations, types = initialize_state()
    current_iter = 0

    while current_iter <= max_iter:

        # Choose a random agent and update them
        i = randint(0, n)
        moved = update_agent(i, locations, types)

        if flip_prob > 0:
            # flip agent i's type with probability epsilon
            U = uniform()
            if U < flip_prob:
                current_type = types[i]
                types[i] = 0 if current_type == 1 else 1

        # Every so many updates, plot and test for convergence
        if current_iter % test_freq == 0:
            cycle = current_iter / n
            plot_distribution(locations, types, f'iteration {current_iter}')
            if count_happy(locations, types) == n:
                print(f"Converged at iteration {current_iter}")
                break

        current_iter += 1

    if current_iter > max_iter:
        print(f"Terminating at iteration {current_iter}")
```

When we run this we again find that mixed neighborhoods break down and segregation emerges.

Here's a sample run.

```
sim_random_select(max_iter=50_000, flip_prob=0.01, test_freq=10_000)
```

iteration 0

iteration 10000

iteration 20000

iteration 30000

iteration 40000
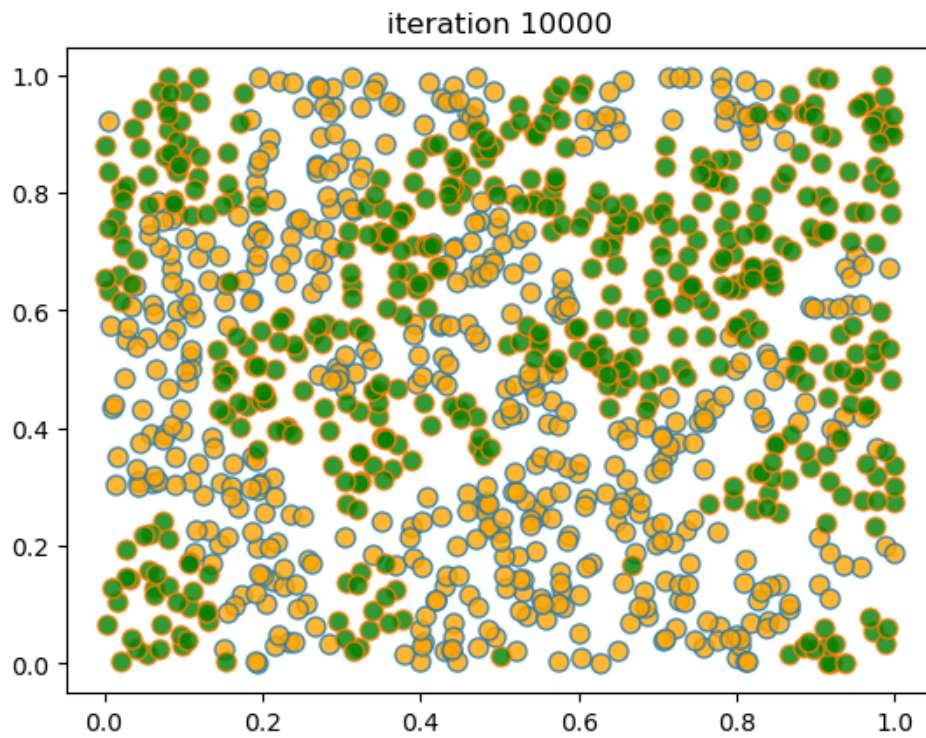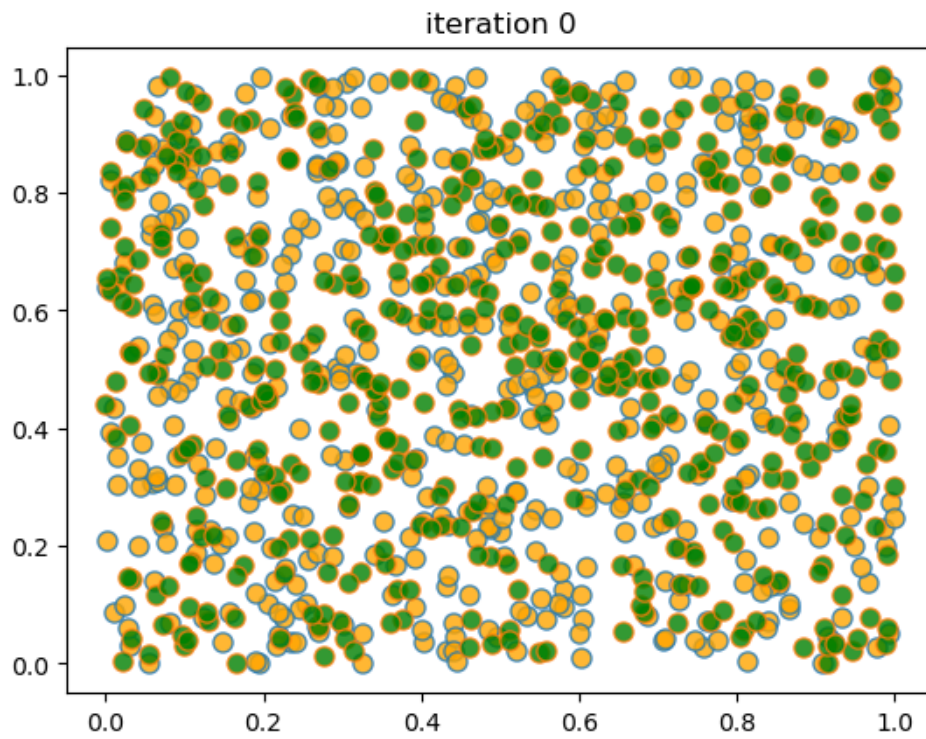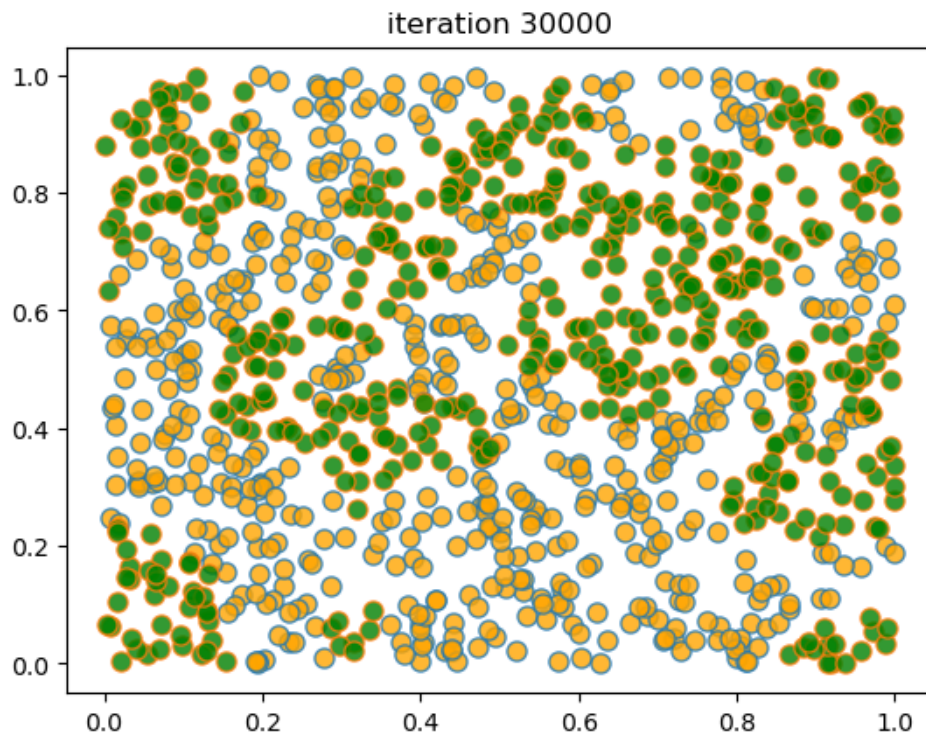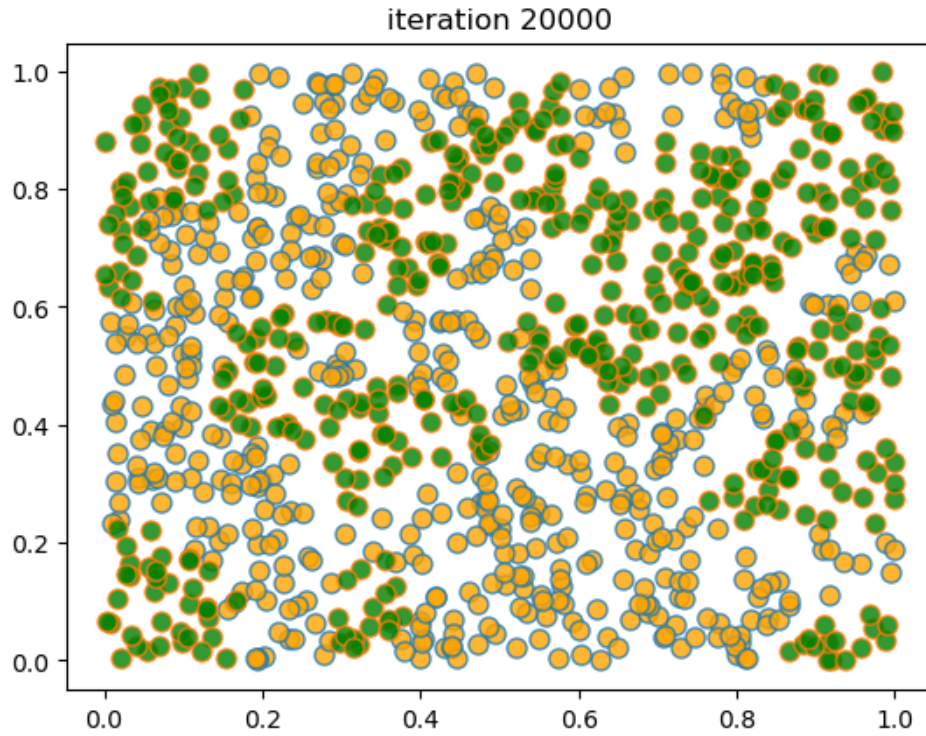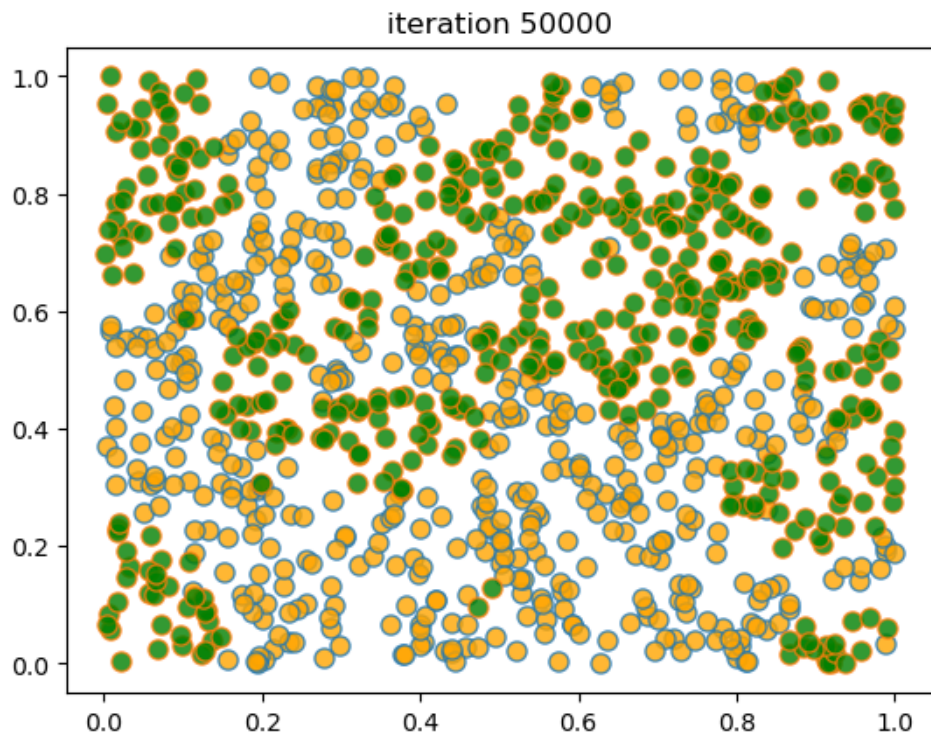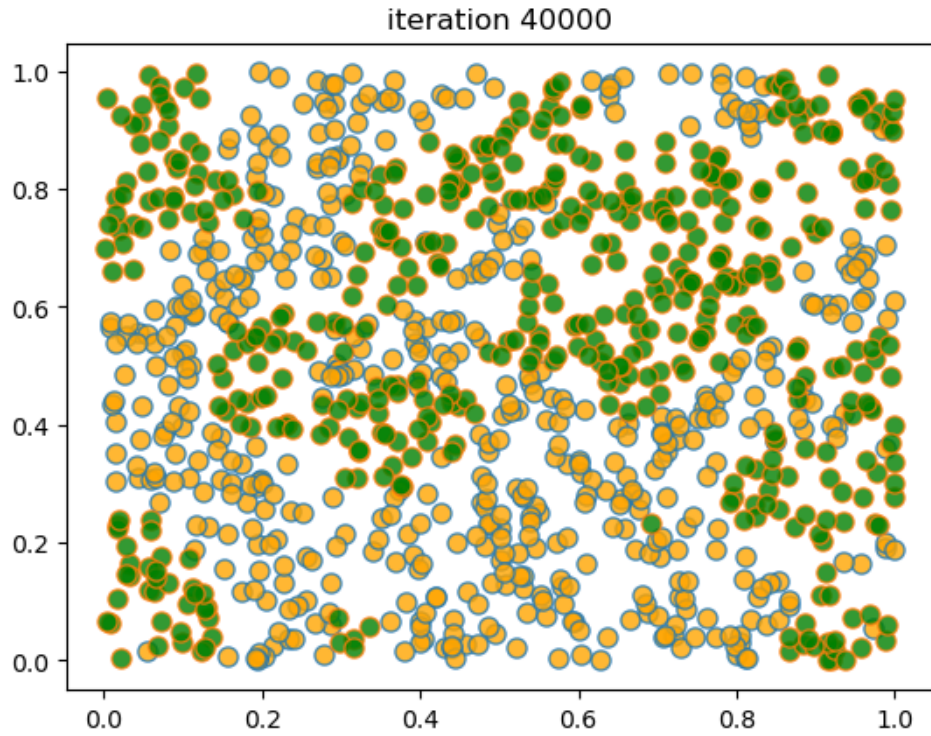
iteration 50000


```
Terminating at iteration 50001
```

Part VII

Nonlinear Dynamics

DYNAMICS IN ONE DIMENSION

23.1 Overview

In economics many variables depend on their past values

For example, it seems reasonable to believe that inflation last year with affects inflation this year.

(Perhaps high inflation last year will lead people to demand higher wages to compensate, which will feed into higher prices this year.)

Letting π_t be inflation this year and π_{t-1} be inflation last year, we can write this relationship in a general form as

$$\pi_t = f(\pi_{t-1})$$

where f is some function describing the relationship between the variables.

This equation is an example of one-dimensional discrete time dynamic system.

In this lecture we cover the foundations of one-dimensional discrete time dynamics.

(While most quantitative models have two or more state variables, the one-dimensional setting is a good place to learn foundations and understand key concepts.)

Let's start with some standard imports:

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

23.2 Some definitions

This section sets out the objects of interest and the kinds of properties we study.

23.2.1 Composition of functions

For this lecture you should know the following.

If

- g is a function from A to B and

- f is a function from B to C,

then the **composition** $f \circ g$ of f and g is defined by

$$(f \circ g)(x) = f(g(x))$$

For example, if

- $A = B = C = \mathbb{R}$, the set of real numbers,

- $g(x) = x^2$ and $f(x) = \sqrt{x}$, then $(f \circ g)(x) = \sqrt{x^2} = |x|$.

If f is a function from A to itself, then f^2 is the composition of f with itself.

For example, if $A = (0, \infty)$, the set of positive numbers, and $f(x) = \sqrt{x}$, then

$$f^2(x) = \sqrt{\sqrt{x}} = x^{1/4}$$

Similarly, if n is a positive integer, then f^n is n compositions of f with itself.

In the example above, $f^n(x) = x^{1/(2^n)}$.

23.2.2 Dynamic systems

A **(discrete time) dynamic system** is a set S and a function g that sends set S back into to itself.

Examples of dynamic systems include

- $S = (0, 1)$ and $g(x) = \sqrt{x}$

- $S = (0, 1)$ and $g(x) = x^2$

- $S = \mathbb{Z}$ (the integers) and $g(x) = 2x$

On the other hand, if $S = (-1, 1)$ and $g(x) = x + 1$, then S and g do not form a dynamic system, since $g(1) = 2$.

- g does not always send points in S back into S.

We care about dynamic systems because we can use them to study dynamics!

Given a dynamic system consisting of set S and function g, we can create a sequence $\{x_t\}$ of points in S by setting

$$x_{t+1} = g(x_t) \quad \text{with } x_0 \text{ given.} \tag{23.1}$$

This means that we choose some number x_0 in S and then take

$$x_0, \quad x_1 = g(x_0), \quad x_2 = g(x_1) = g(g(x_0)), \quad \text{etc.} \tag{23.2}$$

This sequence $\{x_t\}$ is called the **trajectory** of x_0 under g.

In this setting, S is called the **state space** and x_t is called the **state variable**.

Recalling that g^n is the n compositions of g with itself, we can write the trajectory more simply as

$$x_t = g^t(x_0) \quad \text{for } t = 0, 1, 2, ...$$

In all of what follows, we are going to assume that S is a subset of $\mathbb{R}$, the real numbers.

Equation (**??**) is sometimes called a **first order difference equation**

- first order means dependence on only one lag (i.e., earlier states such as x_{t-1} do not enter into (**??**)).

23.2.3 Example: a linear model

One simple example of a dynamic system is when $S = \mathbb{R}$ and $g(x) = ax + b$, where a, b are constants (sometimes called ``parameters'').

This leads to the **linear difference equation**

$$x_{t+1} = ax_t + b \quad \text{with } x_0 \text{ given.}$$

The trajectory of x_0 is

$$x_0, \quad ax_0 + b, \quad a^2 x_0 + ab + b, \quad \text{etc.} \tag{23.3}$$

Continuing in this way, and using our knowledge of *geometric series*, we find that, for any $t = 0, 1, 2, ...$,

$$x_t = a^t x_0 + b\frac{1 - a^t}{1 - a} \tag{23.4}$$

We have an exact expression for x_t for all non-negative integer t and hence a full understanding of the dynamics.

Notice in particular that $|a| < 1$, then, by (**??**), we have

$$x_t \to \frac{b}{1 - a} \quad \text{as } t \to \infty \tag{23.5}$$

regardless of x_0.

This is an example of what is called global stability, a topic we return to below.

23.2.4 Example: a nonlinear model

In the linear example above, we obtained an exact analytical expression for x_t in terms of arbitrary non-negative integer t and x_0.

This made analysis of dynamics very easy.

When models are nonlinear, however, the situation can be quite different.

For example, in a later lecture *The Solow-Swan Growth Model*, we will study the Solow-Swan growth model, which has dynamics

$$k_{t+1} = sAk_t^\alpha + (1 - \delta)k_t \tag{23.6}$$

Here $k = K/L$ is the per capita capital stock, s is the saving rate, A is the total factor productivity, α is the capital share, and δ is the depreciation rate.

All these parameter are positive and $0 < \alpha, \delta < 1$.

If you try to iterate like we did in (**??**), you will find that the algebra gets messy quickly.

Analyzing the dynamics of this model requires a different method (see below).

23.3 Stability

Consider a dynamic system consisting of set $S \subset \mathbb{R}$ and g mapping S to S.

23.3.1 Steady states

A **steady state** of this system is a point x^* in S such that $x^* = g(x^*)$.

In other words, x^* is a **fixed point** of the function g in S.

For example, for the linear model $x_{t+1} = ax_t + b$, you can use the definition to check that

- $x^* := b/(1 - a)$ is a steady state whenever $a \neq 1$,

- if $a = 1$ and $b = 0$, then every $x \in \mathbb{R}$ is a steady state,

- if $a = 1$ and $b \neq 0$, then the linear model has no steady state in $\mathbb{R}$.

23.3.2 Global stability

A steady state x^* of the dynamic system is called **globally stable** if, for all $x_0 \in S$,

$$x_t = g^t(x_0) \to x^* \text{ as } t \to \infty$$

For example, in the linear model $x_{t+1} = ax_t + b$ with $a \neq 1$, the steady state x^*

- is globally stable if $|a| < 1$ and

- fails to be globally stable otherwise.

This follows directly from (**??**).

23.3.3 Local stability

A steady state x^* of the dynamic system is called **locally stable** if there exists an $\epsilon > 0$ such that

$$|x_0 - x^*| < \epsilon \implies x_t = g^t(x_0) \to x^* \text{ as } t \to \infty$$

Obviously every globally stable steady state is also locally stable.

Here is an example where the converse is not true.

Example 23.3.1

Consider the self-map g on $\mathbb{R}$ defined by $g(x) = x^2$. The fixed point 1 is not stable.

For example, $g^t(x) \to \infty$ for any $x > 1$.

However, 0 is locally stable, because $-1 < x < 1$ implies that $g^t(x) \to 0$ as $t \to \infty$.

Since we have more than one fixed point, 0 is not globally stable.

23.4 Graphical analysis

As we saw above, analyzing the dynamics for nonlinear models is nontrivial.

There is no single way to tackle all nonlinear models.

However, there is one technique for one-dimensional models that provides a great deal of intuition.

This is a graphical approach based on **45-degree diagrams**.

Let's look at an example: the Solow-Swan model with dynamics given in (**??**).

We begin with some plotting code that you can ignore at first reading.

The function of the code is to produce 45-degree diagrams and time series plots.

Let's create a 45-degree diagram for the Solow-Swan model with a fixed set of parameters. Here's the update function corresponding to the model.

```
def g(k, A = 2, s = 0.3, alpha = 0.3, delta = 0.4):
    return A * s * k**alpha + (1 - delta) * k
```

Here is the 45-degree plot.

```
xmin, xmax = 0, 4   # Suitable plotting region.

plot45(g, xmin, xmax, 0, num_arrows=0)
```
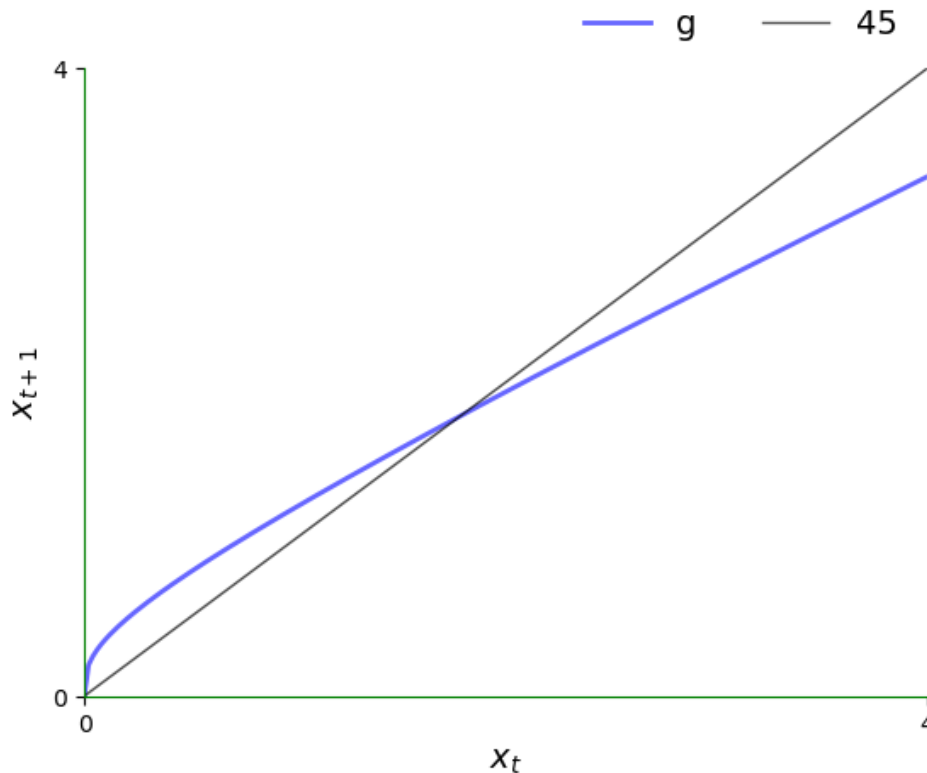

The plot shows the function g and the 45-degree line.

Think of k_t as a value on the horizontal axis.

To calculate k_{t+1}, we can use the graph of g to see its value on the vertical axis.

Clearly,

- If g lies above the 45-degree line at this point, then we have $k_{t+1} > k_t$.

- If g lies below the 45-degree line at this point, then we have $k_{t+1} < k_t$.

- If g hits the 45-degree line at this point, then we have $k_{t+1} = k_t$, so k_t is a steady state.

For the Solow-Swan model, there are two steady states when $S = \mathbb{R}_+ = [0, \infty)$.

- the origin $k = 0$

- the unique positive number such that $k = szk^\alpha + (1 - \delta)k$.

By using some algebra, we can show that in the second case, the steady state is

$$k^* = \left(\frac{sz}{\delta}\right)^{1/(1-\alpha)}$$

23.4.1 Trajectories

By the preceding discussion, in regions where g lies above the 45-degree line, we know that the trajectory is increasing.

The next figure traces out a trajectory in such a region so we can see this more clearly.

The initial condition is $k_0 = 0.25$.

```
k0 = 0.25
```

```
plot45(g, xmin, xmax, k0, num_arrows=5, var='k')
```
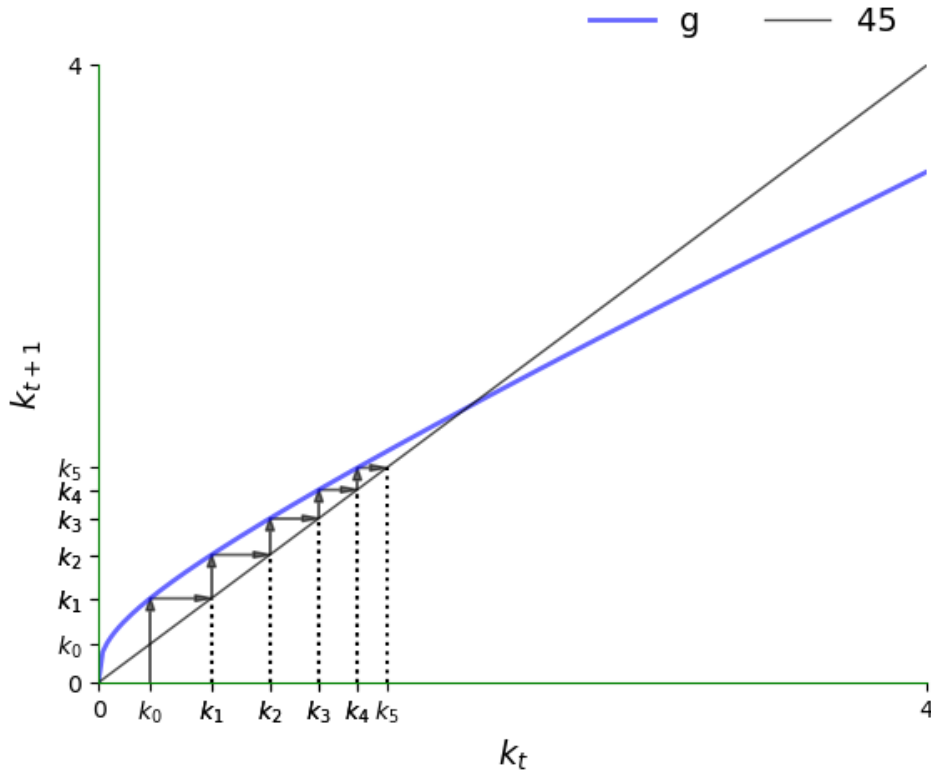
We can plot the time series of per capita capital corresponding to the figure above as follows:

```
ts_plot(g, xmin, xmax, k0, var='k')
```
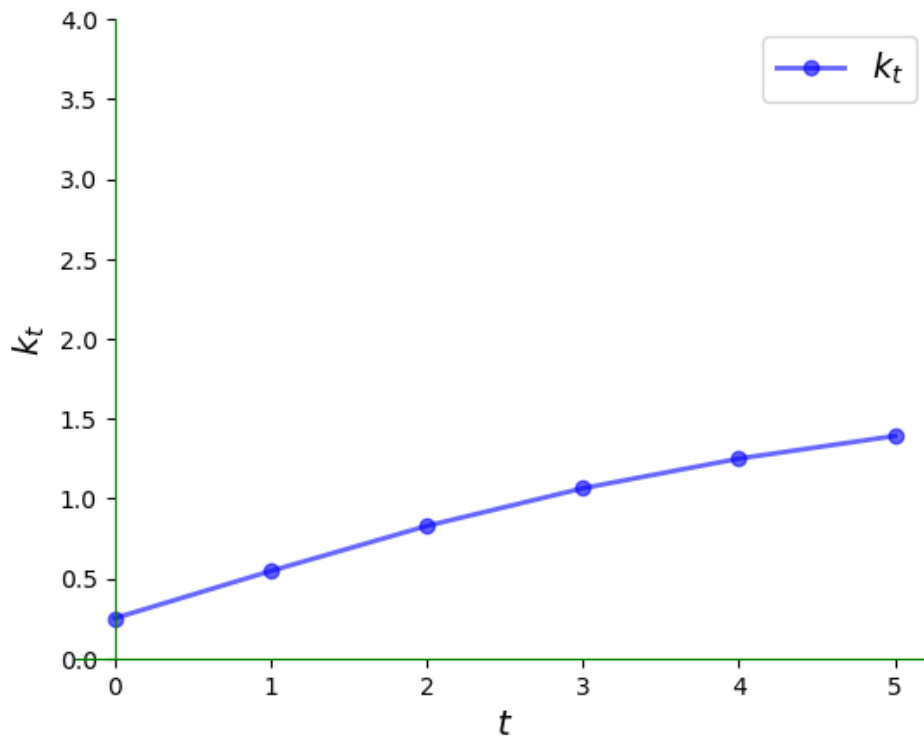

Here's a somewhat longer view:

```
ts_plot(g, xmin, xmax, k0, ts_length=20, var='k')
```
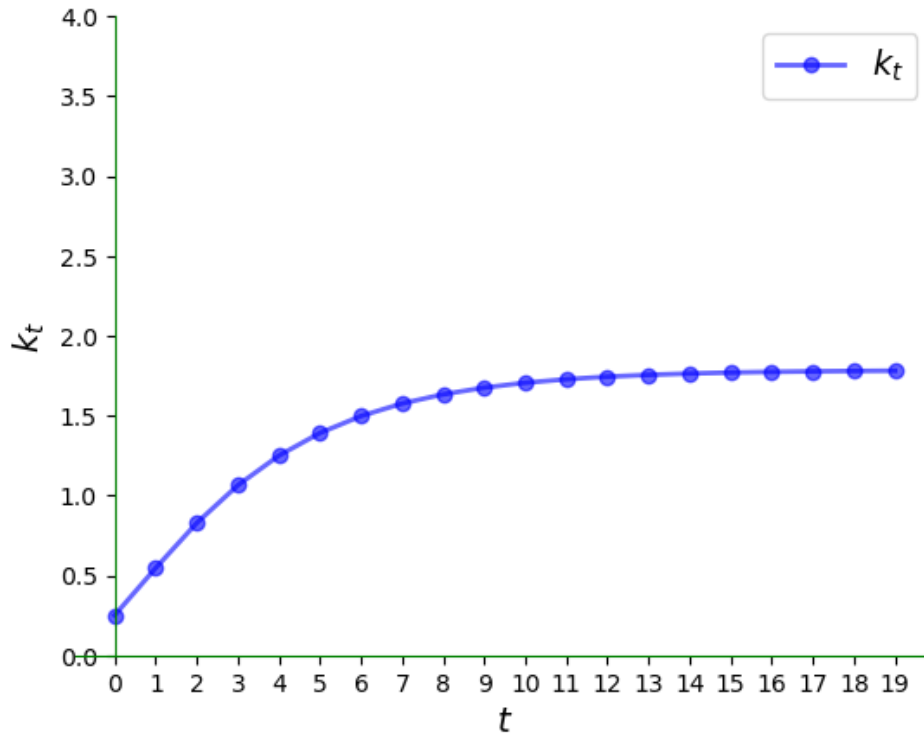
When per capita capital stock is higher than the unique positive steady state, we see that it declines:

```
k0 = 2.95

plot45(g, xmin, xmax, k0, num_arrows=5, var='k')
```
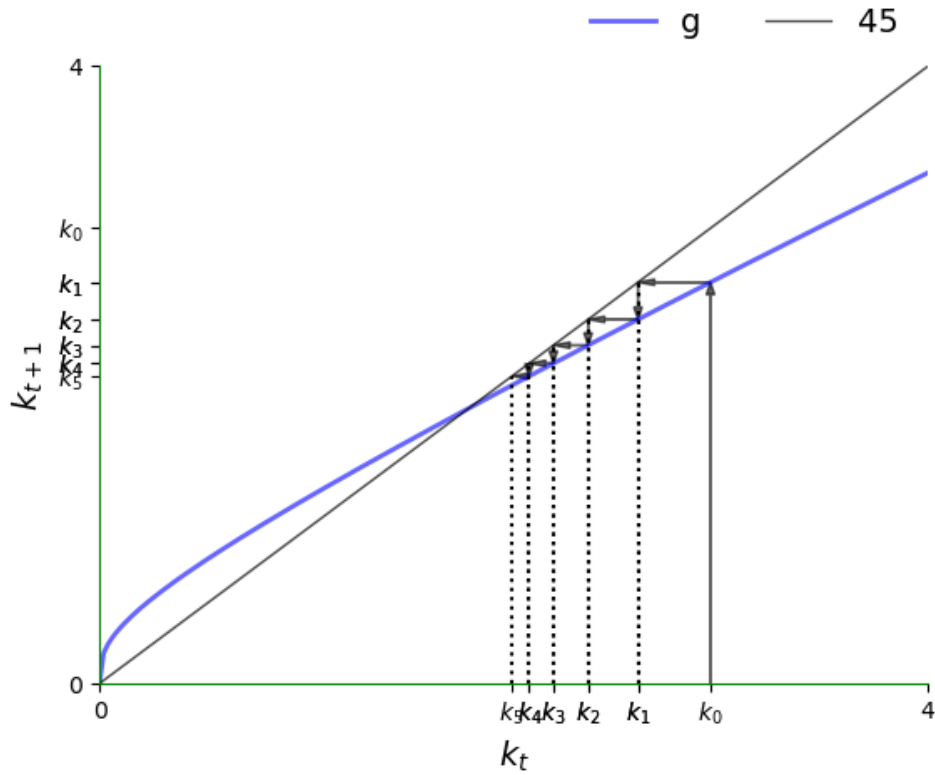
Here is the time series:

```
ts_plot(g, xmin, xmax, k0, var='k')
```
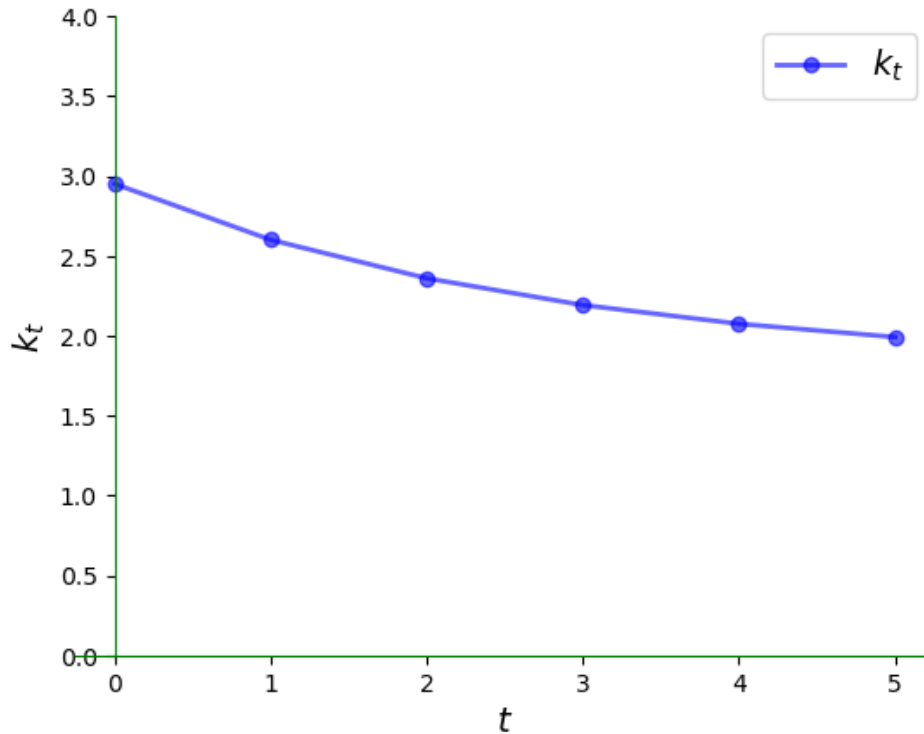
23.4.2 Complex dynamics

The Solow-Swan model is nonlinear but still generates very regular dynamics.

One model that generates irregular dynamics is the **quadratic map**

$$g(x) = 4x(1 - x), \qquad x \in [0, 1]$$

Let's have a look at the 45-degree diagram.

```
xmin, xmax = 0, 1
g = lambda x: 4 * x * (1 - x)


x0 = 0.3
plot45(g, xmin, xmax, x0, num_arrows=0)
```
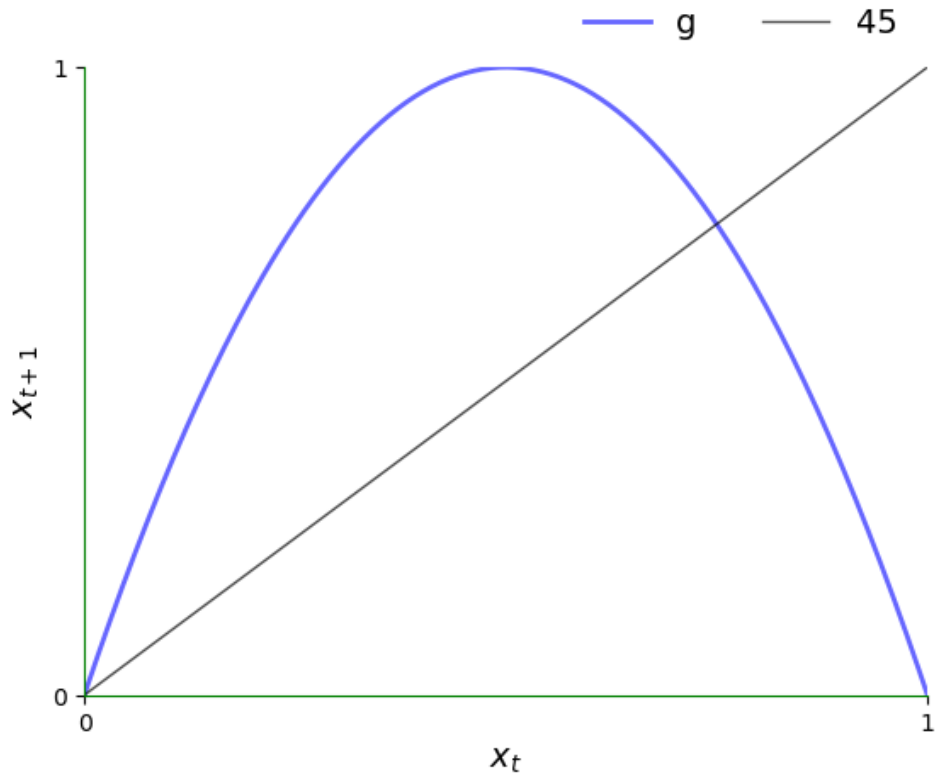
Now let's look at a typical trajectory.

```
plot45(g, xmin, xmax, x0, num_arrows=6)
```
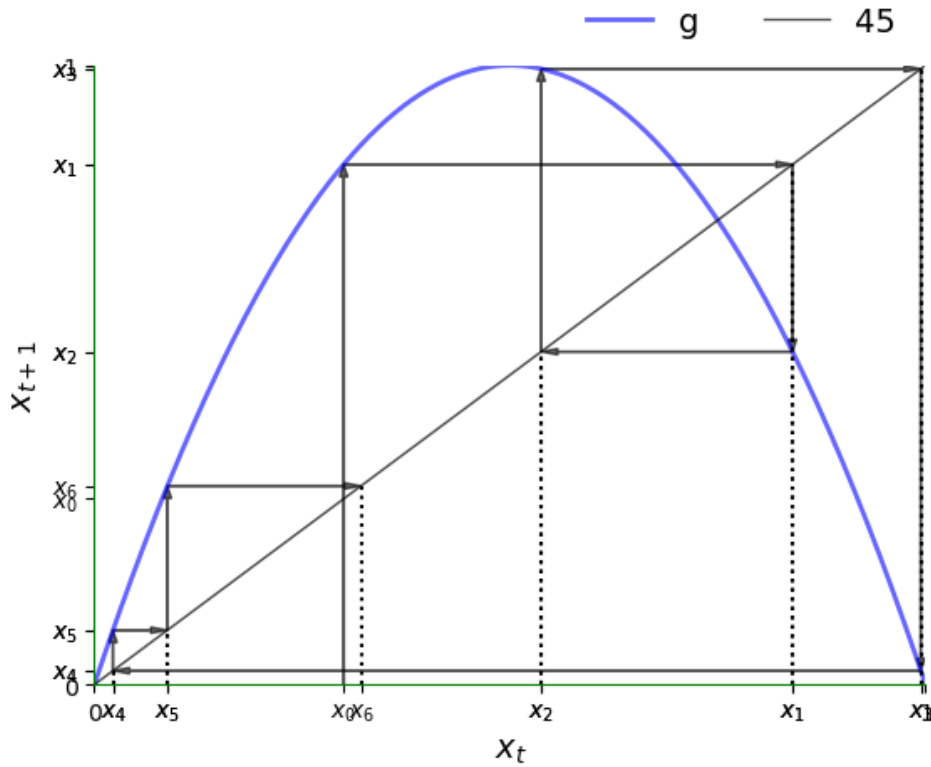
Notice how irregular it is.

Here is the corresponding time series plot.

```
ts_plot(g, xmin, xmax, x0, ts_length=6)
```
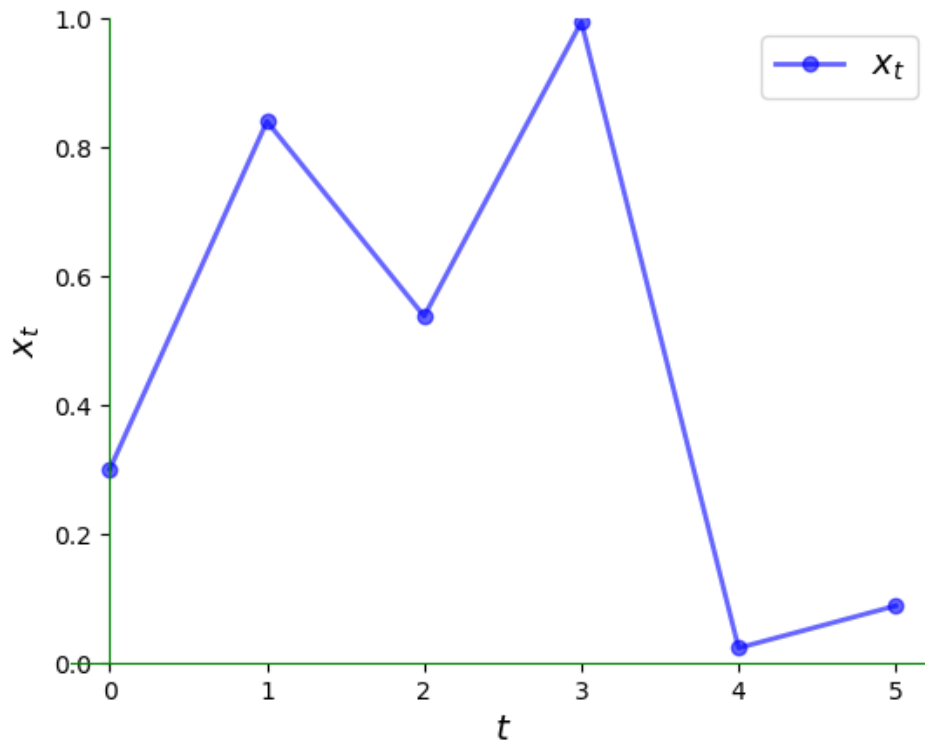
The irregularity is even clearer over a longer time horizon:

```
ts_plot(g, xmin, xmax, x0, ts_length=20)
```
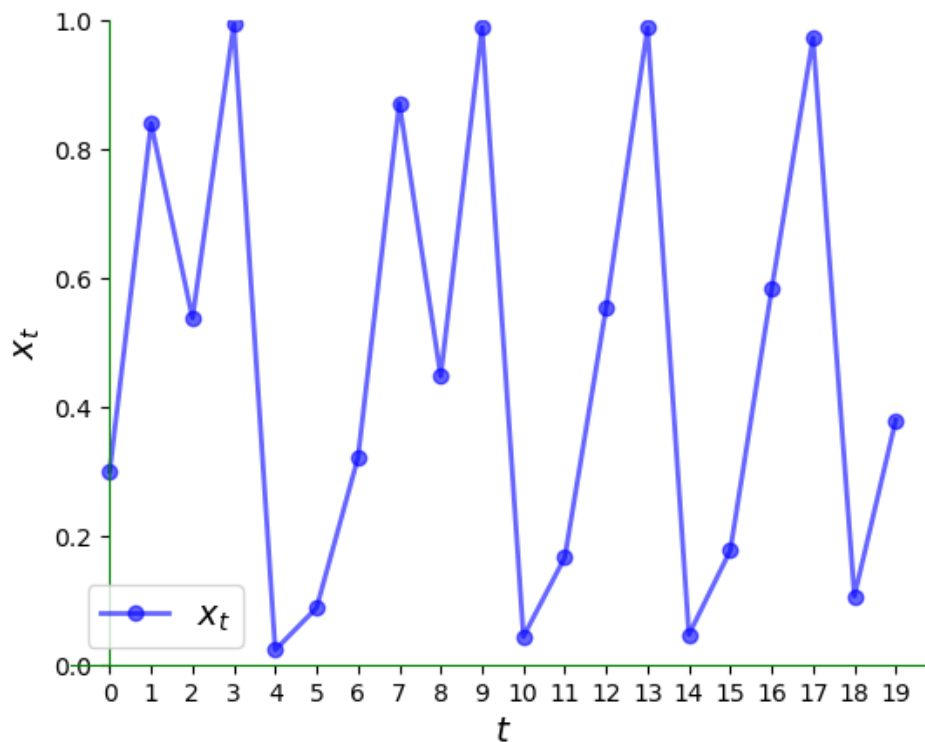
23.5 Exercises

Exercise 23.5.1

Consider again the linear model $x_{t+1} = ax_t + b$ with $a \neq 1$.

The unique steady state is $b/(1-a)$.

The steady state is globally stable if $|a| < 1$.

Try to illustrate this graphically by looking at a range of initial conditions.

What differences do you notice in the cases $a \in (-1, 0)$ and $a \in (0, 1)$?

Use $a = 0.5$ and then $a = -0.5$ and study the trajectories.

Set $b = 1$ throughout.

Solution to Exercise 23.5.1

We will start with the case $a = 0.5$.

Let's set up the model and plotting region:

```
a, b = 0.5, 1
xmin, xmax = -1, 3
g = lambda x: a * x + b
```

Now let's plot a trajectory:

```
x0 = -0.5
plot45(g, xmin, xmax, x0, num_arrows=5)
```
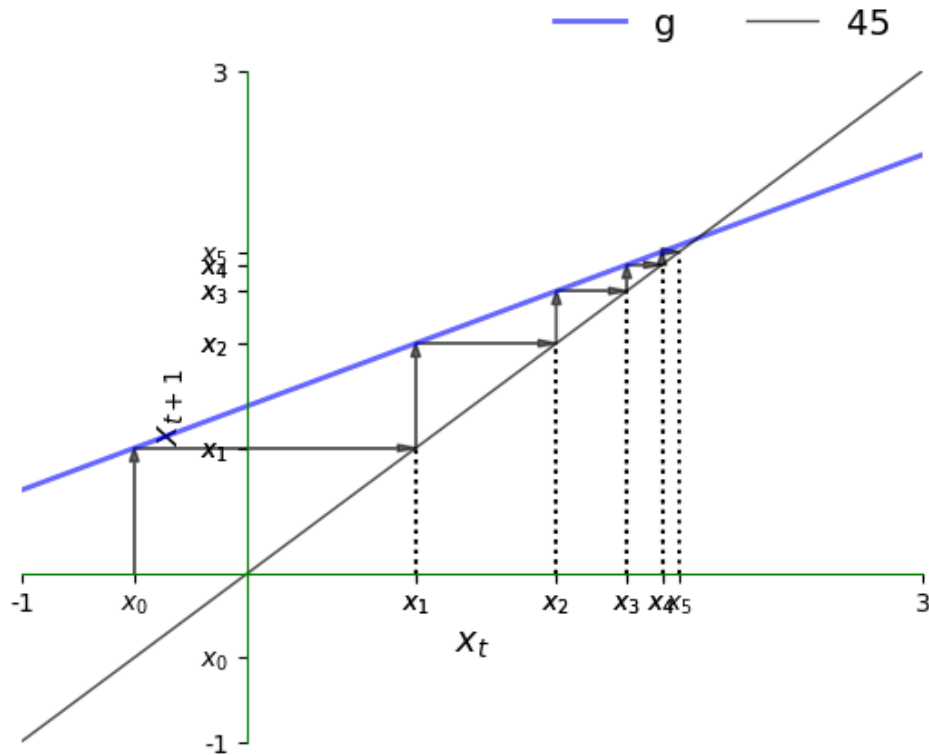
Here is the corresponding time series, which converges towards the steady state.

```
ts_plot(g, xmin, xmax, x0, ts_length=10)
```
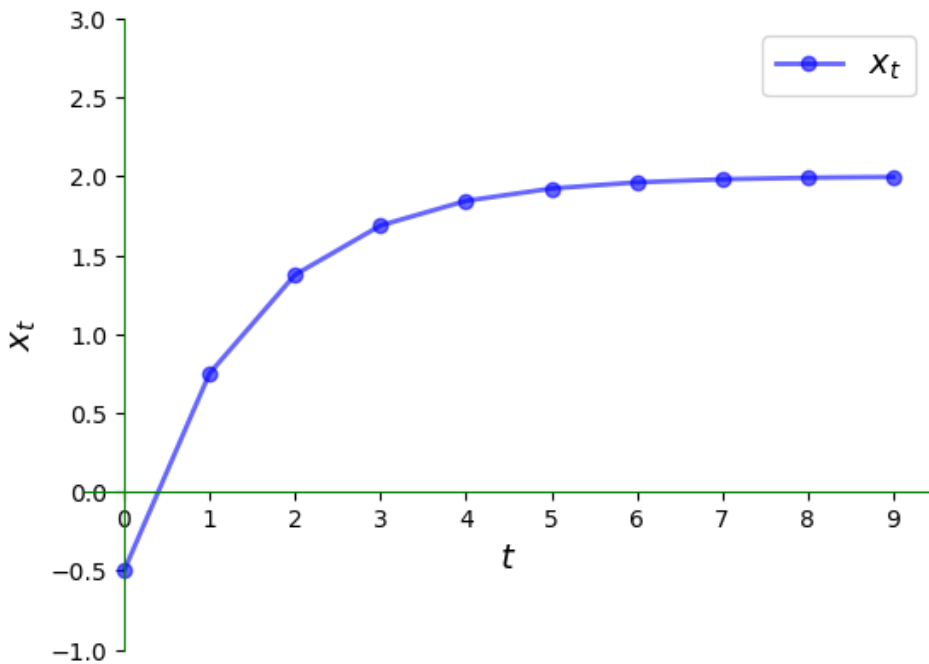
Now let's try $a = -0.5$ and see what differences we observe.

Let's set up the model and plotting region:

```
a, b = -0.5, 1
xmin, xmax = -1, 3
g = lambda x: a * x + b
```

Now let's plot a trajectory:

```
x0 = -0.5
plot45(g, xmin, xmax, x0, num_arrows=5)
```
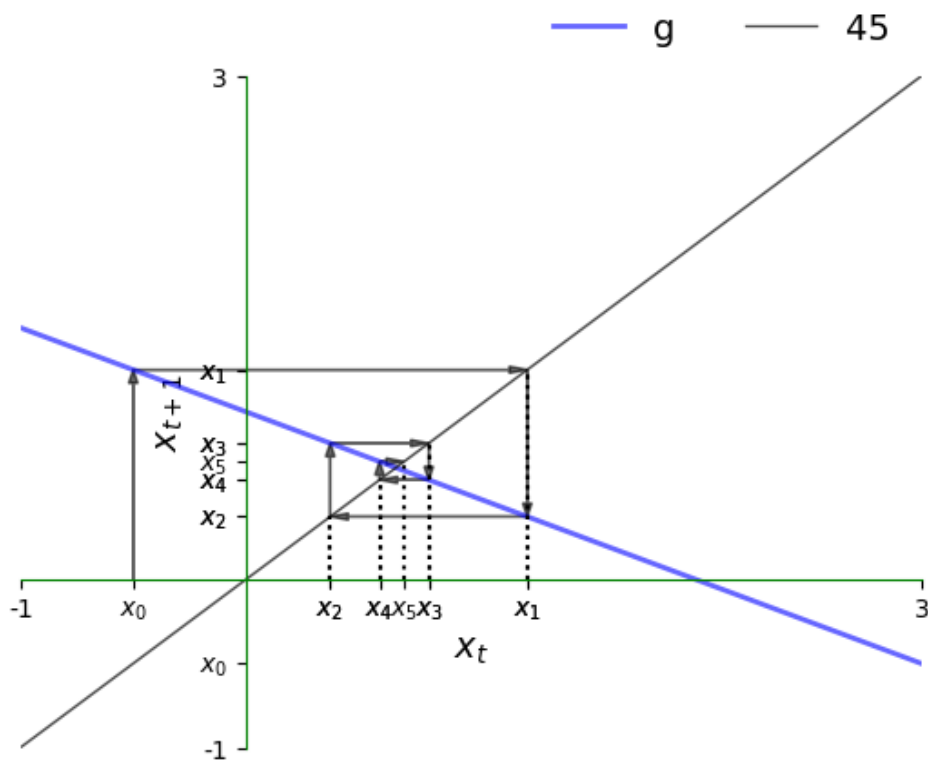

Here is the corresponding time series, which converges towards the steady state.

```
ts_plot(g, xmin, xmax, x0, ts_length=10)
```
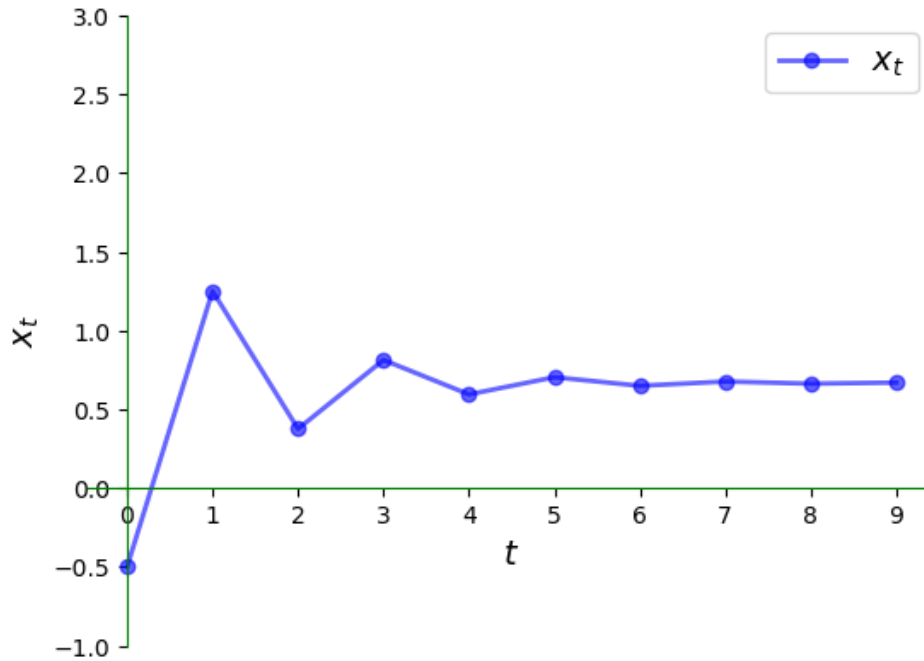
Once again, we have convergence to the steady state but the nature of convergence differs.

In particular, the time series jumps from above the steady state to below it and back again.

In the current context, the series is said to exhibit **damped oscillations**.

THE SOLOW-SWAN GROWTH MODEL

In this lecture we review a famous model due to Robert Solow (1925–2023) and Trevor Swan (1918–1989).

The model is used to study growth over the long run.

Although the model is simple, it contains some interesting lessons.

We will use the following imports.

```python
import matplotlib.pyplot as plt
import numpy as np
```

24.1 The model

In a Solow–Swan economy, agents save a fixed fraction of their current incomes.

Savings sustain or increase the stock of capital.

Capital is combined with labor to produce output, which in turn is paid out to workers and owners of capital.

To keep things simple, we ignore population and productivity growth.

For each integer $t \geq 0$, output Y_t in period t is given by $Y_t = F(K_t, L_t)$, where K_t is capital, L_t is labor and F is an aggregate production function.

The function F is assumed to be nonnegative and **homogeneous of degree one**, meaning that

$$F(\lambda K, \lambda L) = \lambda F(K, L) \quad \text{for all } \lambda \geq 0$$

Production functions with this property include

- the **Cobb-Douglas** function $F(K, L) = AK^{\alpha}L^{1-\alpha}$ with $0 \leq \alpha \leq 1$.

- the **CES** function $F(K, L) = \left\{aK^{\rho} + bL^{\rho}\right\}^{1/\rho}$ with $a, b, \rho > 0$.

Here, α is the output elasticity of capital and ρ is a parameter that determines the elasticity of substitution between capital and labor.

We assume a closed economy, so aggregate domestic investment equals aggregate domestic saving.

The saving rate is a constant s satisfying $0 \leq s \leq 1$, so that aggregate investment and saving both equal sY_t.

Capital depreciates: without replenishing through investment, one unit of capital today becomes $1 - \delta$ units tomorrow.

Thus,

$$K_{t+1} = sF(K_t, L_t) + (1 - \delta)K_t$$

Without population growth, L_t equals some constant L.

Setting $k_t := K_t / L$ and using homogeneity of degree one now yields

$$k_{t+1} = s\frac{F(K_t, L)}{L} + (1 - \delta)\frac{K_t}{L} = s\frac{F(K_t, L)}{L} + (1 - \delta)k_t = sF(k_t, 1) + (1 - \delta)k_t$$

With $f(k) := F(k, 1)$, the final expression for capital dynamics is

$$k_{t+1} = g(k_t) \text{ where } g(k) := sf(k) + (1 - \delta)k \tag{24.1}$$

Our aim is to learn about the evolution of k_t over time, given an exogenous initial capital stock k_0.

24.2 A graphical perspective

To understand the dynamics of the sequence $(k_t)_{t \geq 0}$ we use a 45-degree diagram.

To do so, we first need to specify the functional form for f and assign values to the parameters.

We choose the Cobb–Douglas specification $f(k) = Ak^\alpha$ and set $A = 2.0$, $\alpha = 0.3$, $s = 0.3$ and $\delta = 0.4$.

The function g from (**??**) is then plotted, along with the 45-degree line.

Let's define the constants.

```
A, s, alpha, delta = 2, 0.3, 0.3, 0.4
x0 = 0.25
xmin, xmax = 0, 3
```

Now, we define the function g.

```
def g(A, s, alpha, delta, k):
    return A * s * k**alpha + (1 - delta) * k
```

Let's plot the 45-degree diagram of g.

```python
def plot45(kstar=None):
    xgrid = np.linspace(xmin, xmax, 12000)

    fig, ax = plt.subplots()

    ax.set_xlim(xmin, xmax)

    g_values = g(A, s, alpha, delta, xgrid)

    ymin, ymax = np.min(g_values), np.max(g_values)
    ax.set_ylim(ymin, ymax)

    lb = r'$g(k) = sAk^{\alpha} + (1 - \delta)k$'
    ax.plot(xgrid, g_values,  lw=2, alpha=0.6, label=lb)
    ax.plot(xgrid, xgrid, 'k-', lw=1, alpha=0.7, label='$45^{\circ}$')

    if kstar:
        fps = (kstar,)

        ax.plot(fps, fps, 'go', ms=10, alpha=0.6)

        ax.annotate(r'$k^* = (sA / \delta)^{(1/(1-\alpha))}$',
                 xy=(kstar, kstar),
                 xycoords='data',
                 xytext=(-40, -60),
                 textcoords='offset points',
                 fontsize=14,
                 arrowprops=dict(arrowstyle="->"))

    ax.legend(loc='upper left', frameon=False, fontsize=12)

    ax.set_xticks((0, 1, 2, 3))
    ax.set_yticks((0, 1, 2, 3))

    ax.set_xlabel('$k_t$', fontsize=12)
    ax.set_ylabel('$k_{t+1}$', fontsize=12)

    plt.show()
```

```python
plot45()
```
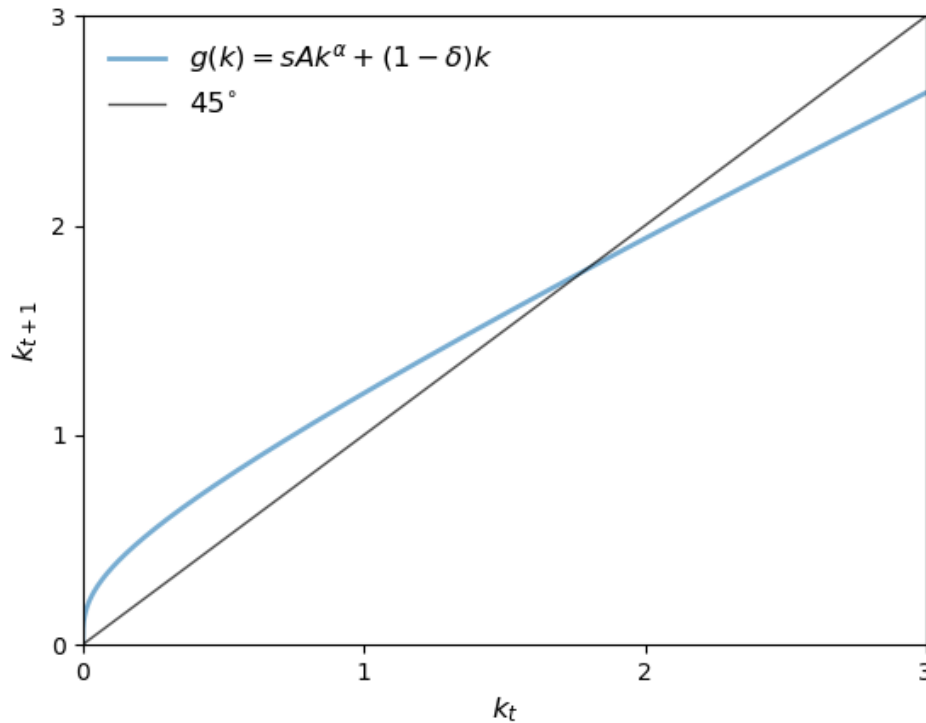
Suppose, at some k_t, the value $g(k_t)$ lies strictly above the 45-degree line.

Then we have $k_{t+1} = g(k_t) > k_t$ and capital per worker rises.

If $g(k_t) < k_t$ then capital per worker falls.

If $g(k_t) = k_t$, then we are at a **steady state** and k_t remains constant.

(A *steady state* of the model is a fixed point of the mapping g.)

From the shape of the function g in the figure, we see that there is a unique steady state in $(0, \infty)$.

It solves $k = sAk^\alpha + (1 - \delta)k$ and hence is given by

$$k^* := \left(\frac{sA}{\delta}\right)^{1/(1-\alpha)} \tag{24.2}$$

If initial capital is below k^*, then capital increases over time.

If initial capital is above this level, then the reverse is true.

Let's plot the 45-degree diagram to show the k^* in the plot.

```
kstar = ((s * A) / delta)**(1/(1 - alpha))
plot45(kstar)
```
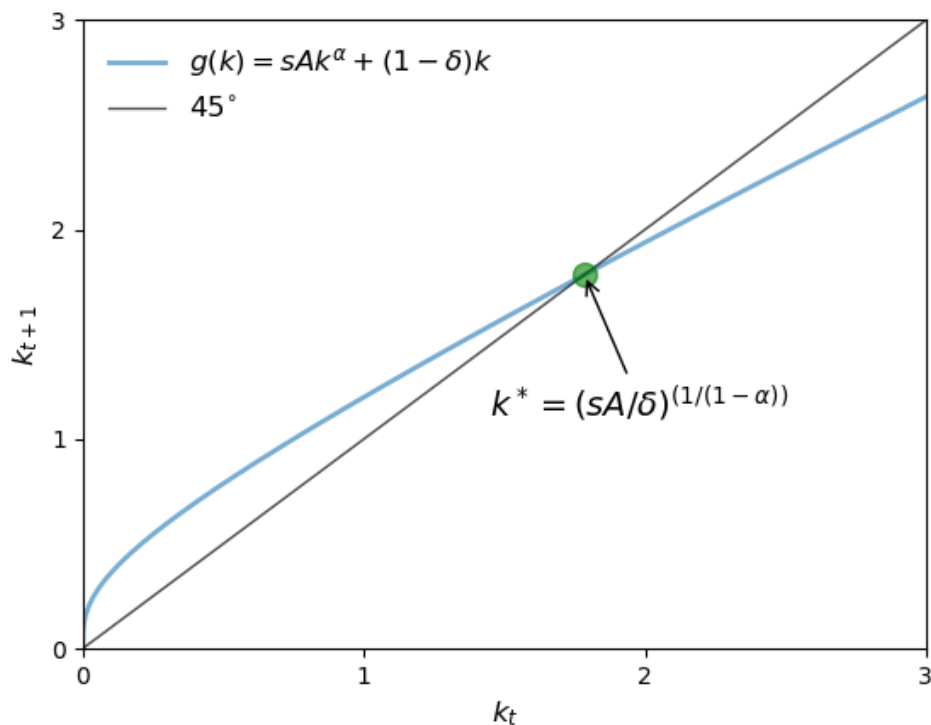
From our graphical analysis, it appears that (k_t) converges to k^*, regardless of initial capital k_0.

This is a form of *global stability*.

The next figure shows three time paths for capital, from three distinct initial conditions, under the parameterization listed above.

At this parameterization, $k^* \approx 1.78$.

Let's define the constants and three distinct initial conditions

```
A, s, alpha, delta = 2, 0.3, 0.3, 0.4
x0 = np.array([.25, 1.25, 3.25])

ts_length = 20
xmin, xmax = 0, ts_length
ymin, ymax = 0, 3.5
```

```
def simulate_ts(x0_values, ts_length):

    k_star = (s * A / delta)**(1/(1-alpha))
    fig, ax = plt.subplots(figsize=[11, 5])
    ax.set_xlim(xmin, xmax)
    ax.set_ylim(ymin, ymax)
```

24.2. A graphical perspective

```
    ts = np.zeros(ts_length)

    # simulate and plot time series
    for x_init in x0_values:
        ts[0] = x_init
        for t in range(1, ts_length):
            ts[t] = g(A, s, alpha, delta, ts[t-1])
        ax.plot(np.arange(ts_length), ts, '-o', ms=4, alpha=0.6,
                label=r'$k_0=%g$' %x_init)
    ax.plot(np.arange(ts_length), np.full(ts_length,k_star),
            alpha=0.6, color='red', label=r'$k^*$')
    ax.legend(fontsize=10)

    ax.set_xlabel(r'$t$', fontsize=14)
    ax.set_ylabel(r'$k_t$', fontsize=14)

    plt.show()
```

```
simulate_ts(x0, ts_length)
```
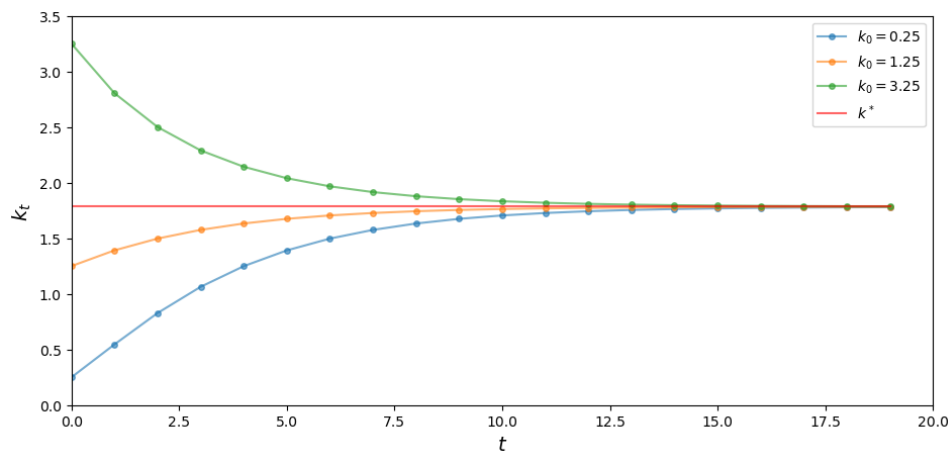

As expected, the time paths in the figure all converge to k^*.

24.3 Growth in continuous time

In this section, we investigate a continuous time version of the Solow–Swan growth model.

We will see how the smoothing provided by continuous time can simplify our analysis.

Recall that the discrete time dynamics for capital are given by $k_{t+1} = sf(k_t) + (1 - \delta)k_t$.

A simple rearrangement gives the rate of change per unit of time:

$$\Delta k_t = sf(k_t) - \delta k_t \quad \text{where} \quad \Delta k_t := k_{t+1} - k_t$$

Taking the time step to zero gives the continuous time limit

$$k'_t = sf(k_t) - \delta k_t \qquad \text{with} \qquad k'_t := \frac{d}{dt}k_t \tag{24.3}$$

Our aim is to learn about the evolution of k_t over time, given an initial stock k_0.

A **steady state** for (??) is a value k^* at which capital is unchanging, meaning $k'_t = 0$ or, equivalently, $sf(k^*) = \delta k^*$.

We assume $f(k) = Ak^\alpha$, so k^* solves $sAk^\alpha = \delta k$.

The solution is the same as the discrete time case—see (??).

The dynamics are represented in the next figure, maintaining the parameterization we used above.

Writing $k'_t = g(k_t)$ with $g(k) = sAk^\alpha - \delta k$, values of k with $g(k) > 0$ imply $k'_t > 0$, so capital is increasing.

When $g(k) < 0$, the opposite occurs. Once again, high marginal returns to savings at low levels of capital combined with low rates of return at high levels of capital combine to yield global stability.

To see this in a figure, let's define the constants

```
A, s, alpha, delta = 2, 0.3, 0.3, 0.4
```

Next we define the function g for growth in continuous time

```
def g_con(A, s, alpha, delta, k):
    return A * s * k**alpha - delta * k
```

```
def plot_gcon(kstar=None):

    k_grid = np.linspace(0, 2.8, 10000)

    fig, ax = plt.subplots(figsize=[11, 5])
    ax.plot(k_grid, g_con(A, s, alpha, delta, k_grid), label='$g(k)$')
    ax.plot(k_grid, 0 * k_grid, label="$k'=0$")
```

<div align="right">(continues on next page)</div>

```
    if kstar:
        fps = (kstar,)

        ax.plot(fps, 0, 'go', ms=10, alpha=0.6)


        ax.annotate(r'$k^* = (sA / \delta)^{(1/(1-\alpha))}$',
                 xy=(kstar, 0),
                 xycoords='data',
                 xytext=(0, 60),
                 textcoords='offset points',
                 fontsize=12,
                 arrowprops=dict(arrowstyle="->"))

    ax.legend(loc='lower left', fontsize=12)

    ax.set_xlabel("$k$",fontsize=10)
    ax.set_ylabel("$k'$", fontsize=10)

    ax.set_xticks((0, 1, 2, 3))
    ax.set_yticks((-0.3, 0, 0.3))

    plt.show()
```

```
kstar = ((s * A) / delta)**(1/(1 - alpha))
plot_gcon(kstar)
```
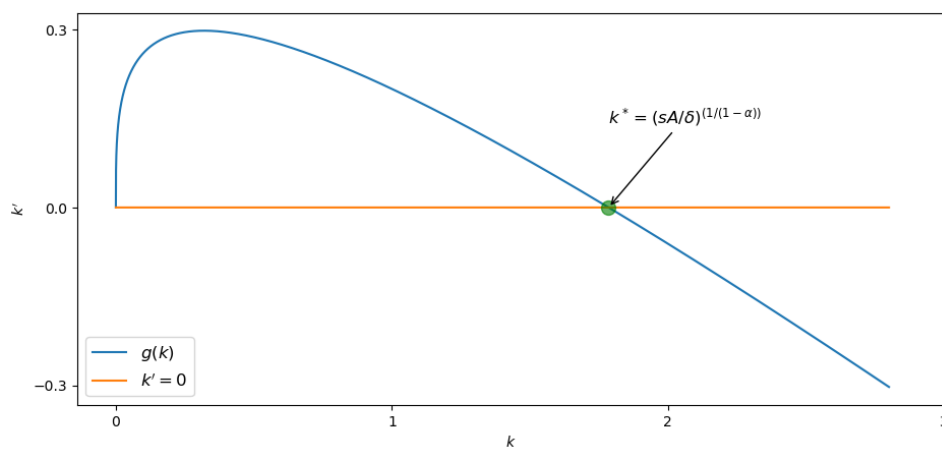

This shows global stability heuristically for a fixed parameterization, but how would we show the same thing formally for a continuum of plausible parameters?

In the discrete time case, a neat expression for k_t is hard to obtain.

In continuous time the process is easier: we can obtain a relatively simple expression for k_t that specifies the entire path.

The first step is to set $x_t := k_t^{1-\alpha}$, so that $x_t' = (1-\alpha)k_t^{-\alpha}k_t'$.

Substituting into $k_t' = sAk_t^\alpha - \delta k_t$ leads to the linear differential equation

$$x_t' = (1-\alpha)(sA - \delta x_t) \tag{24.4}$$

This equation, which is a linear ordinary differential equation, has the solution

$$x_t = \left(k_0^{1-\alpha} - \frac{sA}{\delta}\right)e^{-\delta(1-\alpha)t} + \frac{sA}{\delta}$$

(You can confirm that this function x_t satisfies (**??**) by differentiating it with respect to t.)

Converting back to k_t yields

$$k_t = \left[\left(k_0^{1-\alpha} - \frac{sA}{\delta}\right)e^{-\delta(1-\alpha)t} + \frac{sA}{\delta}\right]^{1/(1-\alpha)} \tag{24.5}$$

Since $\delta > 0$ and $\alpha \in (0,1)$, we see immediately that $k_t \to k^*$ as $t \to \infty$ independent of k_0.

Thus, global stability holds.

24.4 Exercises

Exercise 24.4.1

Plot per capita consumption c at the steady state, as a function of the savings rate s, where $0 \leq s \leq 1$.

Use the Cobb–Douglas specification $f(k) = Ak^\alpha$.

Set $A = 2.0, \alpha = 0.3$, and $\delta = 0.5$

Also, find the approximate value of s that maximizes the $c^*(s)$ and show it in the plot.

Solution to Exercise 24.4.1

Steady state consumption at savings rate s is given by

$$c^*(s) = (1-s)f(k^*) = (1-s)A(k^*)^\alpha$$

```
A = 2.0
alpha = 0.3
delta = 0.5
```

```
s_grid = np.linspace(0, 1, 1000)
k_star = ((s_grid * A) / delta)**(1/(1 - alpha))
c_star = (1 - s_grid) * A * k_star ** alpha
```

Let's find the value of s that maximizes c^* using [scipy.optimize.minimize_scalar](). We will use $-c^*(s)$ since `minimize_scalar` finds the minimum value.

```
from scipy.optimize import minimize_scalar
```

```
def calc_c_star(s):
    k = ((s * A) / delta)**(1/(1 - alpha))
    return - (1 - s) * A * k ** alpha
```

```
return_values = minimize_scalar(calc_c_star, bounds=(0, 1))
s_star_max = return_values.x
c_star_max = -return_values.fun
print(f"Function is maximized at s = {round(s_star_max, 4)}")
```

```
Function is maximized at s = 0.3
```

```
x_s_max = np.array([s_star_max, s_star_max])
y_s_max = np.array([0, c_star_max])


fig, ax = plt.subplots(figsize=[11, 5])


fps = (c_star_max,)


# Highlight the maximum point with a marker
ax.plot((s_star_max, ), (c_star_max,), 'go', ms=8, alpha=0.6)


ax.annotate(r'$s^*$',
         xy=(s_star_max, c_star_max),
         xycoords='data',
         xytext=(20, -50),
         textcoords='offset points',
         fontsize=12,
         arrowprops=dict(arrowstyle="->"))
ax.plot(s_grid, c_star, label=r'$c*(s)$')
ax.plot(x_s_max, y_s_max, alpha=0.5, ls='dotted')
ax.set_xlabel(r'$s$')
ax.set_ylabel(r'$c^*(s)$')
```

```
ax.legend()

plt.show()
```
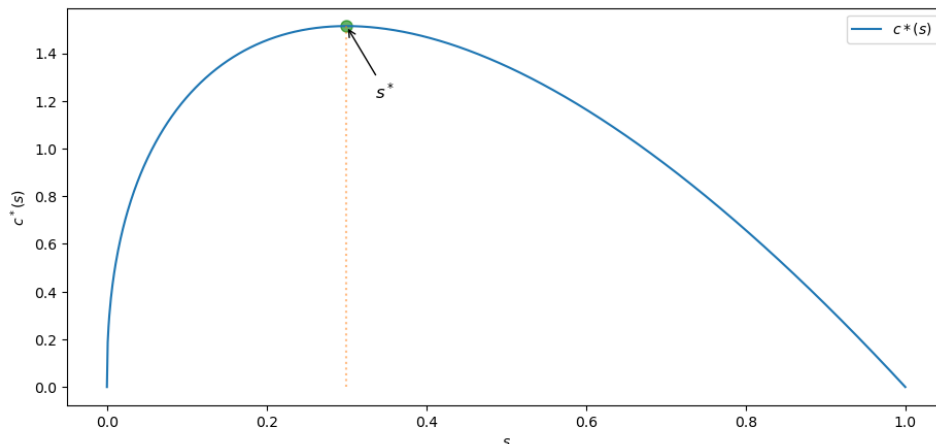

One can also try to solve this mathematically by differentiating $c^*(s)$ and solve for $\frac{d}{ds}c^*(s) = 0$ using sympy.

```
from sympy import solve, Symbol
```

```
s_symbol = Symbol('s', real=True)
k = ((s_symbol * A) / delta)**(1/(1 - alpha))
c = (1 - s_symbol) * A * k ** alpha
```

Let's differentiate c and solve using sympy.solve

```
# Solve using sympy
s_star = solve(c.diff())[0]
print(f"s_star = {s_star}")
```

```
s_star = 0.300000000000000
```

Incidentally, the rate of savings which maximizes steady state level of per capita consumption is called the Golden Rule savings rate.

Exercise 24.4.2

Stochastic Productivity

To bring the Solow – Swan model closer to data, we need to think about handling random fluctuations in aggregate

quantities.

Among other things, this will eliminate the unrealistic prediction that per-capita output $y_t = Ak_t^\alpha$ converges to a constant $y^* := A(k^*)^\alpha$.

We shift to discrete time for the following discussion.

One approach is to replace constant productivity with some stochastic sequence $(A_t)_{t \geq 1}$.

Dynamics are now

$$k_{t+1} = sA_{t+1}f(k_t) + (1 - \delta)k_t \tag{24.6}$$

We suppose f is Cobb–Douglas and (A_t) is IID and lognormal.

Now the long run convergence obtained in the deterministic case breaks down, since the system is hit with new shocks at each point in time.

Consider $A = 2.0, s = 0.6, \alpha = 0.3$, and $\delta = 0.5$

Generate and plot the time series k_t.

Solution to Exercise 24.4.2

Let's define the constants for lognormal distribution and initial values used for simulation

```python
# Define the constants
sig = 0.2
mu = np.log(2) - sig**2 / 2
A = 2.0
s = 0.6
alpha = 0.3
delta = 0.5
x0 = [.25, 3.25] # list of initial values used for simulation
```

Let's define the function *k_next* to find the next value of k

```python
def lgnorm():
    return np.exp(mu + sig * np.random.randn())


def k_next(s, alpha, delta, k):
    return lgnorm() * s * k**alpha + (1 - delta) * k
```

```python
def ts_plot(x_values, ts_length):
    fig, ax = plt.subplots(figsize=[11, 5])
```

```
    ts = np.zeros(ts_length)

    # simulate and plot time series
    for x_init in x_values:
        ts[0] = x_init
        for t in range(1, ts_length):
            ts[t] = k_next(s, alpha, delta, ts[t-1])
        ax.plot(np.arange(ts_length), ts, '-o', ms=4,
                alpha=0.6, label=r'$k_0=%g$' %x_init)

    ax.legend(loc='best', fontsize=10)

    ax.set_xlabel(r'$t$', fontsize=12)
    ax.set_ylabel(r'$k_t$', fontsize=12)



    plt.show()
```

```
ts_plot(x0, 50)
```
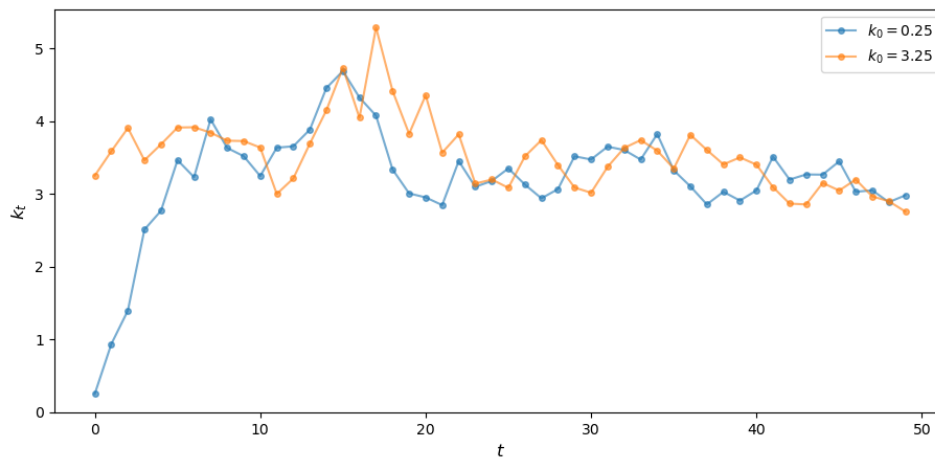
THE COBWEB MODEL

The cobweb model is a model of prices and quantities in a given market, and how they evolve over time.

25.1 Overview

The cobweb model dates back to the 1930s and, while simple, it remains significant because it shows the fundamental importance of *expectations*.

To give some idea of how the model operates, and why expectations matter, imagine the following scenario.

There is a market for soybeans, say, where prices and traded quantities depend on the choices of buyers and sellers.

The buyers are represented by a demand curve —they buy more at low prices and less at high prices.

The sellers have a supply curve —they wish to sell more at high prices and less at low prices.

However, the sellers (who are farmers) need time to grow their crops.

Suppose now that the price is currently high.

Seeing this high price, and perhaps expecting that the high price will remain for some time, the farmers plant many fields with soybeans.

Next period the resulting high supply floods the market, causing the price to drop.

Seeing this low price, the farmers now shift out of soybeans, restricting supply and causing the price to climb again.

You can imagine how these dynamics could cause cycles in prices and quantities that persist over time.

The cobweb model puts these ideas into equations so we can try to quantify them, and to study conditions under which cycles persist (or disappear).

In this lecture, we investigate and simulate the basic model under different assumptions regarding the way that producers form expectations.

Our discussion and simulations draw on high quality lectures by Cars Hommes.

We will use the following imports.

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

25.2 History

Early papers on the cobweb cycle include [Waugh, 1964] and [Harlow, 1960].

The paper [Harlow, 1960] uses the cobweb theorem to explain the prices of hog in the US over 1920–1950.

The next plot replicates part of Figure 2 from that paper, which plots the price of hogs at yearly frequency.

Notice the cyclical price dynamics, which match the kind of cyclical soybean price dynamics discussed above.

```
hog_prices = [55, 57, 80, 70, 60, 65, 72, 65, 51, 49, 45, 80, 85,
              78, 80, 68, 52, 65, 83, 78, 60, 62, 80, 87, 81, 70,
              69, 65, 62, 85, 87, 65, 63, 75, 80, 62]
years = np.arange(1924, 1960)
fig, ax = plt.subplots()
ax.plot(years, hog_prices, '-o', ms=4, label='hog price')
ax.set_xlabel('year')
ax.set_ylabel('dollars')
ax.legend()
ax.grid()
plt.show()
```
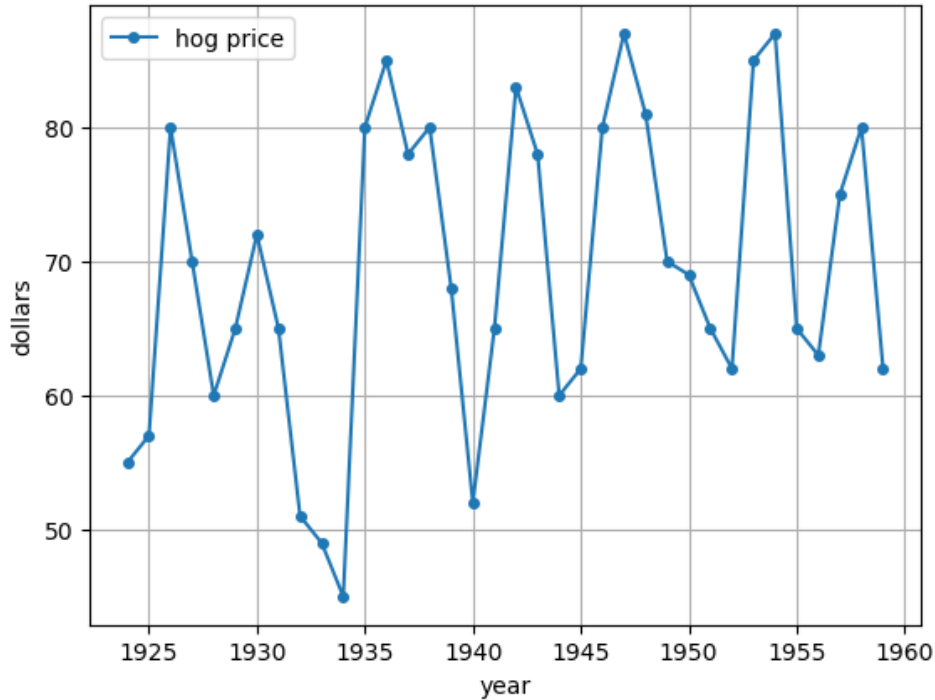
25.3 The model

Let's return to our discussion of a hypothetical soybean market, where price is determined by supply and demand.

We suppose that demand for soybeans is given by

$$D(p_t) = a - bp_t$$

where a, b are nonnegative constants and p_t is the spot (i.e, current market) price at time t.

($D(p_t)$ is the quantity demanded in some fixed unit, such as thousands of tons.)

Because the crop of soybeans for time t is planted at $t - 1$, supply of soybeans at time t depends on *expected* prices at time t, which we denote p_t^e.

We suppose that supply is nonlinear in expected prices, and takes the form

$$S(p_t^e) = \tanh(\lambda(p_t^e - c)) + d$$

where λ is a positive constant, c, d are nonnegative constants and tanh is a type of hyperbolic function.

Let's make a plot of supply and demand for particular choices of the parameter values.

First we store the parameters in a class and define the functions above as methods.

```python
class Market:

    def __init__(self,
                 a=8,       # demand parameter
                 b=1,       # demand parameter
                 c=6,       # supply parameter
                 d=1,       # supply parameter
                 λ=2.0):    # supply parameter
        self.a, self.b, self.c, self.d = a, b, c, d
        self.λ = λ

    def demand(self, p):
        a, b = self.a, self.b
        return a - b * p

    def supply(self, p):
        c, d, λ = self.c, self.d, self.λ
        return np.tanh(λ * (p - c)) + d
```

Now let's plot.

```python
p_grid = np.linspace(5, 8, 200)
m = Market()
fig, ax = plt.subplots()

ax.plot(p_grid, m.demand(p_grid), label="$D$")
ax.plot(p_grid, m.supply(p_grid), label="$S$")
ax.set_xlabel("price")
ax.set_ylabel("quantity")
ax.legend()

plt.show()
```
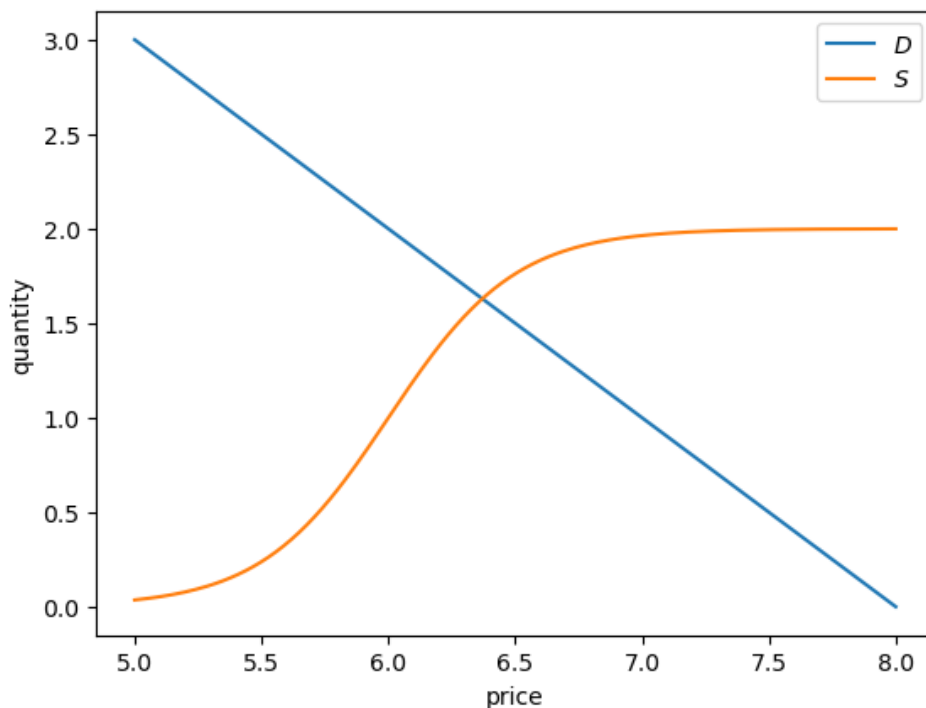
Market equilibrium requires that supply equals demand, or

$$a - bp_t = S(p_t^e)$$

Rewriting in terms of p_t gives

$$p_t = -\frac{1}{b}[S(p_t^e) - a]$$

Finally, to complete the model, we need to describe how price expectations are formed.

We will assume that expected prices at time t depend on past prices.

In particular, we suppose that

$$p_t^e = f(p_{t-1}, p_{t-2}) \tag{25.1}$$

where f is some function.

Thus, we are assuming that producers expect the time-t price to be some function of lagged prices, up to 2 lags.

(We could of course add additional lags and readers are encouraged to experiment with such cases.)

Combining the last two equations gives the dynamics for prices:

$$p_t = -\frac{1}{b}[S(f(p_{t-1}, p_{t-2})) - a] \tag{25.2}$$

The price dynamics depend on the parameter values and also on the function f that determines how producers form expectations.

25.4 Naive expectations

To go further in our analysis we need to specify the function f; that is, how expectations are formed.

Let's start with naive expectations, which refers to the case where producers expect the next period spot price to be whatever the price is in the current period.

In other words,

$$p_t^e = p_{t-1}$$

Using (**??**), we then have

$$p_t = -\frac{1}{b}[S(p_{t-1}) - a]$$

We can write this as

$$p_t = g(p_{t-1})$$

where g is the function defined by

$$g(p) = -\frac{1}{b}[S(p) - a] \tag{25.3}$$

Here we represent the function g

```python
def g(model, current_price):
    """
    Function to find the next price given the current price
    and Market model
    """
    a, b = model.a, model.b
    next_price = - (model.supply(current_price) - a) / b
    return next_price
```

Let's try to understand how prices will evolve using a 45-degree diagram, which is a tool for studying one-dimensional dynamics.

The function `plot45` defined below helps us draw the 45-degree diagram.

Now we can set up a market and plot the 45-degree diagram.

```python
m = Market()
```

```python
plot45(m, 0, 9, 2, num_arrows=3)
```
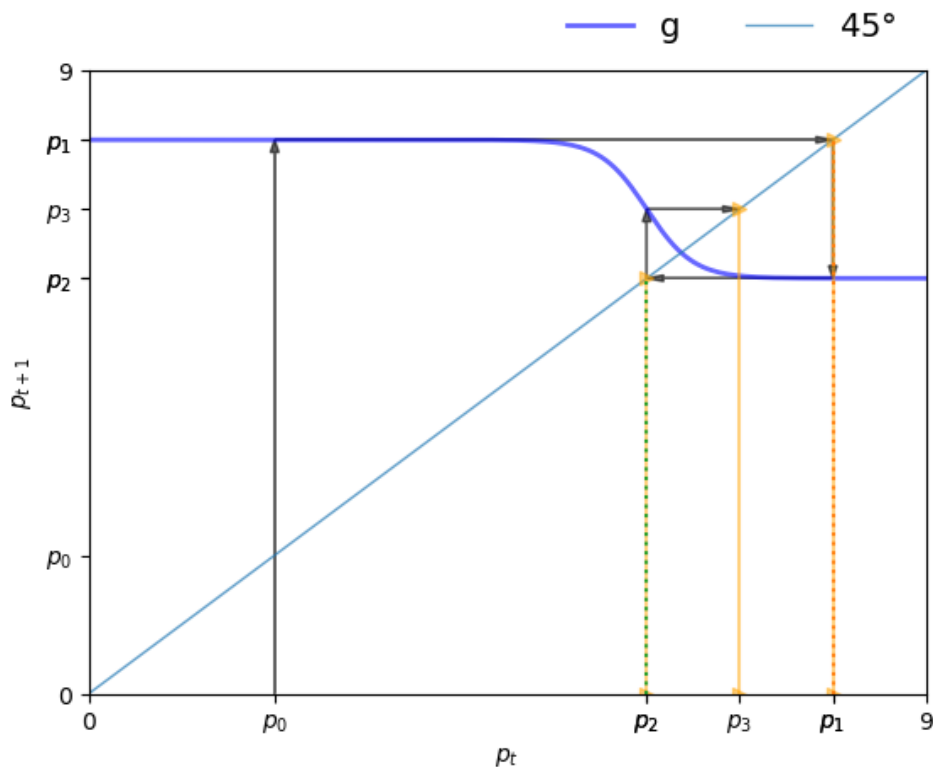
The plot shows the function g defined in (**??**) and the 45-degree line.

Think of p_t as a value on the horizontal axis.

Since $p_{t+1} = g(p_t)$, we use the graph of g to see p_{t+1} on the vertical axis.

Clearly,

- If g lies above the 45-degree line at p_t, then we have $p_{t+1} > p_t$.

- If g lies below the 45-degree line at p_t, then we have $p_{t+1} < p_t$.

- If g hits the 45-degree line at p_t, then we have $p_{t+1} = p_t$, so p_t is a steady state.

Consider the sequence of prices starting at p_0, as shown in the figure.

We find p_1 on the vertical axis and then shift it to the horizontal axis using the 45-degree line (where values on the two axes are equal).

Then from p_1 we obtain p_2 and continue.

We can see the start of a cycle.

To confirm this, let's plot a time series.

```python
def ts_plot_price(model,            # Market model
                  p0,               # Initial price
                  y_a=3, y_b= 12,   # Controls y-axis
                  ts_length=10):    # Length of time series
    """
    Function to simulate and plot the time series of price.

    """
    fig, ax = plt.subplots()
    ax.set_xlabel(r'$t$', fontsize=12)
    ax.set_ylabel(r'$p_t$', fontsize=12)
    p = np.empty(ts_length)
    p[0] = p0
    for t in range(1, ts_length):
        p[t] = g(model, p[t-1])
    ax.plot(np.arange(ts_length),
            p,
            'bo-',
            alpha=0.6,
            lw=2,
            label=r'$p_t$')
    ax.legend(loc='best', fontsize=10)
    ax.set_ylim(y_a, y_b)
    ax.set_xticks(np.arange(ts_length))
    plt.show()
```

```python
ts_plot_price(m, 4, ts_length=15)
```
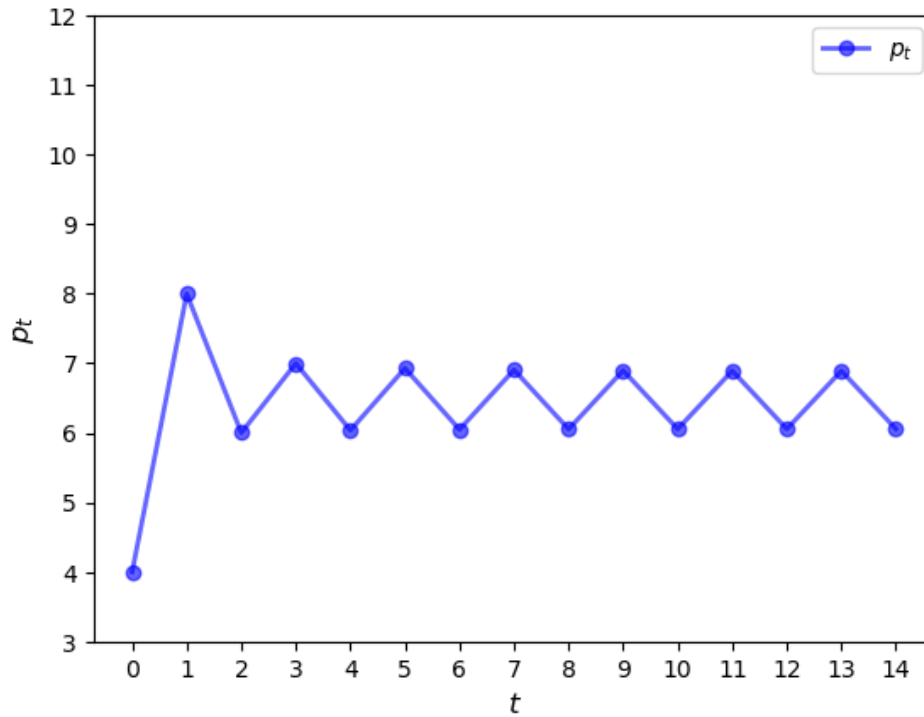
We see that a cycle has formed and the cycle is persistent.

(You can confirm this by plotting over a longer time horizon.)

The cycle is "stable", in the sense that prices converge to it from most starting conditions.

For example,

```
ts_plot_price(m, 10, ts_length=15)
```
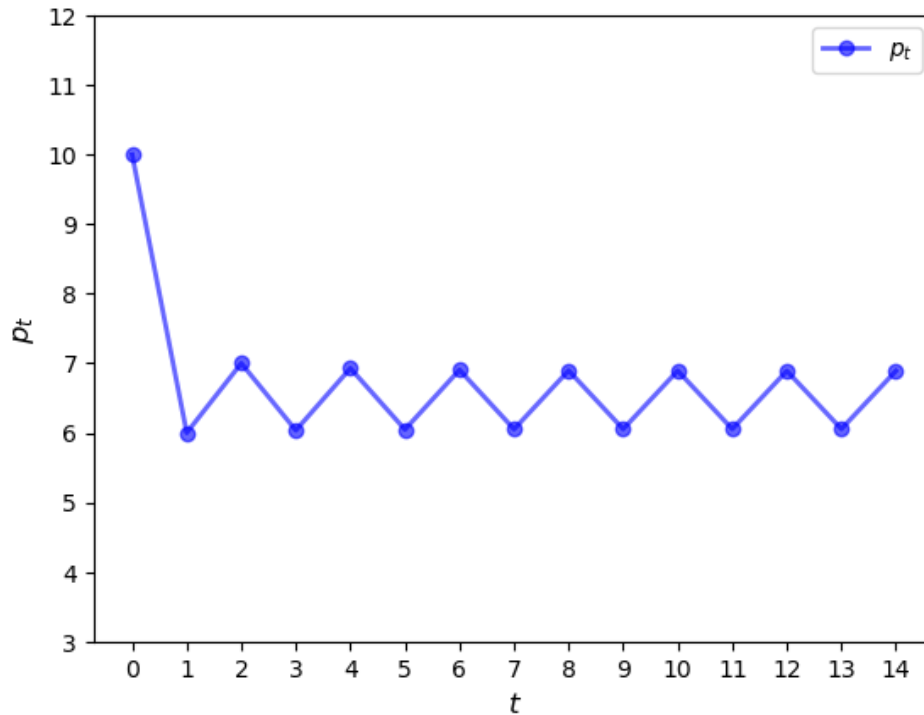
25.5 Adaptive expectations

Naive expectations are quite simple and also important in driving the cycle that we found.

What if expectations are formed in a different way?

Next we consider adaptive expectations.

This refers to the case where producers form expectations for the next period price as a weighted average of their last guess and the current spot price.

That is,

$$p_t^e = \alpha p_{t-1} + (1 - \alpha)p_{t-1}^e \qquad (0 \le \alpha \le 1) \tag{25.4}$$

Another way to write this is

$$p_t^e = p_{t-1}^e + \alpha(p_{t-1} - p_{t-1}^e) \tag{25.5}$$

This equation helps to show that expectations shift

1. up when prices last period were above expectations

2. down when prices last period were below expectations

Using (**??**), we obtain the dynamics

$$p_t = -\frac{1}{b}[S(\alpha p_{t-1} + (1 - \alpha)p_{t-1}^e) - a]$$

Let's try to simulate the price and observe the dynamics using different values of α.

```python
def find_next_price_adaptive(model, curr_price_exp):
    """
    Function to find the next price given the current price expectation
    and Market model
    """
    return - (model.supply(curr_price_exp) - model.a) / model.b
```

The function below plots price dynamics under adaptive expectations for different values of α.

```python
def ts_price_plot_adaptive(model, p0, ts_length=10, α=[1.0, 0.9, 0.75]):
    fig, axs = plt.subplots(1, len(α), figsize=(12, 5))
    for i_plot, a in enumerate(α):
        pe_last = p0
        p_values = np.empty(ts_length)
        p_values[0] = p0
        for i in range(1, ts_length):
            p_values[i] = find_next_price_adaptive(model, pe_last)
            pe_last = a*p_values[i] + (1 - a)*pe_last

        axs[i_plot].plot(np.arange(ts_length), p_values)
        axs[i_plot].set_title(r'$\alpha={}$'.format(a))
        axs[i_plot].set_xlabel('t')
        axs[i_plot].set_ylabel('price')
    plt.show()
```

Let's call the function with prices starting at $p_0 = 5$.

```python
ts_price_plot_adaptive(m, 5, ts_length=30)
```
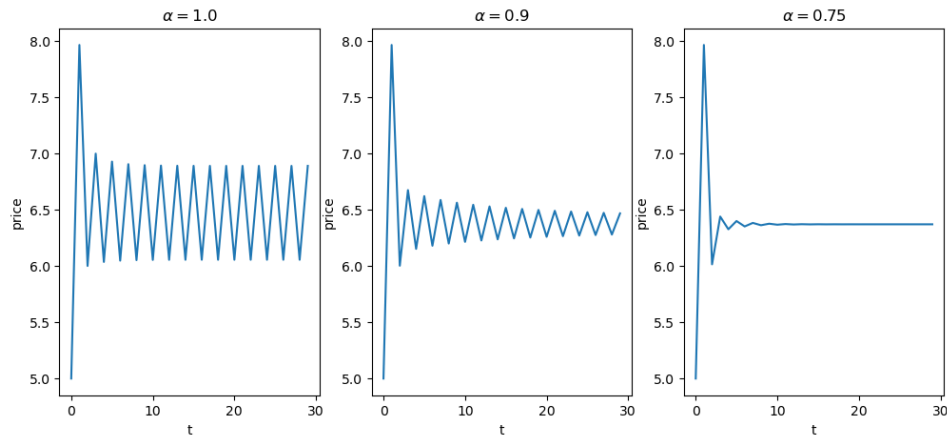
Note that if $\alpha = 1$, then adaptive expectations are just naive expectation.

Decreasing the value of α shifts more weight to the previous expectations, which stabilizes expected prices.

This increased stability can be seen in the figures.

25.6 Exercises

Exercise 25.6.1

Using the default `Market` class and naive expectations, plot a time series simulation of supply (rather than the price).

Show, in particular, that supply also cycles.

Solution to Exercise 25.6.1

```python
def ts_plot_supply(model, p0, ts_length=10):
    """
    Function to simulate and plot the supply function
    given the initial price.
    """
    pe_last = p0
    s_values = np.empty(ts_length)
    for i in range(ts_length):
        # store quantity
        s_values[i] = model.supply(pe_last)
        # update price
        pe_last = - (s_values[i] - model.a) / model.b
```

```
fig, ax = plt.subplots()
ax.plot(np.arange(ts_length),
        s_values,
        'bo-',
        alpha=0.6,
        lw=2,
        label=r'supply')

ax.legend(loc='best', fontsize=10)
ax.set_xticks(np.arange(ts_length))
ax.set_xlabel("time")
ax.set_ylabel("quantity")
plt.show()
```

```
m = Market()
ts_plot_supply(m, 5, 15)
```
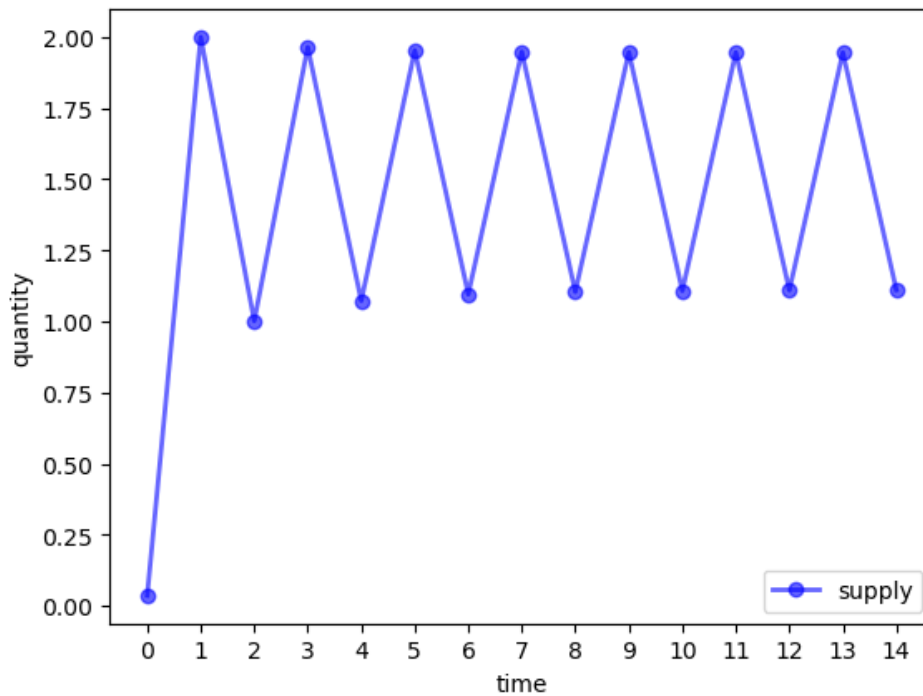
Exercise 25.6.2

Backward looking average expectations

Backward looking average expectations refers to the case where producers form expectations for the next period price as a linear combination of their last guess and the second last guess.

That is,

$$p_t^e = \alpha p_{t-1} + (1 - \alpha)p_{t-2} \tag{25.6}$$

Simulate and plot the price dynamics for $\alpha \in \{0.1, 0.3, 0.5, 0.8\}$ where $p_0 = 1$ and $p_1 = 2.5$.

Solution to Exercise 25.6.2

```python
def find_next_price_blae(model, curr_price_exp):
    """
    Function to find the next price given the current price expectation
    and Market model
    """
    return - (model.supply(curr_price_exp) - model.a) / model.b
```

```python
def ts_plot_price_blae(model, p0, p1, alphas, ts_length=15):
    """
    Function to simulate and plot the time series of price
    using backward looking average expectations.
    """
    fig, axes = plt.subplots(len(alphas), 1, figsize=(8, 16))

    for ax, a in zip(axes.flatten(), alphas):
        p = np.empty(ts_length)
        p[0] = p0
        p[1] = p1
        for t in range(2, ts_length):
            pe = a*p[t-1] + (1 - a)*p[t-2]
            p[t] = -(model.supply(pe) - model.a) / model.b
        ax.plot(np.arange(ts_length),
                p,
                'o-',
                alpha=0.6,
                label=r'$\alpha={}$'.format(a))
        ax.legend(loc='best', fontsize=10)
        ax.set_xlabel(r'$t$', fontsize=12)
        ax.set_ylabel(r'$p_t$', fontsize=12)
    plt.show()
```

```
m = Market()
ts_plot_price_blae(m,
                   p0=5,
                   p1=6,
                   alphas=[0.1, 0.3, 0.5, 0.8],
                   ts_length=20)
```
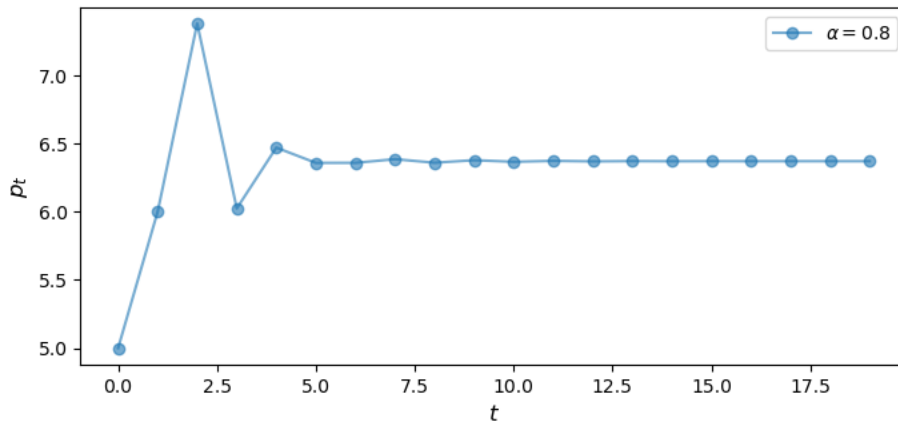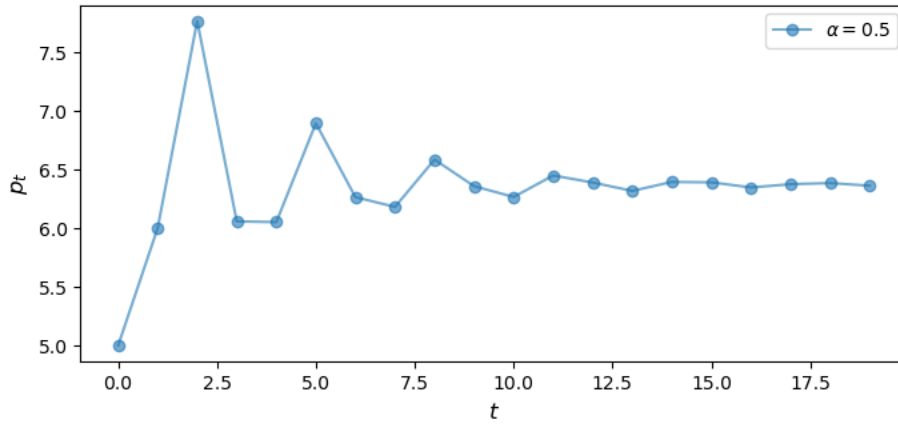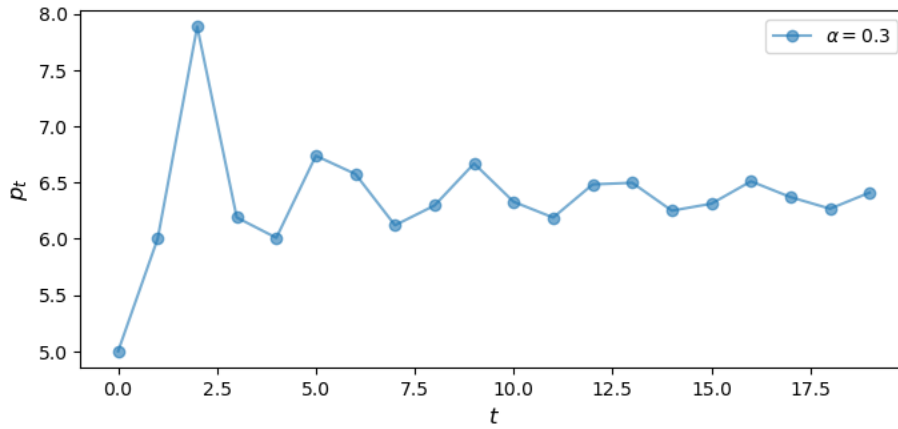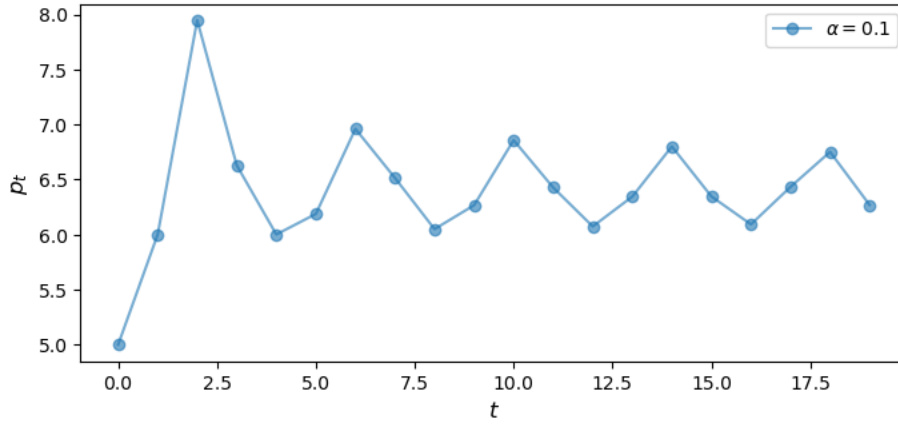
THE OVERLAPPING GENERATIONS MODEL

In this lecture we study the famous overlapping generations (OLG) model, which is used by policy makers and researchers to examine

- fiscal policy

- monetary policy

- long-run growth

and many other topics.

The first rigorous version of the OLG model was developed by Paul Samuelson [Samuelson, 1958].

Our aim is to gain a good understanding of a simple version of the OLG model.

26.1 Overview

The dynamics of the OLG model are quite similar to those of the Solow-Swan growth model.

At the same time, the OLG model adds an important new feature: the choice of how much to save is endogenous.

To see why this is important, suppose, for example, that we are interested in predicting the effect of a new tax on long-run growth.

We could add a tax to the Solow-Swan model and look at the change in the steady state.

But this ignores the fact that households will change their savings and consumption behavior when they face the new tax rate.

Such changes can substantially alter the predictions of the model.

Hence, if we care about accurate predictions, we should model the decision problems of the agents.

In particular, households in the model should decide how much to save and how much to consume, given the environment that they face (technology, taxes, prices, etc.)

The OLG model takes up this challenge.

We will present a simple version of the OLG model that clarifies the decision problem of households and studies the implications for long-run growth.

Let's start with some imports.

```python
import numpy as np
from scipy import optimize
from collections import namedtuple
import matplotlib.pyplot as plt
```

26.2 Environment

We assume that time is discrete, so that $t = 0, 1, ...$

An individual born at time t lives for two periods, t and $t + 1$.

We call an agent

- "young" during the first period of their lives and

- "old" during the second period of their lives.

Young agents work, supplying labor and earning labor income.

They also decide how much to save.

Old agents do not work, so all income is financial.

Their financial income is from interest on their savings from wage income, which is then combined with the labor of the new young generation at $t + 1$.

The wage and interest rates are determined in equilibrium by supply and demand.

To make the algebra slightly easier, we are going to assume a constant population size.

We normalize the constant population size in each period to 1.

We also suppose that each agent supplies one "unit" of labor hours, so total labor supply is 1.

26.3 Supply of capital

First let's consider the household side.

26.3.1 Consumer's problem

Suppose that utility for individuals born at time t takes the form

$$U_t = u(c_t) + \beta u(c_{t+1}) \tag{26.1}$$

Here

- $u : \mathbb{R}_+ \to \mathbb{R}$ is called the "flow" utility function
- $\beta \in (0, 1)$ is the discount factor
- c_t is time t consumption of the individual born at time t
- c_{t+1} is time $t + 1$ consumption of the same individual

We assume that u is strictly increasing.

Savings behavior is determined by the optimization problem

$$\max_{c_t, c_{t+1}} \{u(c_t) + \beta u(c_{t+1})\} \tag{26.2}$$

subject to

$$c_t + s_t \leq w_t \quad \text{and} \quad c_{t+1} \leq R_{t+1} s_t$$

Here

- s_t is savings by an individual born at time t
- w_t is the wage rate at time t
- R_{t+1} is the gross interest rate on savings invested at time t, paid at time $t + 1$

Since u is strictly increasing, both of these constraints will hold as equalities at the maximum.

Using this fact and substituting s_t from the first constraint into the second we get $c_{t+1} = R_{t+1}(w_t - c_t)$.

The first-order condition for a maximum can be obtained by plugging c_{t+1} into the objective function, taking the derivative with respect to c_t, and setting it to zero.

This leads to the **Euler equation** of the OLG model, which describes the optimal intertemporal consumption dynamics:

$$u'(c_t) = \beta R_{t+1} u'(R_{t+1}(w_t - c_t)) \tag{26.3}$$

From the first constraint we get $c_t = w_t - s_t$, so the Euler equation can also be expressed as

$$u'(w_t - s_t) = \beta R_{t+1} u'(R_{t+1} s_t) \tag{26.4}$$

Suppose that, for each w_t and R_{t+1}, there is exactly one s_t that solves *(26.3.4)*.

Then savings can be written as a fixed function of w_t and R_{t+1}.

We write this as

$$s_t = s(w_t, R_{t+1}) \tag{26.5}$$

The precise form of the function s will depend on the choice of flow utility function u.

Together, w_t and R_{t+1} represent the *prices* in the economy (price of labor and rental rate of capital).

Thus, *(26.3.5)* states the quantity of savings given prices.

26.3.2 Example: log preferences

In the special case $u(c) = \log c$, the Euler equation simplifies to $s_t = \beta(w_t - s_t)$.

Solving for saving, we get

$$s_t = s(w_t, R_{t+1}) = \frac{\beta}{1 + \beta} w_t \tag{26.6}$$

In this special case, savings does not depend on the interest rate.

26.3.3 Savings and investment

Since the population size is normalized to 1, s_t is also total savings in the economy at time t.

In our closed economy, there is no foreign investment, so net savings equals total investment, which can be understood as supply of capital to firms.

In the next section we investigate demand for capital.

Equating supply and demand will allow us to determine equilibrium in the OLG economy.

26.4 Demand for capital

First we describe the firm's problem and then we write down an equation describing demand for capital given prices.

26.4.1 Firm's problem

For each integer $t \geq 0$, output y_t in period t is given by the **Cobb-Douglas production function**

$$y_t = k_t^\alpha \ell_t^{1-\alpha} \tag{26.7}$$

Here k_t is capital, ℓ_t is labor, and α is a parameter (sometimes called the "output elasticity of capital").

The profit maximization problem of the firm is

$$\max_{k_t, \ell_t} \{k_t^\alpha \ell_t^{1-\alpha} - R_t k_t - w_t \ell_t\} \tag{26.8}$$

The first-order conditions are obtained by taking the derivative of the objective function with respect to capital and labor respectively and setting them to zero:

$$(1-\alpha)(k_t/\ell_t)^\alpha = w_t \quad \text{and} \quad \alpha(k_t/\ell_t)^{\alpha-1} = R_t$$

26.4.2 Demand

Using our assumption $\ell_t = 1$ allows us to write

$$w_t = (1-\alpha)k_t^\alpha \tag{26.9}$$

and

$$R_t = \alpha k_t^{\alpha-1} \tag{26.10}$$

Rearranging *(26.4.4)* gives the aggregate demand for capital at time $t+1$

$$k^d(R_{t+1}) := \left(\frac{\alpha}{R_{t+1}}\right)^{1/(1-\alpha)} \tag{26.11}$$

In Python code this is

```python
def capital_demand(R, α):
    return (α/R) ** (1/(1-α))
```

```python
def capital_supply(R, β, w):
    R = np.ones_like(R)
    return R * (β / (1 + β)) * w
```

The next figure plots the supply of capital, as in *(26.3.6)*, as well as the demand for capital, as in *(26.4.5)*, as functions of the interest rate R_{t+1}.

(For the special case of log utility, supply does not depend on the interest rate, so we have a constant function.)

26.5 Equilibrium

In this section we derive equilibrium conditions and investigate an example.

26.5.1 Equilibrium conditions

In equilibrium, savings at time t equals investment at time t, which equals capital supply at time $t + 1$.

Equilibrium is computed by equating these quantities, setting

$$s(w_t, R_{t+1}) = k^d(R_{t+1}) = \left(\frac{\alpha}{R_{t+1}} \right)^{1/(1-\alpha)} \tag{26.12}$$

In principle, we can now solve for the equilibrium price R_{t+1} given w_t.

(In practice, we first need to specify the function u and hence s.)

When we solve this equation, which concerns time $t + 1$ outcomes, time t quantities are already determined, so we can treat w_t as a constant.

From equilibrium R_{t+1} and *(26.4.5)*, we can obtain the equilibrium quantity k_{t+1}.

26.5.2 Example: log utility

In the case of log utility, we can use *(26.5.1)* and *(26.3.6)* to obtain

$$\frac{\beta}{1 + \beta} w_t = \left(\frac{\alpha}{R_{t+1}} \right)^{1/(1-\alpha)} \tag{26.13}$$

Solving for the equilibrium interest rate gives

$$R_{t+1} = \alpha \left(\frac{\beta}{1 + \beta} w_t \right)^{\alpha - 1} \tag{26.14}$$

In Python we can compute this via

```
def equilibrium_R_log_utility(α, β, w):
    R = α * ( (β * w) / (1 + β)) ** (α - 1)
    return R
```

In the case of log utility, since capital supply does not depend on the interest rate, the equilibrium quantity is fixed by supply.

That is,

$$k_{t+1} = s(w_t, R_{t+1}) = \frac{\beta}{1 + \beta} w_t \tag{26.15}$$

Let's redo our plot above but now inserting the equilibrium quantity and price.

```
R_vals = np.linspace(0.3, 1)
α, β = 0.5, 0.9
w = 2.0

fig, ax = plt.subplots()

ax.plot(R_vals, capital_demand(R_vals, α),
        label="aggregate demand")
ax.plot(R_vals, capital_supply(R_vals, β, w),
        label="aggregate supply")

R_e = equilibrium_R_log_utility(α, β, w)
k_e = (β / (1 + β)) * w

ax.plot(R_e, k_e, 'o',label='equilibrium')

ax.set_xlabel("$R_{t+1}$")
ax.set_ylabel("$k_{t+1}$")
ax.legend()
plt.show()
```
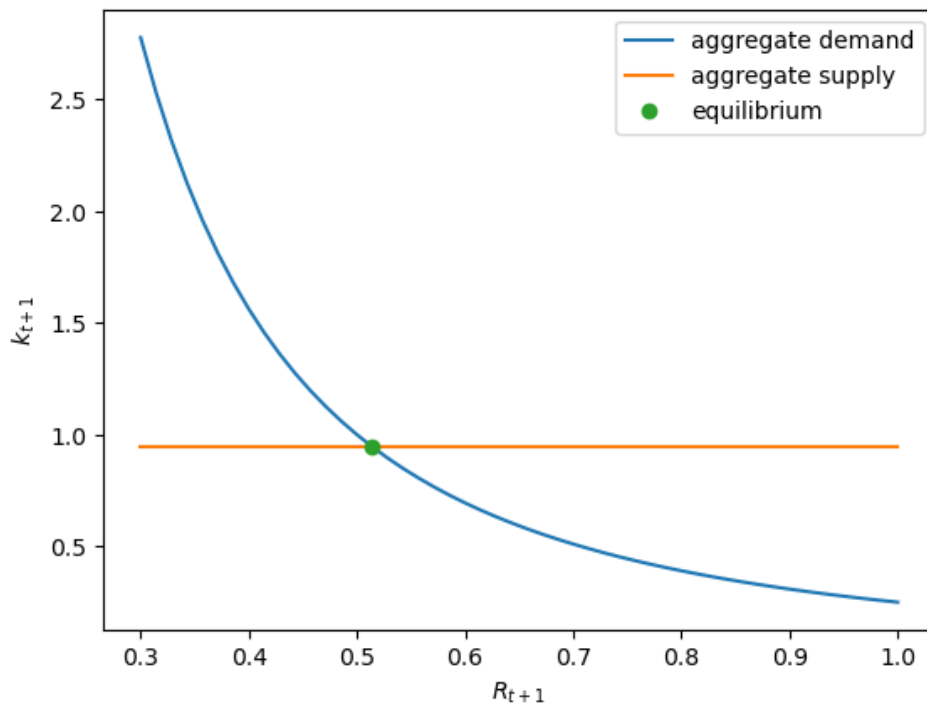
26.6 Dynamics

In this section we discuss dynamics.

For now we will focus on the case of log utility, so that the equilibrium is determined by *(26.5.4)*.

26.6.1 Evolution of capital

The discussion above shows how equilibrium k_{t+1} is obtained given w_t.

From *(26.4.3)* we can translate this into k_{t+1} as a function of k_t

In particular, since $w_t = (1 - \alpha)k_t^\alpha$, we have

$$k_{t+1} = \frac{\beta}{1 + \beta}(1 - \alpha)(k_t)^\alpha \tag{26.16}$$

If we iterate on this equation, we get a sequence for capital stock.

Let's plot the 45-degree diagram of these dynamics, which we write as

$$k_{t+1} = g(k_t) \quad \text{where } g(k) := \frac{\beta}{1 + \beta}(1 - \alpha)(k)^\alpha$$

```python
def k_update(k, α, β):
    return β * (1 - α) * k**α / (1 + β)
```

```python
α, β = 0.5, 0.9
kmin, kmax = 0, 0.1
n = 1000
k_grid = np.linspace(kmin, kmax, n)
k_grid_next = k_update(k_grid, α, β)

fig, ax = plt.subplots(figsize=(6, 6))

ymin, ymax = np.min(k_grid_next), np.max(k_grid_next)

ax.plot(k_grid, k_grid_next,  lw=2, alpha=0.6, label='$g$')
ax.plot(k_grid, k_grid, 'k-', lw=1, alpha=0.7, label='$45^{\circ}$')


ax.legend(loc='upper left', frameon=False, fontsize=12)
ax.set_xlabel('$k_t$', fontsize=12)
ax.set_ylabel('$k_{t+1}$', fontsize=12)

plt.show()
```
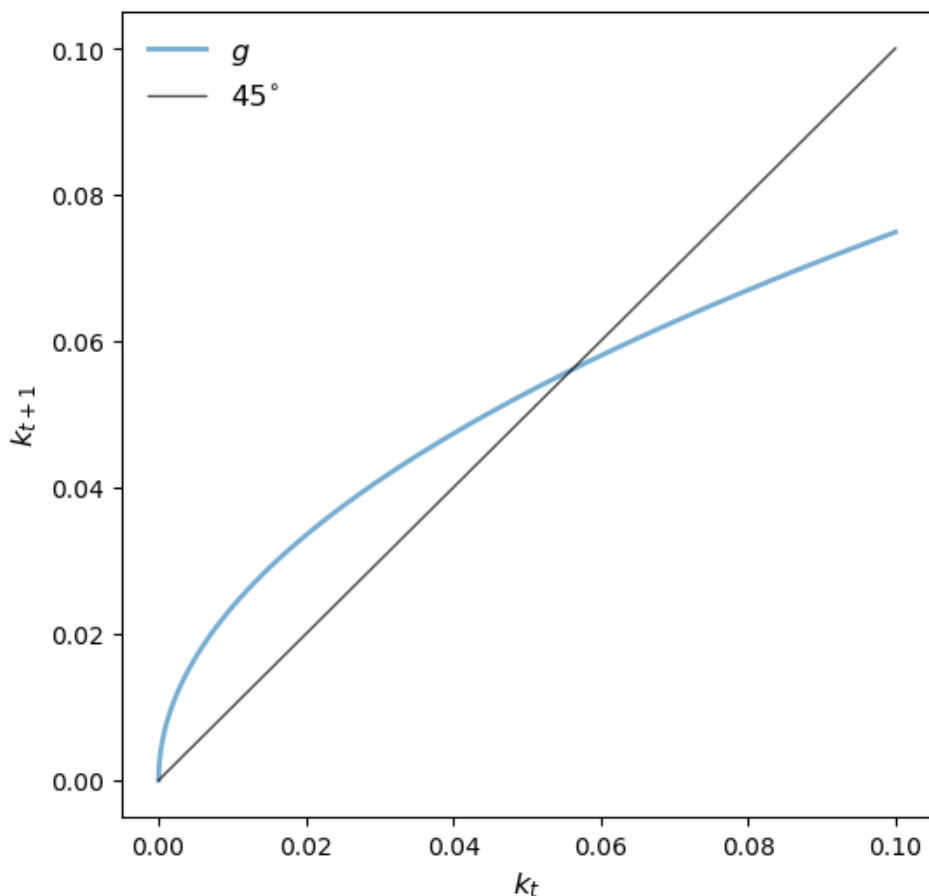
26.6.2 Steady state (log case)

The diagram shows that the model has a unique positive steady state, which we denote by k^*.

We can solve for k^* by setting $k^* = g(k^*)$, or

$$k^* = \frac{\beta(1-\alpha)(k^*)^\alpha}{(1+\beta)} \tag{26.17}$$

Solving this equation yields

$$k^* = \left(\frac{\beta(1-\alpha)}{1+\beta}\right)^{1/(1-\alpha)} \tag{26.18}$$

We can get the steady state interest rate from *(26.4.4)*, which yields

$$R^* = \alpha(k^*)^{\alpha-1} = \frac{\alpha}{1-\alpha}\frac{1+\beta}{\beta}$$

In Python we have

```
k_star = ((β * (1 - α))/(1 + β))**(1/(1-α))
R_star = (α/(1 - α)) * ((1 + β) / β)
```

26.6.3 Time series

The 45-degree diagram above shows that time series of capital with positive initial conditions converge to this steady state.

Let's plot some time series that visualize this.

```
ts_length = 25
k_series = np.empty(ts_length)
k_series[0] = 0.02
for t in range(ts_length - 1):
    k_series[t+1] = k_update(k_series[t], α, β)

fig, ax = plt.subplots()
ax.plot(k_series, label="capital series")
ax.plot(range(ts_length), np.full(ts_length, k_star), 'k--', label="$k^*$")
ax.set_ylim(0, 0.1)
ax.set_ylabel("capital")
ax.set_xlabel("$t$")
ax.legend()
plt.show()
```
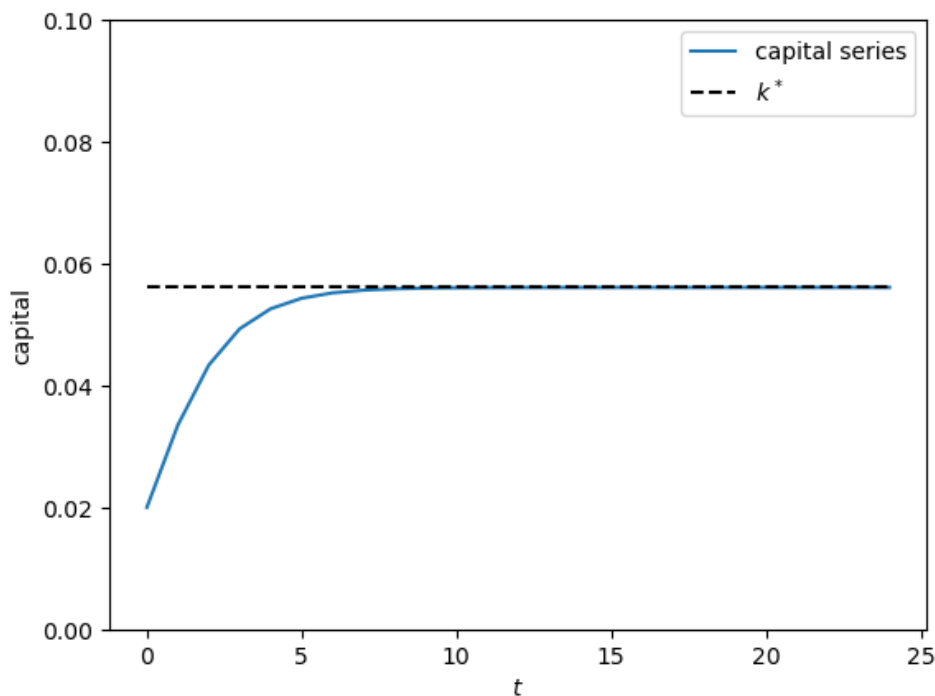
If you experiment with different positive initial conditions, you will see that the series always converges to k^*.

Below we also plot the gross interest rate over time.

```
R_series = α * k_series**(α - 1)

fig, ax = plt.subplots()
ax.plot(R_series, label="gross interest rate")
ax.plot(range(ts_length), np.full(ts_length, R_star), 'k--', label="$R^*$")
ax.set_ylim(0, 4)
ax.set_ylabel("gross interest rate")
ax.set_xlabel("$t$")
ax.legend()
plt.show()
```
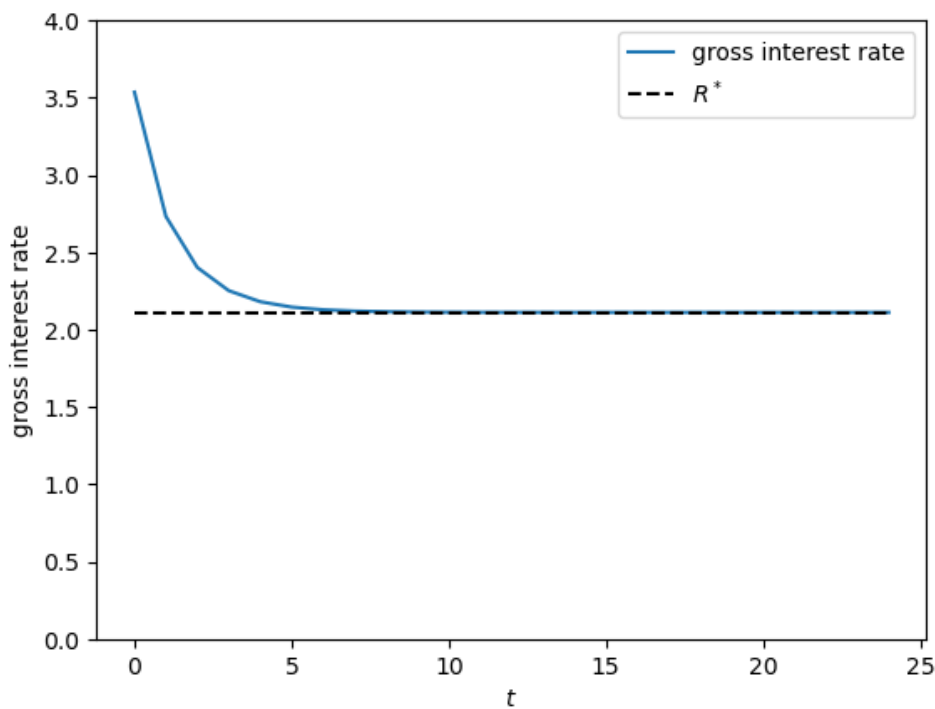

The interest rate reflects the marginal product of capital, which is high when capital stock is low.

26.7 CRRA preferences

Previously, in our examples, we looked at the case of log utility.

Log utility is a rather special case of CRRA utility with $\gamma \to 1$.

In this section, we are going to assume that $u(c) = \frac{c^{1-\gamma}-1}{1-\gamma}$, where $\gamma > 0, \gamma \neq 1$.

This function is called the CRRA utility function.

In other respects, the model is the same.

Below we define the utility function in Python and construct a `namedtuple` to store the parameters.

```python
def crra(c, γ):
    return c**(1 - γ) / (1 - γ)


Model = namedtuple('Model', ['α',          # Cobb-Douglas parameter
                             'β',          # discount factor
                             'γ'])         # parameter in CRRA utility

                   )

def create_olg_model(α=0.4, β=0.9, γ=0.5):
    return Model(α=α, β=β, γ=γ)
```

Let's also redefine the capital demand function to work with this `namedtuple`.

```python
def capital_demand(R, model):
    return (α/R)**(1/(1-model.α))
```

26.7.1 Supply

For households, the Euler equation becomes

$$(w_t - s_t)^{-\gamma} = \beta R_{t+1}^{1-\gamma}(s_t)^{-\gamma} \tag{26.19}$$

Solving for savings, we have

$$s_t = s(w_t, R_{t+1}) = w_t \left[1 + \beta^{-1/\gamma} R_{t+1}^{(\gamma-1)/\gamma}\right]^{-1} \tag{26.20}$$

Notice how, unlike the log case, savings now depends on the interest rate.

```python
def savings_crra(w, R, model):
    α, β, γ = model
    return w / (1 + β**(-1/γ) * R**((γ-1)/γ))
```

```
model = create_olg_model()
w = 2.0

fig, ax = plt.subplots()

ax.plot(R_vals, capital_demand(R_vals, model),
        label="aggregate demand")
ax.plot(R_vals, savings_crra(w, R_vals, model),
        label="aggregate supply")

ax.set_xlabel("$R_{t+1}$")
ax.set_ylabel("$k_{t+1}$")
ax.legend()
plt.show()
```
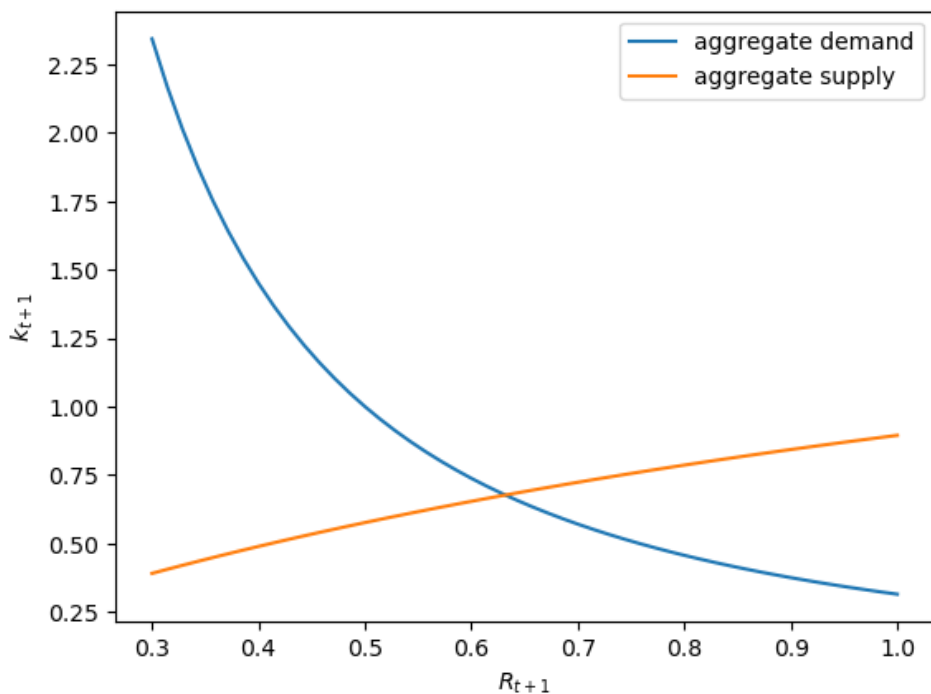

26.7.2 Equilibrium

Equating aggregate demand for capital (see *(26.4.5)*) with our new aggregate supply function yields equilibrium capital.

Thus, we set

$$w_t \left[1 + \beta^{-1/\gamma} R_{t+1}^{(\gamma-1)/\gamma}\right]^{-1} = \left(\frac{R_{t+1}}{\alpha}\right)^{1/(\alpha-1)} \tag{26.21}$$

This expression is quite complex and we cannot solve for R_{t+1} analytically.

Combining *(26.4.4)* and *(26.7.3)* yields

$$k_{t+1} = \left[1 + \beta^{-1/\gamma}(\alpha k_{t+1}^{\alpha-1})^{(\gamma-1)/\gamma}\right]^{-1}(1-\alpha)(k_t)^\alpha \tag{26.22}$$

Again, with this equation and k_t as given, we cannot solve for k_{t+1} by pencil and paper.

In the exercise below, you will be asked to solve these equations numerically.

26.8 Exercises

Exercise 26.8.1

Solve for the dynamics of equilibrium capital stock in the CRRA case numerically using *(26.7.4)*.

Visualize the dynamics using a 45-degree diagram.

Solution to Exercise 26.8.1

To solve for k_{t+1} given k_t we use Newton's method.

Let

$$f(k_{t+1}, k_t) = k_{t+1}\left[1 + \beta^{-1/\gamma}\left(\alpha k_{t+1}^{\alpha-1}\right)^{(\gamma-1)/\gamma}\right] - (1-\alpha)k_t^\alpha = 0 \tag{26.23}$$

If k_t is given then f is a function of unknown k_{t+1}.

Then we can use `scipy.optimize.newton` to solve $f(k_{t+1}, k_t) = 0$ for k_{t+1}.

First let's define f.

```
def f(k_prime, k, model):
    α, β, γ = model.α, model.β, model.γ
    z = (1 - α) * k**α
    a = α**(1-1/γ)
    b = k_prime**((α * γ - α + 1) / γ)
    p = k_prime + k_prime * β**(-1/γ) * a * b
    return p - z
```

Now let's define a function that finds the value of k_{t+1}.

```
def k_update(k, model):
    return optimize.newton(lambda k_prime: f(k_prime, k, model), 0.1)
```

Finally, here is the 45-degree diagram.

```
kmin, kmax = 0, 0.5
n = 1000
k_grid = np.linspace(kmin, kmax, n)
k_grid_next = np.empty_like(k_grid)


for i in range(n):
    k_grid_next[i] = k_update(k_grid[i], model)


fig, ax = plt.subplots(figsize=(6, 6))


ymin, ymax = np.min(k_grid_next), np.max(k_grid_next)

ax.plot(k_grid, k_grid_next,  lw=2, alpha=0.6, label='$g$')
ax.plot(k_grid, k_grid, 'k-', lw=1, alpha=0.7, label='$45^{\circ}$')


ax.legend(loc='upper left', frameon=False, fontsize=12)
ax.set_xlabel('$k_t$', fontsize=12)
ax.set_ylabel('$k_{t+1}$', fontsize=12)

plt.show()
```
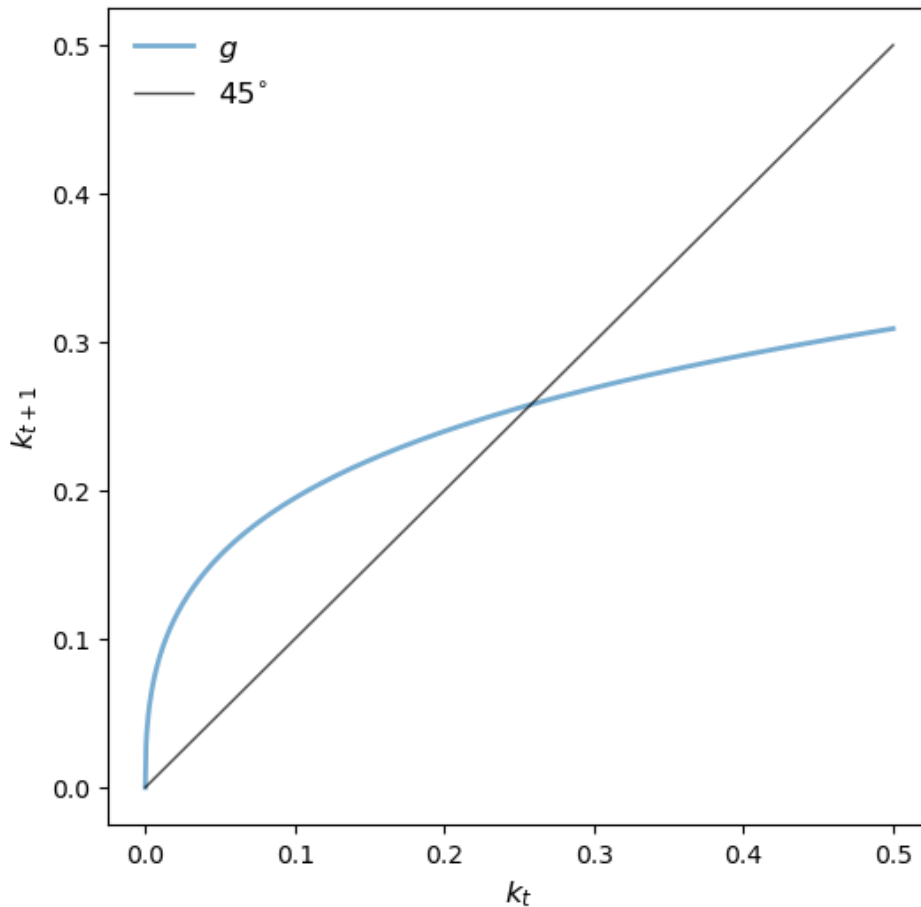
Exercise 26.8.2

The 45-degree diagram from the last exercise shows that there is a unique positive steady state.

The positive steady state can be obtained by setting $k_{t+1} = k_t = k^*$ in *(26.7.4)*, which yields

$$k^* = \frac{(1-\alpha)(k^*)^\alpha}{1 + \beta^{-1/\gamma}(\alpha(k^*)^{\alpha-1})^{(\gamma-1)/\gamma}}$$

Unlike the log preference case, the CRRA utility steady state k^* cannot be obtained analytically.

Instead, we solve for k^* using Newton's method.

Solution to Exercise 26.8.2

We introduce a function h such that positive steady state is the root of h.

$$h(k^*) = k^* \left[1 + \beta^{-1/\gamma}(\alpha(k^*)^{\alpha-1})^{(\gamma-1)/\gamma}\right] - (1-\alpha)(k^*)^\alpha \tag{26.24}$$

Here it is in Python

```
def h(k_star, model):
    α, β, γ = model.α, model.β, model.γ
    z = (1 - α) * k_star**α
    R1 = α ** (1-1/γ)
    R2 = k_star**((α * γ - α + 1) / γ)
    p = k_star + k_star * β**(-1/γ) * R1 * R2
    return p - z
```

Let's apply Newton's method to find the root:

```
k_star = optimize.newton(h, 0.2, args=(model,))
print(f"k_star = {k_star}")
```

```
k_star = 0.25788950250843484
```

Exercise 26.8.3

Generate three time paths for capital, from three distinct initial conditions, under the parameterization listed above.

Use initial conditions for k_0 of $0.001, 1.2, 2.6$ and time series length 10.

Solution to Exercise 26.8.3

Let's define the constants and three distinct intital conditions

```
ts_length = 10
k0 = np.array([0.001, 1.2, 2.6])
```

```
def simulate_ts(model, k0_values, ts_length):

    fig, ax = plt.subplots()

    ts = np.zeros(ts_length)

    # simulate and plot time series
    for k_init in k0_values:
        ts[0] = k_init
        for t in range(1, ts_length):
            ts[t] = k_update(ts[t-1], model)
```

```
        ax.plot(np.arange(ts_length), ts, '-o', ms=4, alpha=0.6,
                label=r'$k_0=%g$' %k_init)
    ax.plot(np.arange(ts_length), np.full(ts_length, k_star),
            alpha=0.6, color='red', label=r'$k^*$')
    ax.legend(fontsize=10)

    ax.set_xlabel(r'$t$', fontsize=14)
    ax.set_ylabel(r'$k_t$', fontsize=14)

    plt.show()
```

```
simulate_ts(model, k0, ts_length)
```

TWENTYSEVEN
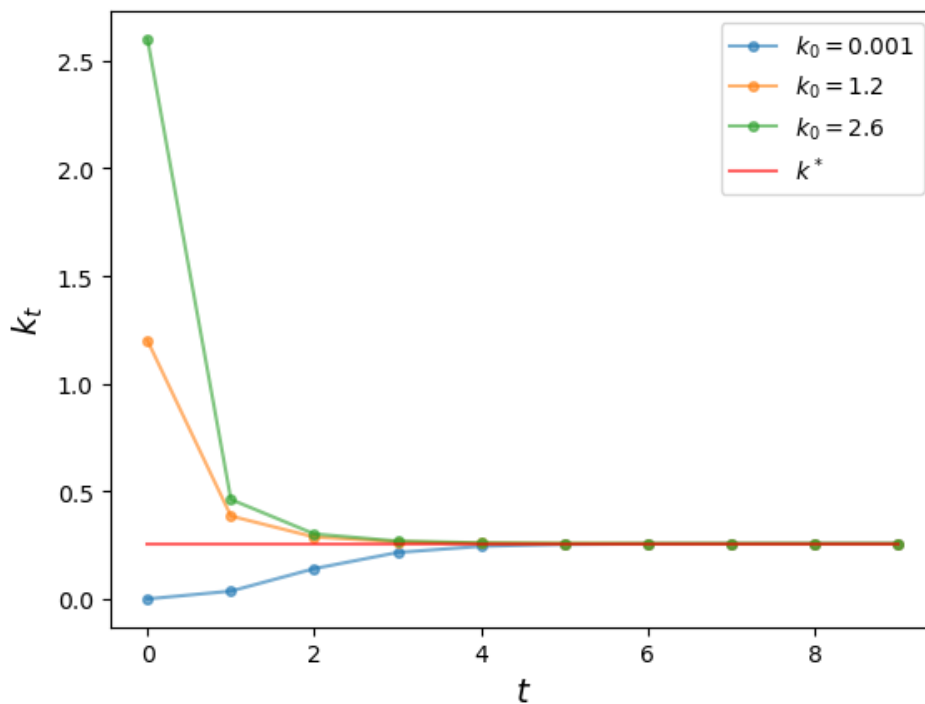
COMMODITY PRICES

27.1 Outline

For more than half of all countries around the globe, commodities account for the majority of total exports.

Examples of commodities include copper, diamonds, iron ore, lithium, cotton and coffee beans.

In this lecture we give an introduction to the theory of commodity prices.

The lecture is quite advanced relative to other lectures in this series.

We need to compute an equilibrium, and that equilibrium is described by a price function.

We will solve an equation where the price function is the unknown.

This is harder than solving an equation for an unknown number, or vector.

The lecture will discuss one way to solve a functional equation (an equation where the unknown object is a function).

For this lecture we need the `yfinance` library.

```
!pip install yfinance
```

We will use the following imports

```python
import numpy as np
import yfinance as yf
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
from scipy.optimize import brentq
from scipy.stats import beta
```

27.2 Data

The figure below shows the price of cotton in USD since the start of 2016.
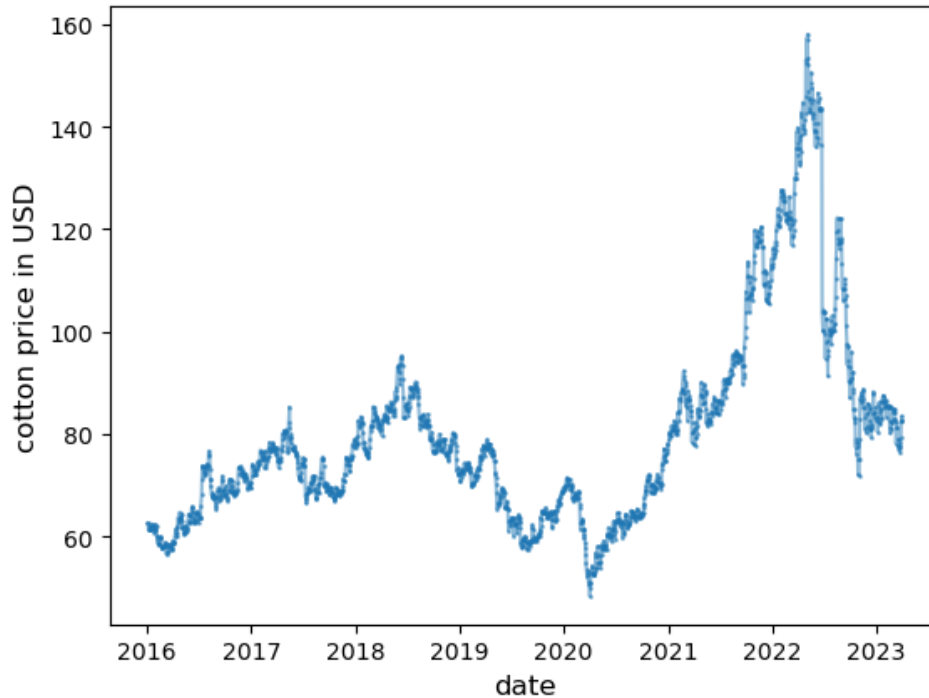
The figure shows surprisingly large movements in the price of cotton.

What causes these movements?

In general, prices depend on the choices and actions of

1. suppliers,

2. consumers, and

3. speculators.

Our focus will be on the interaction between these parties.

We will connect them together in a dynamic model of supply and demand, called the *competitive storage model*.

This model was developed by [Samuelson, 1971], [Wright and Williams, 1982], [Scheinkman and Schechtman, 1983], [Deaton and Laroque, 1992], [Deaton and Laroque, 1996], and [Chambers and Bailey, 1996].

27.3 The competitive storage model

In the competitive storage model, commodities are assets that

1. can be traded by speculators and

2. have intrinsic value to consumers.

Total demand is the sum of consumer demand and demand by speculators.

Supply is exogenous, depending on "harvests".

Note: These days, goods such as basic computer chips and integrated circuits are often treated as commodities in financial markets, being highly standardized, and, for these kinds of commodities, the word "harvest"is not appropriate.

Nonetheless, we maintain it for simplicity.

The equilibrium price is determined competitively.

It is a function of the current state (which determines current harvests and predicts future harvests).

27.4 The model

Consider a market for a single commodity, whose price is given at t by p_t.

The harvest of the commodity at time t is Z_t.

We assume that the sequence $\{Z_t\}_{t \geq 1}$ is IID with common density function ϕ, where ϕ is nonnegative.

Speculators can store the commodity between periods, with I_t units purchased in the current period yielding αI_t units in the next.

Here the parameter $\alpha \in (0, 1)$ is a depreciation rate for the commodity.

For simplicity, the risk free interest rate is taken to be zero, so expected profit on purchasing I_t units is

$$\mathbb{E}_t \, p_{t+1} \cdot \alpha I_t - p_t I_t = (\alpha \mathbb{E}_t \, p_{t+1} - p_t) I_t$$

Here $\mathbb{E}_t \, p_{t+1}$ is the expectation of p_{t+1} taken at time t.

27.5 Equilibrium

In this section we define the equilibrium and discuss how to compute it.

27.5.1 Equilibrium conditions

Speculators are assumed to be risk neutral, which means that they buy the commodity whenever expected profits are positive.

As a consequence, if expected profits are positive, then the market is not in equilibrium.

Hence, to be in equilibrium, prices must satisfy the "no-arbitrage" condition

$$\alpha \mathbb{E}_t \, p_{t+1} - p_t \leq 0 \tag{27.1}$$

This means that if the expected price is lower than the current price, there is no room for arbitrage.

Profit maximization gives the additional condition

$$\alpha \mathbb{E}_t \, p_{t+1} - p_t < 0 \text{ implies } I_t = 0 \tag{27.2}$$

We also require that the market clears, with supply equaling demand in each period.

We assume that consumers generate demand quantity $D(p)$ corresponding to price p.

Let $P := D^{-1}$ be the inverse demand function.

Regarding quantities,

- supply is the sum of carryover by speculators and the current harvest, and

- demand is the sum of purchases by consumers and purchases by speculators.

Mathematically,

- supply is given by $X_t = \alpha I_{t-1} + Z_t$, which takes values in $S := \mathbb{R}_+$, while

- demand $= D(p_t) + I_t$

Thus, the market equilibrium condition is

$$\alpha I_{t-1} + Z_t = D(p_t) + I_t \tag{27.3}$$

The initial condition $X_0 \in S$ is treated as given.

27.5.2 An equilibrium function

How can we find an equilibrium?

Our path of attack will be to seek a system of prices that depend only on the current state.

(Our solution method involves using an ansatz, which is an educated guess —in this case for the price function.)

In other words, we take a function p on S and set $p_t = p(X_t)$ for every t.

Prices and quantities then follow

$$p_t = p(X_t), \quad I_t = X_t - D(p_t), \quad X_{t+1} = \alpha I_t + Z_{t+1} \tag{27.4}$$

We choose p so that these prices and quantities satisfy the equilibrium conditions above.

More precisely, we seek a p such that *(27.5.1)* and *(27.5.2)* hold for the corresponding system *(27.5.4)*.

$$p^*(x) = \max \left\{ \alpha \int_0^\infty p^*(\alpha I(x) + z)\phi(z)dz, P(x) \right\} \qquad (x \in S) \tag{27.5}$$

where

$$I(x) := x - D(p^*(x)) \qquad (x \in S) \tag{27.6}$$

It turns out that such a p^* will suffice, in the sense that *(27.5.1)* and *(27.5.2)* hold for the corresponding system *(27.5.4)*.

To see this, observe first that

$$\mathbb{E}_t \, p_{t+1} = \mathbb{E}_t \, p^*(X_{t+1}) = \mathbb{E}_t \, p^*(\alpha I(X_t) + Z_{t+1}) = \int_0^\infty p^*(\alpha I(X_t) + z)\phi(z)dz$$

Thus *(27.5.1)* requires that

$$\alpha \int_0^\infty p^*(\alpha I(X_t) + z)\phi(z)dz \le p^*(X_t)$$

This inequality is immediate from *(27.5.5)*.

Second, regarding *(27.5.2)*, suppose that

$$\alpha \int_0^\infty p^*(\alpha I(X_t) + z)\phi(z)dz < p^*(X_t)$$

Then by *(27.5.5)* we have $p^*(X_t) = P(X_t)$

But then $D(p^*(X_t)) = X_t$ and $I_t = I(X_t) = 0$.

As a consequence, both *(27.5.1)* and *(27.5.2)* hold.

We have found an equilibrium, which verifies the ansatz.

27.5.3 Computing the equilibrium

We now know that an equilibrium can be obtained by finding a function p^* that satisfies *(27.5.5)*.

It can be shown that, under mild conditions there is exactly one function on S satisfying *(27.5.5)*.

Moreover, we can compute this function using successive approximation.

This means that we start with a guess of the function and then update it using *(27.5.5)*.

This generates a sequence of functions $p_1, p_2, \ldots$

We continue until this process converges, in the sense that p_k and p_{k+1} are very close together.

Then we take the final p_k that we computed as our approximation of p^*.

To implement our update step, it is helpful if we put *(27.5.5)* and *(27.5.6)* together.

This leads us to the update rule

$$p_{k+1}(x) = \max\left\{ \alpha \int_0^\infty p_k(\alpha(x - D(p_{k+1}(x))) + z)\phi(z)dz, P(x) \right\} \tag{27.7}$$

In other words, we take p_k as given and, at each x, solve for q in

$$q = \max\left\{ \alpha \int_0^\infty p_k(\alpha(x - D(q)) + z)\phi(z)dz, P(x) \right\} \tag{27.8}$$

Actually we can't do this at every x, so instead we do it on a grid of points $x_1, \ldots, x_n$.

Then we get the corresponding values $q_1, \ldots, q_n$.

Then we compute p_{k+1} as the linear interpolation of the values $q_1, \ldots, q_n$ over the grid $x_1, \ldots, x_n$.

Then we repeat, seeking convergence.

27.6 Code

The code below implements this iterative process, starting from $p_0 = P$.

The distribution ϕ is set to a shifted Beta distribution (although many other choices are possible).

The integral in *(27.5.8)* is computed via *Monte Carlo*.

```
α, a, c = 0.8, 1.0, 2.0
beta_a, beta_b = 5, 5
mc_draw_size = 250
gridsize = 150
grid_max = 35
grid = np.linspace(a, grid_max, gridsize)
```

```python
beta_dist = beta(5, 5)
Z = a + beta_dist.rvs(mc_draw_size) * c     # Shock observations
D = P = lambda x: 1.0 / x
tol = 1e-4


def T(p_array):

    new_p = np.empty_like(p_array)

    # Interpolate to obtain p as a function.
    p = interp1d(grid,
                 p_array,
                 fill_value=(p_array[0], p_array[-1]),
                 bounds_error=False)

    # Update
    for i, x in enumerate(grid):

        h = lambda q: q - max(α * np.mean(p(α * (x - D(q)) + Z)), P(x))
        new_p[i] = brentq(h, 1e-8, 100)

    return new_p


fig, ax = plt.subplots()

price = P(grid)
ax.plot(grid, price, alpha=0.5, lw=1, label="inverse demand curve")
error = tol + 1
while error > tol:
    new_price = T(price)
    error = max(np.abs(new_price - price))
    price = new_price

ax.plot(grid, price, 'k-', alpha=0.5, lw=2, label=r'$p^*$')
ax.legend()
ax.set_xlabel('$x$')
ax.set_ylabel("prices")

plt.show()
```
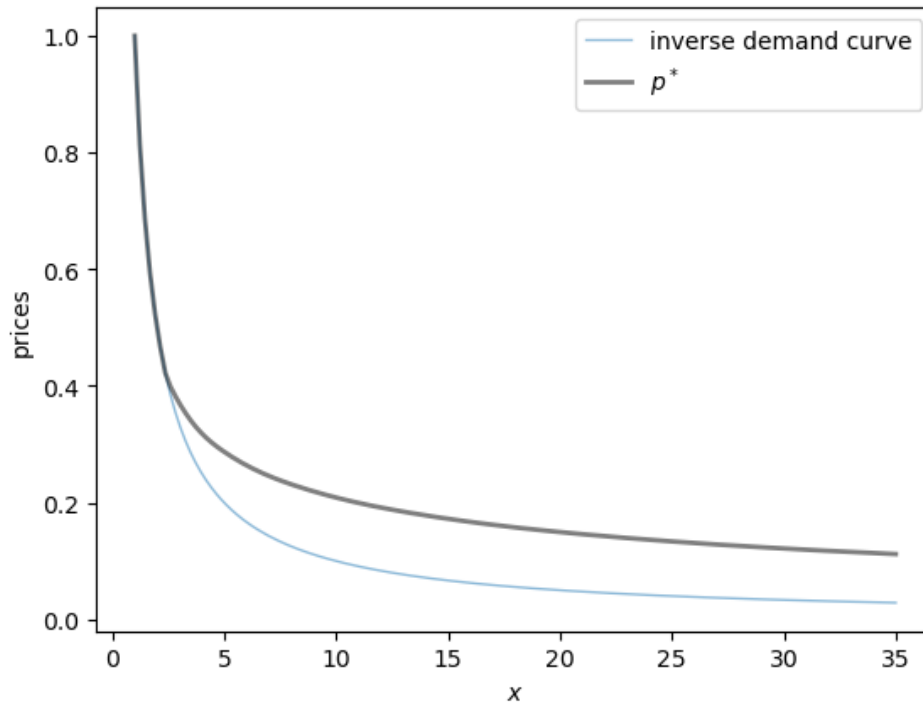
The figure above shows the inverse demand curve P, which is also p_0, as well as our approximation of p^*.

Once we have an approximation of p^*, we can simulate a time series of prices.

```python
# Turn the price array into a price function
p_star = interp1d(grid,
                  price,
                  fill_value=(price[0], price[-1]),
                  bounds_error=False)


def carry_over(x):
    return α * (x - D(p_star(x)))


def generate_cp_ts(init=1, n=50):
    X = np.empty(n)
    X[0] = init
    for t in range(n-1):
            Z = a + c * beta_dist.rvs()
            X[t+1] = carry_over(X[t]) + Z
    return p_star(X)

fig, ax = plt.subplots()
ax.plot(generate_cp_ts(), label="price")
ax.set_xlabel("time")
ax.legend()
```

```
plt.show()
```
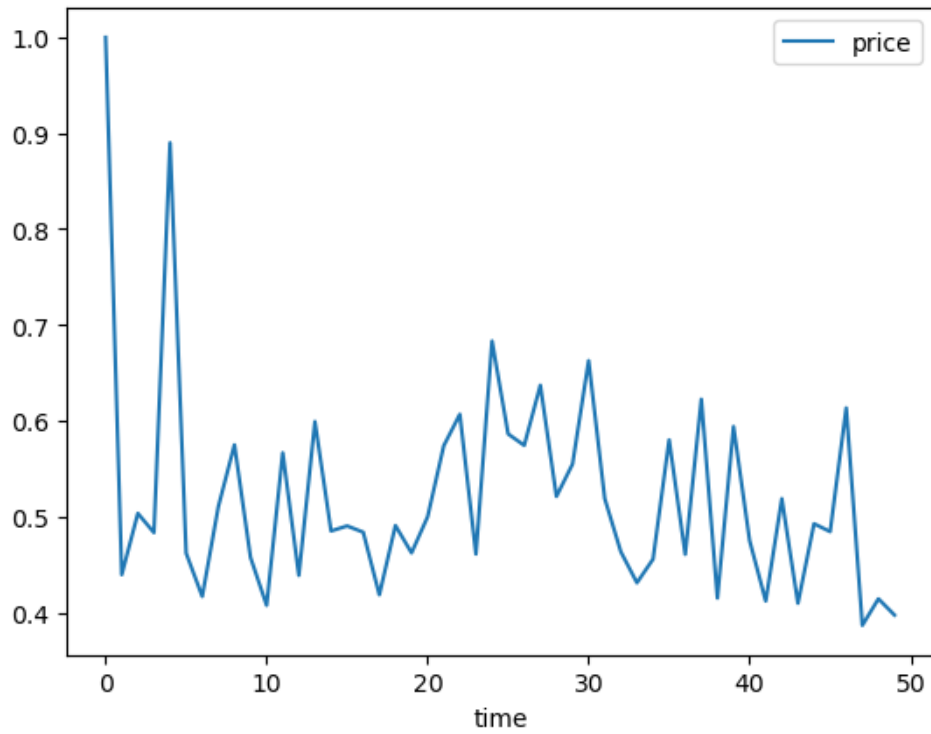
Part VIII

Monetary-Fiscal Policy Interactions

MONEY FINANCED GOVERNMENT DEFICITS AND PRICE LEVELS

28.1 Overview

This lecture extends and modifies the model in this lecture *A Monetarist Theory of Price Levels* by modifying the law of motion that governed the supply of money.

The model in this lecture consists of two components

- a demand function for money

- a law of motion for the supply of money

The demand function describes the public's demand for "real balances", defined as the ratio of nominal money balances to the price level

- it assumes that the demand for real balance today varies inversely with the rate of inflation that the public forecasts to prevail between today and tomorrow

- it assumes that the public's forecast of that rate of inflation is perfect

The law of motion for the supply of money assumes that the government prints money to finance government expenditures

Our model equates the demand for money to the supply at each time $t \geq 0$.

Equality between those demands and supply gives a *dynamic* model in which money supply and price level *sequences* are simultaneously determined by a set of simultaneous linear equations.

These equations take the form of what is often called vector linear **difference equations**.

In this lecture, we'll roll up our sleeves and solve those equations in two different ways.

(One of the methods for solving vector linear difference equations will take advantage of a decomposition of a matrix that is studied in this lecture *Eigenvalues and Eigenvectors*.)

In this lecture we will encounter these concepts from macroeconomics:

- an **inflation tax** that a government gathers by printing paper or electronic money

- a dynamic **Laffer curve** in the inflation tax rate that has two stationary equilibria

- perverse dynamics under rational expectations in which the system converges to the higher stationary inflation tax rate

- a peculiar comparative stationary-state outcome connected with that stationary inflation rate: it asserts that inflation can be *reduced* by running *higher* government deficits, i.e., by raising more resources by printing money.

The same qualitative outcomes prevail in this lecture *Inflation Rate Laffer Curves* that studies a nonlinear version of the model in this lecture.

These outcomes set the stage for the analysis to be presented in this lecture *Laffer Curves with Adaptive Expectations* that studies a nonlinear version of the present model; it assumes a version of "adaptive expectations" instead of rational expectations.

That lecture will show that

- replacing rational expectations with adaptive expectations leaves the two stationary inflation rates unchanged, but that …

- it reverses the perverse dynamics by making the *lower* stationary inflation rate the one to which the system typically converges

- a more plausible comparative dynamic outcome emerges in which now inflation can be *reduced* by running *lower* government deficits

This outcome will be used to justify a selection of a stationary inflation rate that underlies the analysis of unpleasant monetarist arithmetic to be studied in this lecture *Some Unpleasant Monetarist Arithmetic*.

We'll use these tools from linear algebra:

- matrix multiplication

- matrix inversion

- eigenvalues and eigenvectors of a matrix

28.2 Demand for and supply of money

We say demand*s* and suppl*ies* (plurals) because there is one of each for each $t \geq 0$.

Let

- m_{t+1} be the supply of currency at the end of time $t \geq 0$

- m_t be the supply of currency brought into time t from time $t-1$

- g be the government deficit that is financed by printing currency at $t \geq 1$

- m_{t+1}^d be the demand at time t for currency to bring into time $t + 1$

- p_t be the price level at time t

- $b_t = \frac{m_{t+1}}{p_t}$ is real balances at the end of time t

- $R_t = \frac{p_t}{p_{t+1}}$ be the gross rate of return on currency held from time t to time $t + 1$

It is often helpful to state units in which quantities are measured:

- m_t and m_t^d are measured in dollars

- g is measured in time t goods

- p_t is measured in dollars per time t goods

- R_t is measured in time $t + 1$ goods per unit of time t goods

- b_t is measured in time t goods

Our job now is to specify demand and supply functions for money.

We assume that the demand for currency satisfies the Cagan-like demand function

$$\frac{m_{t+1}^d}{p_t} = \gamma_1 - \gamma_2 \frac{p_{t+1}}{p_t}, \quad t \geq 0 \tag{28.1}$$

where γ_1, γ_2 are positive parameters.

Now we turn to the supply of money.

We assume that $m_0 > 0$ is an "initial condition" determined outside the model.

We set m_0 at some arbitrary positive value, say \$100.

For $t \geq 1$, we assume that the supply of money is determined by the government's budget constraint

$$m_{t+1} - m_t = p_t g, \quad t \geq 0 \tag{28.2}$$

According to this equation, each period, the government prints money to pay for quantity g of goods.

In an **equilibrium**, the demand for currency equals the supply:

$$m_{t+1}^d = m_{t+1}, \quad t \geq 0 \tag{28.3}$$

Let's take a moment to think about what equation (**??**) tells us.

The demand for money at any time t depends on the price level at time t and the price level at time $t + 1$.

The supply of money at time $t + 1$ depends on the money supply at time t and the price level at time t.

So the infinite sequence of equations (**??**) for $t \geq 0$ imply that the *sequences* $\{p_t\}_{t=0}^\infty$ and $\{m_t\}_{t=0}^\infty$ are tied together and ultimately simultaneously determined.

28.3 Equilibrium price and money supply sequences

The preceding specifications imply that for $t \geq 1$, **real balances** evolve according to

$$\frac{m_{t+1}}{p_t} - \frac{m_t}{p_{t-1}} \frac{p_{t-1}}{p_t} = g$$

or

$$b_t - b_{t-1} R_{t-1} = g \tag{28.4}$$

The demand for real balances is

$$b_t = \gamma_1 - \gamma_2 R_t^{-1}. \tag{28.5}$$

We'll restrict our attention to parameter values and associated gross real rates of return on real balances that assure that the demand for real balances is positive, which according to (**??**) means that

$$b_t = \gamma_1 - \gamma_2 R_t^{-1} > 0$$

which implies that

$$R_t \geq \left(\frac{\gamma_2}{\gamma_1} \right) \equiv \underline{R} \tag{28.6}$$

Gross real rate of return $\underline{R}$ is the smallest rate of return on currency that is consistent with a nonnegative demand for real balances.

We shall describe two distinct but closely related ways of computing a pair $\{p_t, m_t\}_{t=0}^{\infty}$ of sequences for the price level and money supply.

But first it is instructive to describe a special type of equilibrium known as a **steady state**.

In a steady-state equilibrium, a subset of key variables remain constant or **invariant** over time, while remaining variables can be expressed as functions of those constant variables.

Finding such state variables is something of an art.

In many models, a good source of candidates for such invariant variables is a set of *ratios*.

This is true in the present model.

28.3.1 Steady states

In a steady-state equilibrium of the model we are studying,

$$R_t = \bar{R}$$
$$b_t = \bar{b}$$

for $t \geq 0$.

Notice that both $R_t = \frac{p_t}{p_{t+1}}$ and $b_t = \frac{m_{t+1}}{p_t}$ are *ratios*.

To compute a steady state, we seek gross rates of return on currency and real balances $\bar{R}, \bar{b}$ that satisfy steady-state versions of both the government budget constraint and the demand function for real balances:

$$g = \bar{b}(1 - \bar{R})$$
$$\bar{b} = \gamma_1 - \gamma_2 \bar{R}^{-1}$$

Together these equations imply

$$(\gamma_1 + \gamma_2) - \frac{\gamma_2}{\bar{R}} - \gamma_1 \bar{R} = g \tag{28.7}$$

The left side is the steady-state amount of **seigniorage** or government revenues that the government gathers by paying a gross rate of return $\bar{R} \leq 1$ on currency.

The right side is government expenditures.

Define steady-state seigniorage as

$$S(\bar{R}) = (\gamma_1 + \gamma_2) - \frac{\gamma_2}{\bar{R}} - \gamma_1 \bar{R} \tag{28.8}$$

Notice that $S(\bar{R}) \geq 0$ only when $\bar{R} \in [\frac{\gamma_2}{\gamma_1}, 1] \equiv [\underline{R}, \overline{R}]$ and that $S(\bar{R}) = 0$ if $\bar{R} = \underline{R}$ or if $\bar{R} = \overline{R}$.

We shall study equilibrium sequences that satisfy

$$R_t \in [\underline{R}, \overline{R}], \quad t \geq 0.$$

Maximizing steady-state seigniorage (**??**) with respect to $\bar{R}$, we find that the maximizing rate of return on currency is

$$\bar{R}_{\max} = \sqrt{\frac{\gamma_2}{\gamma_1}}$$

and that the associated maximum seigniorage revenue that the government can gather from printing money is

$$(\gamma_1 + \gamma_2) - \frac{\gamma_2}{\bar{R}_{\max}} - \gamma_1 \bar{R}_{\max}$$

It is useful to rewrite equation (**??**) as

$$-\gamma_2 + (\gamma_1 + \gamma_2 - g)\bar{R} - \gamma_1 \bar{R}^2 = 0 \tag{28.9}$$

A steady state gross rate of return $\bar{R}$ solves quadratic equation (**??**).

So two steady states typically exist.

28.4 Some code

Let's start with some imports:

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
plt.rcParams['figure.dpi'] = 300
from collections import namedtuple
```

Let's set some parameter values and compute possible steady-state rates of return on currency $\bar{R}$, the seigniorage maximizing rate of return on currency, and an object that we'll discuss later, namely, an initial price level p_0 associated with the maximum steady-state rate of return on currency.

First, we create a `namedtuple` to store parameters so that we can reuse this `namedtuple` in our functions throughout this lecture

```python
# Create a namedtuple that contains parameters
MoneySupplyModel = namedtuple("MoneySupplyModel",
                              ["γ1", "γ2", "g",
                               "M0", "R_u", "R_l"])


def create_model(γ1=100, γ2=50, g=3.0, M0=100):

    # Calculate the steady states for R
    R_steady = np.roots((-γ1, γ1 + γ2 - g, -γ2))
    R_u, R_l = R_steady
    print("[R_u, R_l] =", R_steady)

    return MoneySupplyModel(γ1=γ1, γ2=γ2, g=g, M0=M0, R_u=R_u, R_l=R_l)
```

Now we compute the $\bar{R}_{\max}$ and corresponding revenue

```python
def seign(R, model):
    γ1, γ2, g = model.γ1, model.γ2, model.g
    return -γ2/R + (γ1 + γ2)  - γ1 * R

msm = create_model()

# Calculate initial guess for p0
p0_guess = msm.M0 / (msm.γ1 - msm.g - msm.γ2 / msm.R_u)
print(f'p0 guess = {p0_guess:.4f}')

# Calculate seigniorage maximizing rate of return
R_max = np.sqrt(msm.γ2/msm.γ1)
g_max = seign(R_max, msm)
print(f'R_max, g_max = {R_max:.4f}, {g_max:.4f}')
```

```
[R_u, R_l] = [0.93556171 0.53443829]
p0 guess = 2.2959
R_max, g_max = 0.7071, 8.5786
```

Now let's plot seigniorage as a function of alternative potential steady-state values of R.

We'll see that there are two steady-state values of R that attain seigniorage levels equal to g, one that we'll denote R_ℓ, another that we'll denote R_u.

They satisfy $R_\ell < R_u$ and are affiliated with a higher inflation tax rate $(1 - R_\ell)$ and a lower inflation tax rate $1 - R_u$.

```
# Generate values for R
R_values = np.linspace(msm.γ2/msm.γ1, 1, 250)

# Calculate the function values
seign_values = seign(R_values, msm)

# Visualize seign_values against R values
fig, ax = plt.subplots(figsize=(11, 5))
plt.plot(R_values, seign_values, label='inflation tax revenue')
plt.axhline(y=msm.g, color='red', linestyle='--', label='government deficit')
plt.xlabel('$R$')
plt.ylabel('seigniorage')

plt.legend()
plt.show()
```
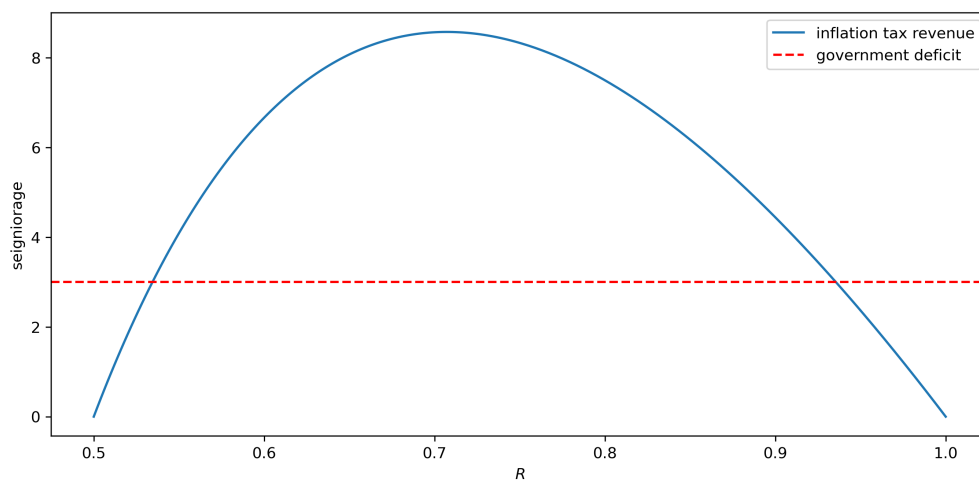

Fig. 28.1: Steady state revenue from inflation tax as function of steady state gross return on currency (solid blue curve) and real government expenditures (dotted red line) plotted against steady-state rate of return currency

Let's print the two steady-state rates of return $\bar{R}$ and the associated seigniorage revenues that the government collects.

(By construction, both steady-state rates of return should raise the same amounts real revenue.)

We hope that the following code will confirm this.

```
g1 = seign(msm.R_u, msm)
print(f'R_u, g_u = {msm.R_u:.4f}, {g1:.4f}')


g2 = seign(msm.R_l, msm)
print(f'R_l, g_l = {msm.R_l:.4f}, {g2:.4f}')
```

```
R_u, g_u = 0.9356, 3.0000
R_l, g_l = 0.5344, 3.0000
```

Now let's compute the maximum steady-state amount of seigniorage that could be gathered by printing money and the state-state rate of return on money that attains it.

28.5 Two computation strategies

We now proceed to compute equilibria, not necessarily steady states.

We shall deploy two distinct computation strategies.

28.5.1 Method 1

- set $R_0 \in [\frac{\gamma_2}{\gamma_1}, R_u]$ and compute $b_0 = \gamma_1 - \gamma_2/R_0$.

- compute sequences $\{R_t, b_t\}_{t=1}^{\infty}$ of rates of return and real balances that are associated with an equilibrium by solving equation (**??**) and (**??**) sequentially for $t \geq 1$:

$$
\begin{aligned}
b_t &= b_{t-1}R_{t-1} + g \\
R_t^{-1} &= \frac{\gamma_1}{\gamma_2} - \gamma_2^{-1}b_t
\end{aligned}
\tag{28.10}
$$

- Construct the associated equilibrium p_0 from

$$
p_0 = \frac{m_0}{\gamma_1 - g - \gamma_2/R_0}
\tag{28.11}
$$

- compute $\{p_t, m_t\}_{t=1}^{\infty}$ by solving the following equations sequentially

$$
\begin{aligned}
p_t &= R_t p_{t-1} \\
m_t &= b_{t-1}p_t
\end{aligned}
\tag{28.12}
$$

Remark 1: method 1 uses an indirect approach to computing an equilibrium by first computing an equilibrium $\{R_t, b_t\}_{t=0}^{\infty}$ sequence and then using it to back out an equilibrium $\{p_t, m_t\}_{t=0}^{\infty}$ sequence.

Remark 2: notice that method 1 starts by picking an **initial condition** R_0 from a set $[\frac{\gamma_2}{\gamma_1}, R_u]$. Equilibrium $\{p_t, m_t\}_{t=0}^\infty$ sequences are not unique. There is actually a continuum of equilibria indexed by a choice of R_0 from the set $[\frac{\gamma_2}{\gamma_1}, R_u]$.

Remark 3: associated with each selection of R_0 there is a unique p_0 described by equation (**??**).

28.5.2 Method 2

This method deploys a direct approach. It defines a "state vector" $y_t = \begin{bmatrix} m_t \\ p_t \end{bmatrix}$ and formulates equilibrium conditions (**??**), (**??**), and (**??**) in terms of a first-order vector difference equation

$$y_{t+1} = M y_t, \quad t \geq 0,$$

where we temporarily take $y_0 = \begin{bmatrix} m_0 \\ p_0 \end{bmatrix}$ as an **initial condition**.

The solution is

$$y_t = M^t y_0.$$

Now let's think about the initial condition y_0.

It is natural to take the initial stock of money $m_0 > 0$ as an initial condition.

But what about p_0?

Isn't it something that we want to be *determined* by our model?

Yes, but sometimes we want too much, because there is actually a continuum of initial p_0 levels that are compatible with the existence of an equilibrium.

As we shall see soon, selecting an initial p_0 in method 2 is intimately tied to selecting an initial rate of return on currency R_0 in method 1.

28.6 Computation method 1

Remember that there exist two steady-state equilibrium values $R_\ell < R_u$ of the rate of return on currency R_t.

We proceed as follows.

Start at $t = 0$

- select a $R_0 \in [\frac{\gamma_2}{\gamma_1}, R_u]$

- compute $b_0 = \gamma_1 - \gamma_0 R_0^{-1}$

Then for $t \geq 1$ construct b_t, R_t by iterating on equation (**??**).

When we implement this part of method 1, we shall discover the following striking outcome:

- starting from an R_0 in $[\frac{\gamma_2}{\gamma_1}, R_u]$, we shall find that $\{R_t\}$ always converges to a limiting "steady state" value $\bar{R}$ that depends on the initial condition R_0.

- there are only two possible limit points $\{R_\ell, R_u\}$.

- for almost every initial condition R_0, $\lim_{t \to +\infty} R_t = R_\ell$.

- if and only if $R_0 = R_u$, $\lim_{t \to +\infty} R_t = R_u$.

The quantity $1 - R_t$ can be interpreted as an **inflation tax rate** that the government imposes on holders of its currency.

We shall soon see that the existence of two steady-state rates of return on currency that serve to finance the government deficit of g indicates the presence of a **Laffer curve** in the inflation tax rate.

Note: Arthur Laffer's curve plots a hump shaped curve of revenue raised from a tax against the tax rate.
Its hump shape indicates that there are typically two tax rates that yield the same amount of revenue. This is due to two countervailing courses, one being that raising a tax rate typically decreases the **base** of the tax as people take decisions to reduce their exposure to the tax.

```python
def simulate_system(R0, model, num_steps):
    γ1, γ2, g = model.γ1, model.γ2, model.g

    # Initialize arrays to store results
    b_values = np.empty(num_steps)
    R_values = np.empty(num_steps)

    # Initial values
    b_values[0] = γ1 - γ2/R0
    R_values[0] = 1 / (γ1/γ2 - (1 / γ2) * b_values[0])

    # Iterate over time steps
    for t in range(1, num_steps):
        b_t = b_values[t - 1] * R_values[t - 1] + g
        R_values[t] = 1 / (γ1/γ2 - (1/γ2) * b_t)
        b_values[t] = b_t

    return b_values, R_values
```

Let's write some code to plot outcomes for several possible initial values R_0.

Let's plot distinct outcomes associated with several $R_0 \in [\frac{\gamma_2}{\gamma_1}, R_u]$.

Each line below shows a path associated with a different R_0.

```
# Create a grid of R_0s
R0s = np.linspace(msm.γ2/msm.γ1, msm.R_u, 9)
R0s = np.append(msm.R_l, R0s)
draw_paths(R0s, msm, line_params, num_steps=20)
```
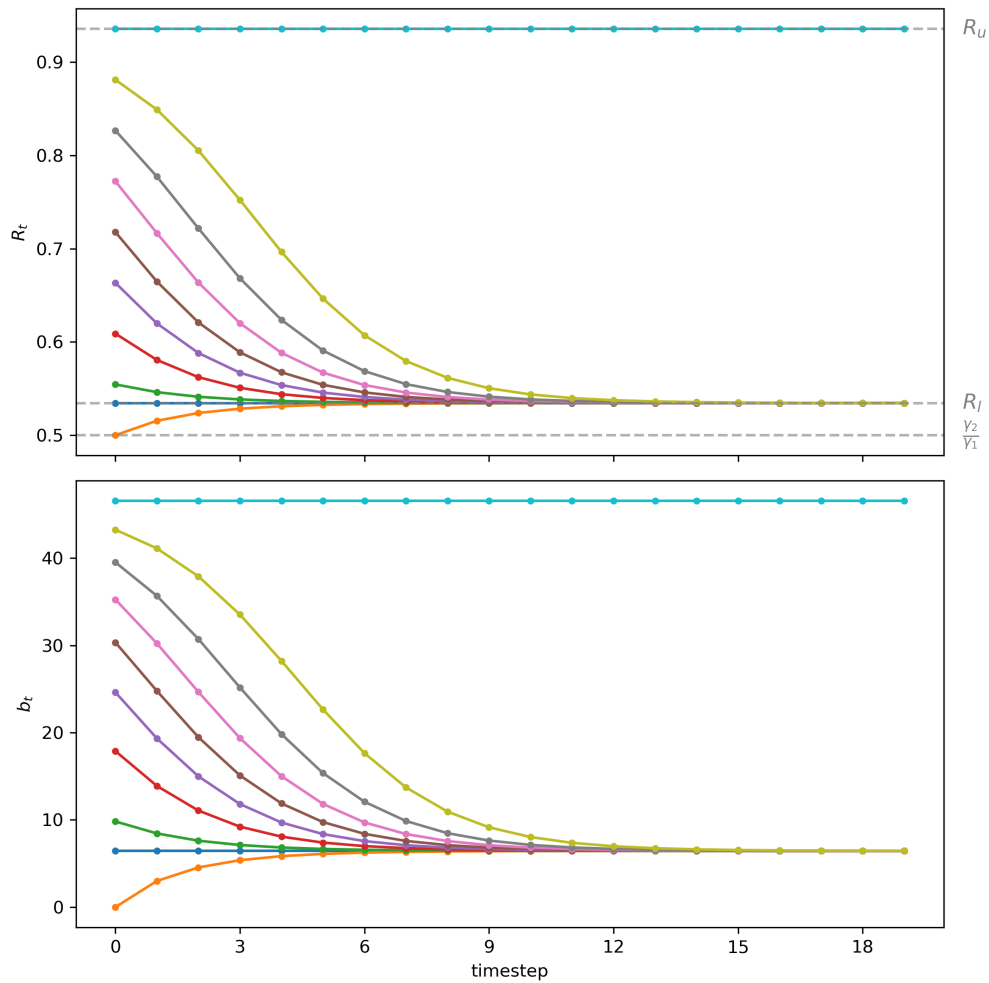

Fig. 28.2: Paths of R_t (top panel) and b_t (bottom panel) starting from different initial condition R_0

Notice how sequences that start from R_0 in the half-open interval $[R_\ell, R_u)$ converge to the steady state associated with to R_ℓ.

28.7 Computation method 2

Set $m_t = m_t^d$ for all $t \geq -1$.

Let

$$y_t = \begin{bmatrix} m_t \\ p_t \end{bmatrix}.$$

Represent equilibrium conditions (**??**), (**??**), and (**??**) as

$$\begin{bmatrix} 1 & \gamma_2 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} m_{t+1} \\ p_{t+1} \end{bmatrix} = \begin{bmatrix} 0 & \gamma_1 \\ 1 & g \end{bmatrix} \begin{bmatrix} m_t \\ p_t \end{bmatrix} \tag{28.13}$$

or

$$H_1 y_t = H_2 y_{t-1}$$

where

$$H_1 = \begin{bmatrix} 1 & \gamma_2 \\ 1 & 0 \end{bmatrix}$$

$$H_2 = \begin{bmatrix} 0 & \gamma_1 \\ 1 & g \end{bmatrix}$$

```
H1 = np.array([[1, msm.γ2],
               [1, 0]])
H2 = np.array([[0, msm.γ1],
               [1, msm.g]])
```

Define

$$H = H_1^{-1} H_2$$

```
H = np.linalg.solve(H1, H2)
print('H = \n', H)
```

```
H =
 [[ 1.    3.  ]
 [-0.02  1.94]]
```

and write the system (**??**) as

$$y_{t+1} = H y_t, \quad t \geq 0 \tag{28.14}$$

so that $\{y_t\}_{t=0}^{\infty}$ can be computed from

$$y_t = H^t y_0, t \geq 0 \tag{28.15}$$

where

$$y_0 = \begin{bmatrix} m_0 \\ p_0 \end{bmatrix}.$$

It is natural to take m_0 as an initial condition determined outside the model.

The mathematics seems to tell us that p_0 must also be determined outside the model, even though it is something that we actually wanted to be determined by the model.

(As usual, we should listen when mathematics talks to us.)

For now, let's just proceed mechanically on faith.

Compute the eigenvector decomposition

$$H = Q\Lambda Q^{-1}$$

where Λ is a diagonal matrix of eigenvalues and the columns of Q are eigenvectors corresponding to those eigenvalues.

It turns out that

$$\Lambda = \begin{bmatrix} R_\ell^{-1} & 0 \\ 0 & R_u^{-1} \end{bmatrix}$$

where R_ℓ and R_u are the lower and higher steady-state rates of return on currency that we computed above.

```python
Λ, Q = np.linalg.eig(H)
print('Λ = \n', Λ)
print('Q = \n', Q)
```

```
Λ =
  [1.06887658 1.87112342]
Q =
  [[-0.99973655 -0.96033288]
  [-0.02295281 -0.27885616]]
```

```python
R_l = 1 / Λ[0]
R_u = 1 / Λ[1]

print(f'R_l = {R_l:.4f}')
print(f'R_u = {R_u:.4f}')
```

```
R_l = 0.9356
R_u = 0.5344
```

Partition Q as

$$Q = \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix}$$

Below we shall verify the following claims:

Claims: If we set

$$p_0 = \bar{p}_0 \equiv Q_{21} Q_{11}^{-1} m_0, \tag{28.16}$$

it turns out that

$$\frac{p_{t+1}}{p_t} = R_u^{-1}, \quad t \geq 0$$

However, if we set

$$p_0 > \bar{p}_0$$

then

$$\lim_{t \to +\infty} \frac{p_{t+1}}{p_t} = R_\ell^{-1}.$$

Let's verify these claims step by step.

Note that

$$H^t = Q \Lambda^t Q^{-1}$$

so that

$$y_t = Q \Lambda^t Q^{-1} y_0$$

```python
def iterate_H(y_0, H, num_steps):
    Λ, Q = np.linalg.eig(H)
    Q_inv = np.linalg.inv(Q)
    y = np.stack(
        [Q @ np.diag(Λ**t) @ Q_inv @ y_0 for t in range(num_steps)], 1)

    return y
```

For almost all initial vectors y_0, the gross rate of inflation $\frac{p_{t+1}}{p_t}$ eventually converges to the larger eigenvalue R_ℓ^{-1}.

The only way to avoid this outcome is for p_0 to take the specific value described by (**??**).

To understand this situation, we use the following transformation

$$y_t^* = Q^{-1} y_t.$$

Dynamics of y_t^* are evidently governed by

$$y_{t+1}^* = \Lambda^t y_t^*. \tag{28.17}$$

This equation represents the dynamics of our system in a way that lets us isolate the force that causes gross inflation to converge to the inverse of the lower steady-state rate of inflation R_ℓ that we discovered earlier.

Staring at equation (**??**) indicates that unless

$$y_0^* = \begin{bmatrix} y_{1,0}^* \\ 0 \end{bmatrix} \tag{28.18}$$

the path of y_t^*, and therefore the paths of both m_t and p_t given by $y_t = Qy_t^*$ will eventually grow at gross rates R_ℓ^{-1} as $t \to +\infty$.

Equation (**??**) also leads us to conclude that there is a unique setting for the initial vector y_0 for which both components forever grow at the lower rate R_u^{-1}.

For this to occur, the required setting of y_0 must evidently have the property that

$$Q^{-1} y_0 = y_0^* = \begin{bmatrix} y_{1,0}^* \\ 0 \end{bmatrix}.$$

But note that since $y_0 = \begin{bmatrix} m_0 \\ p_0 \end{bmatrix}$ and m_0 is given to us an initial condition, p_0 has to do all the adjusting to satisfy this equation.

Sometimes this situation is described informally by saying that while m_0 is truly a **state** variable, p_0 is a **jump** variable that must adjust at $t = 0$ in order to satisfy the equation.

Thus, in a nutshell the unique value of the vector y_0 for which the paths of y_t *don't* eventually grow at rate R_ℓ^{-1} requires setting the second component of y_0^* equal to zero.

The component p_0 of the initial vector $y_0 = \begin{bmatrix} m_0 \\ p_0 \end{bmatrix}$ must evidently satisfy

$$Q^{\{2\}} y_0 = 0$$

where $Q^{\{2\}}$ denotes the second row of Q^{-1}, a restriction that is equivalent to

$$Q^{21} m_0 + Q^{22} p_0 = 0 \tag{28.19}$$

where Q^{ij} denotes the (i,j) component of Q^{-1}.

Solving this equation for p_0, we find

$$p_0 = -(Q^{22})^{-1} Q^{21} m_0. \tag{28.20}$$

28.7.1 More convenient formula

We can get the equivalent but perhaps more convenient formula (**??**) for p_0 that is cast in terms of components of Q instead of components of Q^{-1}.

To get this formula, first note that because $(Q^{21} \ Q^{22})$ is the second row of the inverse of Q and because $Q^{-1}Q = I$, it follows that

$$\begin{bmatrix} Q^{21} & Q^{22} \end{bmatrix} \begin{bmatrix} Q_{11} \\ Q_{21} \end{bmatrix} = 0$$

which implies that

$$Q^{21}Q_{11} + Q^{22}Q_{21} = 0.$$

Therefore,

$$-(Q^{22})^{-1}Q^{21} = Q_{21}Q_{11}^{-1}.$$

So we can write

$$p_0 = Q_{21}Q_{11}^{-1}m_0.$$

which is our formula (**??**).

```
p0_bar = (Q[1, 0]/Q[0, 0]) * msm.M0


print(f'p0_bar = {p0_bar:.4f}')
```

```
p0_bar = 2.2959
```

It can be verified that this formula replicates itself over time in the sense that

$$p_t = Q_{21}Q_{11}^{-1}m_t. \tag{28.21}$$

Now let's visualize the dynamics of m_t, p_t, and R_t starting from different p_0 values to verify our claims above.

We create a function `draw_iterations` to generate the plot

```
p0s = [p0_bar, 2.34, 2.5, 3, 4, 7, 30, 100_000]


draw_iterations(p0s, msm, line_params, num_steps=20)
```

Please notice that for m_t and p_t, we have used log scales for the coordinate (i.e., vertical) axes.

Using log scales allows us to spot distinct constant limiting gross rates of growth R_u^{-1} and R_ℓ^{-1} by eye.
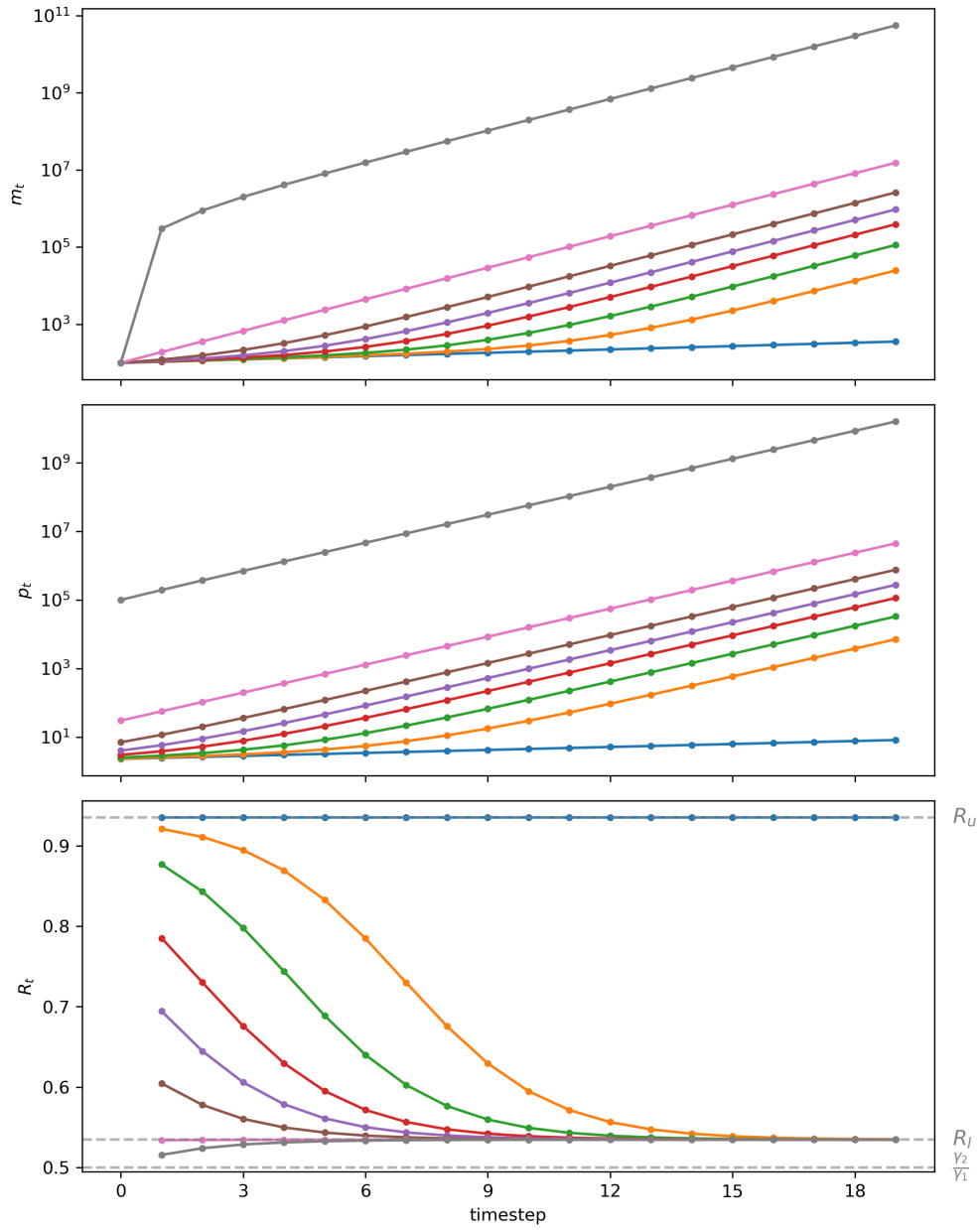
Fig. 28.3: Starting from different initial values of p_0, paths of m_t (top panel, log scale for m), p_t (middle panel, log scale for m), R_t (bottom panel)

28.8 Peculiar stationary outcomes

As promised at the start of this lecture, we have encountered these concepts from macroeconomics:

- an **inflation tax** that a government gathers by printing paper or electronic money

- a dynamic **Laffer curve** in the inflation tax rate that has two stationary equilibria

Staring at the paths of rates of return on the price level in figure Fig. **??** and price levels in Fig. **??** show indicate that almost all paths converge to the *higher* inflation tax rate displayed in the stationary state Laffer curve displayed in figure Fig. **??**.

Thus, we have indeed discovered what we earlier called "perverse" dynamics under rational expectations in which the system converges to the higher of two possible stationary inflation tax rates.

Those dynamics are "perverse" not only in the sense that they imply that the monetary and fiscal authorities that have chosen to finance government expenditures eventually impose a higher inflation tax than required to finance government expenditures, but because of the following "counterintuitive" situation that we can deduce by staring at the stationary state Laffer curve displayed in figure Fig. **??**:

- the figure indicates that inflation can be *reduced* by running *higher* government deficits, i.e., by raising more resources through printing money.

Note: The same qualitative outcomes prevail in this lecture *Inflation Rate Laffer Curves* that studies a nonlinear version of the model in this lecture.

28.9 Equilibrium selection

We have discovered that as a model of price level paths or model is **incomplete** because there is a continuum of "equilibrium" paths for $\{m_{t+1}, p_t\}_{t=0}^{\infty}$ that are consistent with the demand for real balances always equaling the supply.

Through application of our computational methods 1 and 2, we have learned that this continuum can be indexed by choice of one of two scalars:

- for computational method 1, R_0

- for computational method 2, p_0

To apply our model, we have somehow to *complete* it by *selecting* an equilibrium path from among the continuum of possible paths.

We discovered that

- all but one of the equilibrium paths converge to limits in which the higher of two possible stationary inflation tax prevails

- there is a unique equilibrium path associated with "plausible" statements about how reductions in government deficits affect a stationary inflation rate

On grounds of plausibility, we recommend following many macroeconomists in selecting the unique equilibrium that converges to the lower stationary inflation tax rate.

As we shall see, we shall accept this recommendation in lecture *Some Unpleasant Monetarist Arithmetic*.

In lecture, *Laffer Curves with Adaptive Expectations*, we shall explore how [Bruno and Fischer, 1990] and others justified this in other ways.

TWENTYNINE

SOME UNPLEASANT MONETARIST ARITHMETIC

29.1 Overview

This lecture builds on concepts and issues introduced in *Money Financed Government Deficits and Price Levels*.

That lecture describes stationary equilibria that reveal a *Laffer curve* in the inflation tax rate and the associated stationary rate of return on currency.

In this lecture we study a situation in which a stationary equilibrium prevails after date $T > 0$, but not before then.

For $t = 0, \dots, T - 1$, the money supply, price level, and interest-bearing government debt vary along a transition path that ends at $t = T$.

During this transition, the ratio of the real balances $\frac{m_{t+1}}{p_t}$ to indexed one-period government bonds $\tilde{R}B_{t-1}$ maturing at time t decreases each period.

This has consequences for the **gross-of-interest** government deficit that must be financed by printing money for times $t \geq T$.

The critical **money-to-bonds** ratio stabilizes only at time T and afterwards.

And the larger is T, the higher is the gross-of-interest government deficit that must be financed by printing money at times $t \geq T$.

These outcomes are the essential finding of Sargent and Wallace's "unpleasant monetarist arithmetic"[Sargent and Wallace, 1981].

That lecture described supplies and demands for money that appear in lecture.

It also characterized the steady state equilibrium from which we work backwards in this lecture.

In addition to learning about "unpleasant monetarist arithmetic", in this lecture we'll learn how to implement a *fixed point* algorithm for computing an initial price level.

29.2 Setup

Let's start with quick reminders of the model's components set out in *Money Financed Government Deficits and Price Levels*.

Please consult that lecture for more details and Python code that we'll also use in this lecture.

For $t \geq 1$, **real balances** evolve according to

$$\frac{m_{t+1}}{p_t} - \frac{m_t}{p_{t-1}} \frac{p_{t-1}}{p_t} = g$$

or

$$b_t - b_{t-1} R_{t-1} = g \qquad (29.1)$$

where

- $b_t = \frac{m_{t+1}}{p_t}$ is real balances at the end of period t

- $R_{t-1} = \frac{p_{t-1}}{p_t}$ is the gross rate of return on real balances held from $t-1$ to t

The demand for real balances is

$$b_t = \gamma_1 - \gamma_2 R_t^{-1}. \qquad (29.2)$$

where $\gamma_1 > \gamma_2 > 0$.

29.3 Monetary-Fiscal Policy

To the basic model of *Money Financed Government Deficits and Price Levels*, we add inflation-indexed one-period government bonds as an additional way for the government to finance government expenditures.

Let $\widetilde{R} > 1$ be a time-invariant gross real rate of return on government one-period inflation-indexed bonds.

With this additional source of funds, the government's budget constraint at time $t \geq 0$ is now

$$B_t + \frac{m_{t+1}}{p_t} = \widetilde{R} B_{t-1} + \frac{m_t}{p_t} + g$$

Just before the beginning of time 0, the public owns $\check{m}_0$ units of currency (measured in dollars) and $\widetilde{R} \check{B}_{-1}$ units of one-period indexed bonds (measured in time 0 goods); these two quantities are initial conditions set outside the model.

Notice that $\check{m}_0$ is a *nominal* quantity, being measured in dollars, while $\widetilde{R} \check{B}_{-1}$ is a *real* quantity, being measured in time 0 goods.

29.3.1 Open market operations

At time 0, government can rearrange its portfolio of debts subject to the following constraint (on open-market operations):

$$\widetilde{R}B_{-1} + \frac{m_0}{p_0} = \widetilde{R}\check{B}_{-1} + \frac{\check{m}_0}{p_0}$$

or

$$B_{-1} - \check{B}_{-1} = \frac{1}{p_0\widetilde{R}}(\check{m}_0 - m_0) \tag{29.3}$$

This equation says that the government (e.g., the central bank) can *decrease* m_0 relative to $\check{m}_0$ by *increasing* B_{-1} relative to $\check{B}_{-1}$.

This is a version of a standard constraint on a central bank's **open market operations** in which it expands the stock of money by buying government bonds from the public.

29.4 An open market operation at $t = 0$

Following Sargent and Wallace [Sargent and Wallace, 1981], we analyze consequences of a central bank policy that uses an open market operation to lower the price level in the face of a persistent fiscal deficit that takes the form of a positive g.

Just before time 0, the government chooses (m_0, B_{-1}) subject to constraint (**??**).

For $t = 0, 1, \ldots, T-1$,

$$B_t = \widetilde{R}B_{t-1} + g$$
$$m_{t+1} = m_0$$

while for $t \geq T$,

$$B_t = B_{T-1}$$
$$m_{t+1} = m_t + p_t\bar{g}$$

where

$$\bar{g} = \left[(\tilde{R}-1)B_{T-1} + g\right] \tag{29.4}$$

We want to compute an equilibrium $\{p_t, m_t, b_t, R_t\}_{t=0}$ sequence under this scheme for running monetary and fiscal policies.

Here, by **fiscal policy** we mean the collection of actions that determine a sequence of net-of-interest government deficits $\{g_t\}_{t=0}^{\infty}$ that must be financed by issuing to the public either money or interest bearing bonds.

By **monetary policy** or **debt-management policy**, we mean the collection of actions that determine how the government divides its portfolio of debts to the public between interest-bearing parts (government bonds) and non-interest-bearing parts (money).

By an **open market operation**, we mean a government monetary policy action in which the government (or its delegate, say, a central bank) either buys government bonds from the public for newly issued money, or sells bonds to the public and withdraws the money it receives from public circulation.

29.5 Algorithm (basic idea)

We work backwards from $t = T$ and first compute p_T, R_u associated with the low-inflation, low-inflation-tax-rate stationary equilibrium in *Inflation Rate Laffer Curves*.

To start our description of our algorithm, it is useful to recall that a stationary rate of return on currency $\bar{R}$ solves the quadratic equation

$$-\gamma_2 + (\gamma_1 + \gamma_2 - \bar{g})\bar{R} - \gamma_1 \bar{R}^2 = 0 \tag{29.5}$$

Quadratic equation (**??**) has two roots, $R_l < R_u < 1$.

For reasons described at the end of *Money Financed Government Deficits and Price Levels*, we select the larger root R_u.

Next, we compute

$$R_T = R_u$$
$$b_T = \gamma_1 - \gamma_2 R_u^{-1} \tag{29.6}$$
$$p_T = \frac{m_0}{\gamma_1 - \bar{g} - \gamma_2 R_u^{-1}}$$

We can compute continuation sequences $\{R_t, b_t\}_{t=T+1}^{\infty}$ of rates of return and real balances that are associated with an equilibrium by solving equation (**??**) and (**??**) sequentially for $t \geq 1$:

$$b_t = b_{t-1} R_{t-1} + \bar{g}$$
$$R_t^{-1} = \frac{\gamma_1}{\gamma_2} - \gamma_2^{-1} b_t$$
$$p_t = R_t p_{t-1}$$
$$m_t = b_{t-1} p_t$$

29.6 Before time T

Define

$$\lambda \equiv \frac{\gamma_2}{\gamma_1}.$$

Our restrictions that $\gamma_1 > \gamma_2 > 0$ imply that $\lambda \in [0, 1)$.

We want to compute

$$p_0 = \gamma_1^{-1}\left[\sum_{j=0}^{\infty}\lambda^j m_j\right]$$

$$= \gamma_1^{-1}\left[\sum_{j=0}^{T-1}\lambda^j m_0 + \sum_{j=T}^{\infty}\lambda^j m_{1+j}\right]$$

Thus,

$$p_0 = \gamma_1^{-1}m_0\left\{\frac{1-\lambda^T}{1-\lambda} + \frac{\lambda^T}{R_u - \lambda}\right\}$$

$$p_1 = \gamma_1^{-1}m_0\left\{\frac{1-\lambda^{T-1}}{1-\lambda} + \frac{\lambda^{T-1}}{R_u - \lambda}\right\}$$

$$\vdots \qquad \vdots \tag{29.7}$$

$$p_{T-1} = \gamma_1^{-1}m_0\left\{\frac{1-\lambda}{1-\lambda} + \frac{\lambda}{R_u - \lambda}\right\}$$

$$p_T = \gamma_1^{-1}m_0\left\{\frac{1}{R_u - \lambda}\right\}$$

We can implement the preceding formulas by iterating on

$$p_t = \gamma_1^{-1}m_0 + \lambda p_{t+1}, \quad t = T-1, T-2, \dots, 0$$

starting from

$$p_T = \frac{m_0}{\gamma_1 - \bar{g} - \gamma_2 R_u^{-1}} = \gamma_1^{-1}m_0\left\{\frac{1}{R_u - \lambda}\right\} \tag{29.8}$$

Remark 29.6.1

We can verify the equivalence of the two formulas on the right sides of (**??**) by recalling that R_u is a root of the quadratic equation (**??**) that determines steady state rates of return on currency.

29.7 Algorithm (pseudo code)

Now let's describe a computational algorithm in more detail in the form of a description that constitutes pseudo code because it approaches a set of instructions we could provide to a Python coder.

To compute an equilibrium, we deploy the following algorithm.

Algorithm 29.7.1

Given *parameters* include $g, \check{m}_0, \check{B}_{-1}, \widetilde{R} > 1, T$.

We define a mapping from p_0 to $\hat{p}_0$ as follows.

- Set m_0 and then compute B_{-1} to satisfy the constraint on time 0 **open market operations**

$$B_{-1} - \check{B}_{-1} = \frac{\widetilde{R}}{p_0} (\check{m}_0 - m_0)$$

- Compute B_{T-1} from

$$B_{T-1} = \widetilde{R}^T B_{-1} + \left(\frac{1 - \widetilde{R}^T}{1 - \widetilde{R}} \right) g$$

- Compute

$$\bar{g} = g + \left[\tilde{R} - 1 \right] B_{T-1}$$

- Compute R_u, p_T from formulas (**??**) and (**??**) above

- Compute a new estimate of p_0, call it $\hat{p}_0$, from equation (**??**) above

- Note that the preceding steps define a mapping

$$\hat{p}_0 = \mathcal{S}(p_0)$$

- We seek a fixed point of $\mathcal{S}$, i.e., a solution of $p_0 = \mathcal{S}(p_0)$.

- Compute a fixed point by iterating to convergence on the relaxation algorithm

$$p_{0,j+1} = (1 - \theta)\mathcal{S}(p_{0,j}) + \theta p_{0,j},$$

where $\theta \in [0, 1)$ is a relaxation parameter.

29.8 Example Calculations

We'll set parameters of the model so that the steady state after time T is initially the same as in *Inflation Rate Laffer Curves*

In particular, we set $\gamma_1 = 100, \gamma_2 = 50, g = 3.0$. We set $m_0 = 100$ in that lecture, but now the counterpart will be M_T, which is endogenous.

As for new parameters, we'll set $\tilde{R} = 1.01, \check{B}_{-1} = 0, \check{m}_0 = 105, T = 5$.

We'll study a "small" open market operation by setting $m_0 = 100$.

These parameter settings mean that just before time 0, the "central bank" sells the public bonds in exchange for $\check{m}_0 - m_0 = 5$ units of currency.

That leaves the public with less currency but more government interest-bearing bonds.

Since the public has less currency (its supply has diminished) it is plausible to anticipate that the price level at time 0 will be driven downward.

But that is not the end of the story, because this **open market operation** at time 0 has consequences for future settings of m_{t+1} and the gross-of-interest government deficit $\bar{g}_t$.

Let's start with some imports:

```python
import numpy as np
import matplotlib.pyplot as plt
from collections import namedtuple
```

Now let's dive in and implement our pseudo code in Python.

```python
# Create a namedtuple that contains parameters
MoneySupplyModel = namedtuple("MoneySupplyModel",
                              ["γ1", "γ2", "g",
                               "R_tilde", "m0_check", "Bm1_check",
                               "T"])


def create_model(γ1=100, γ2=50, g=3.0,
                 R_tilde=1.01,
                 Bm1_check=0, m0_check=105,
                 T=5):

    return MoneySupplyModel(γ1=γ1, γ2=γ2, g=g,
                            R_tilde=R_tilde,
                            m0_check=m0_check, Bm1_check=Bm1_check,
                            T=T)
```

```python
msm = create_model()
```

```python
def S(p0, m0, model):

    # unpack parameters
    γ1, γ2, g = model.γ1, model.γ2, model.g
    R_tilde = model.R_tilde
    m0_check, Bm1_check = model.m0_check, model.Bm1_check
    T = model.T

    # open market operation
    Bm1 = 1 / (p0 * R_tilde) * (m0_check - m0) + Bm1_check

    # compute B_{T-1}
    BTm1 = R_tilde ** T * Bm1 + ((1 - R_tilde ** T) / (1 - R_tilde)) * g
```

(continues on next page)

```
    # compute g bar
    g_bar = g + (R_tilde - 1) * BTm1

    # solve the quadratic equation
    Ru = np.roots((-γ1, γ1 + γ2 - g_bar, -γ2)).max()

    # compute p0
    λ = γ2 / γ1
    p0_new = (1 / γ1) * m0 * ((1 - λ ** T) / (1 - λ) + λ ** T / (Ru - λ))

    return p0_new
```

```
def compute_fixed_point(m0, p0_guess, model, θ=0.5, tol=1e-6):

    p0 = p0_guess
    error = tol + 1

    while error > tol:
        p0_next = (1 - θ) * S(p0, m0, model) + θ * p0

        error = np.abs(p0_next - p0)
        p0 = p0_next

    return p0
```

Let's look at how price level p_0 in the stationary R_u equilibrium depends on the initial money supply m_0.

Notice that the slope of p_0 as a function of m_0 is constant.

This outcome indicates that our model verifies a quantity theory of money outcome, something that Sargent and Wallace [Sargent and Wallace, 1981] purposefully built into their model to justify the adjective *monetarist* in their title.

```
m0_arr = np.arange(10, 110, 10)
```

```
plt.plot(m0_arr, [compute_fixed_point(m0, 1, msm) for m0 in m0_arr])

plt.ylabel('$p_0$')
plt.xlabel('$m_0$')

plt.show()
```
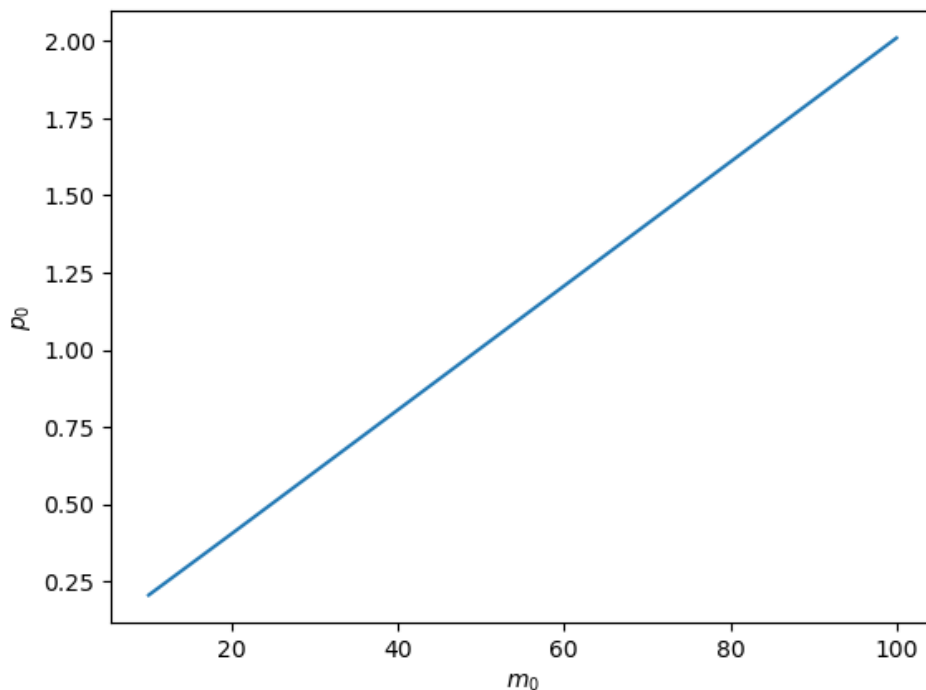
Now let's write and implement code that lets us experiment with the time 0 open market operation described earlier.

```python
def simulate(m0, model, length=15, p0_guess=1):

    # unpack parameters
    γ1, γ2, g = model.γ1, model.γ2, model.g
    R_tilde = model.R_tilde
    m0_check, Bm1_check = model.m0_check, model.Bm1_check
    T = model.T

    # (pt, mt, bt, Rt)
    paths = np.empty((4, length))

    # open market operation
    p0 = compute_fixed_point(m0, 1, model)
    Bm1 = 1 / (p0 * R_tilde) * (m0_check - m0) + Bm1_check
    BTm1 = R_tilde ** T * Bm1 + ((1 - R_tilde ** T) / (1 - R_tilde)) * g
    g_bar = g + (R_tilde - 1) * BTm1
    Ru = np.roots((-γ1, γ1 + γ2 - g_bar, -γ2)).max()

    λ = γ2 / γ1

    # t = 0
    paths[0, 0] = p0
    paths[1, 0] = m0
```

```
    # 1 <= t <= T
    for t in range(1, T+1, 1):
        paths[0, t] = (1 / γ1) * m0 * \
                      ((1 - λ ** (T - t)) / (1 - λ)
                       + (λ ** (T - t) / (Ru - λ)))
        paths[1, t] = m0

    # t > T
    for t in range(T+1, length):
        paths[0, t] = paths[0, t-1] / Ru
        paths[1, t] = paths[1, t-1] + paths[0, t] * g_bar

    # Rt = pt / pt+1
    paths[3, :T] = paths[0, :T] / paths[0, 1:T+1]
    paths[3, T:] = Ru

    # bt = γ1 - γ2 / Rt
    paths[2, :] = γ1 - γ2 / paths[3, :]

    return paths
```

```
def plot_path(m0_arr, model, length=15):

    fig, axs = plt.subplots(2, 2, figsize=(8, 5))
    titles = ['$p_t$', '$m_t$', '$b_t$', '$R_t$']

    for m0 in m0_arr:
        paths = simulate(m0, model, length=length)
        for i, ax in enumerate(axs.flat):
            ax.plot(paths[i])
            ax.set_title(titles[i])

    axs[0, 1].hlines(model.m0_check, 0, length, color='r', linestyle='--')
    axs[0, 1].text(length * 0.8, model.m0_check * 0.9, '$\check{m}_0$')
    plt.show()
```

```
plot_path([80, 100], msm)
```

Fig. **??** summarizes outcomes of two experiments that convey messages of Sargent and Wallace [Sargent and Wallace, 1981].
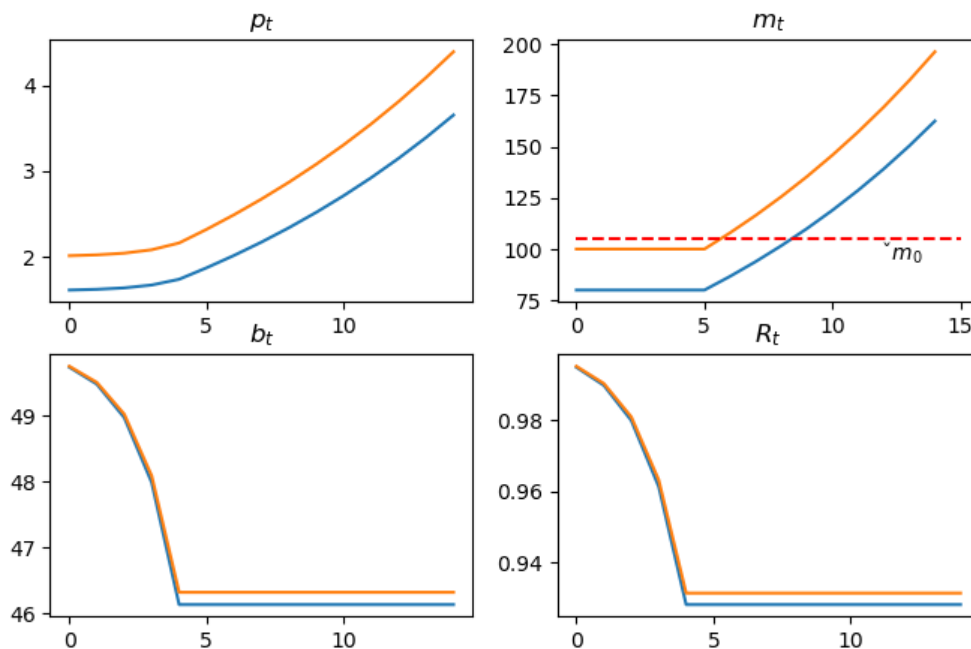
Fig. 29.1: Unpleasant Arithmetic

- An open market operation that reduces the supply of money at time $t = 0$ reduces the price level at time $t = 0$

- The lower is the post-open-market-operation money supply at time 0, lower is the price level at time 0.

- An open market operation that reduces the post open market operation money supply at time 0 also *lowers* the rate of return on money R_u at times $t \geq T$ because it brings a higher gross of interest government deficit that must be financed by printing money (i.e., levying an inflation tax) at time $t \geq T$.

- R is important in the context of maintaining monetary stability and addressing the consequences of increased inflation due to government deficits. Thus, a larger R might be chosen to mitigate the negative impacts on the real rate of return caused by inflation.

INFLATION RATE LAFFER CURVES

30.1 Overview

We study stationary and dynamic *Laffer curves* in the inflation tax rate in a non-linear version of the model studied in *Money Financed Government Deficits and Price Levels*.

We use the log-linear version of the demand function for money that [Cagan, 1956] used in his classic paper in place of the linear demand function used in *Money Financed Government Deficits and Price Levels*.

That change requires that we modify parts of our analysis.

In particular, our dynamic system is no longer linear in state variables.

Nevertheless, the economic logic underlying an analysis based on what we called "method 2" remains unchanged.

We shall discover qualitatively similar outcomes to those that we studied in *Money Financed Government Deficits and Price Levels*.

That lecture presented a linear version of the model in this lecture.

As in that lecture, we discussed these topics:

- an **inflation tax** that a government gathers by printing paper or electronic money

- a dynamic **Laffer curve** in the inflation tax rate that has two stationary equilibria

- perverse dynamics under rational expectations in which the system converges to the higher stationary inflation tax rate

- a peculiar comparative stationary-state analysis connected with that stationary inflation rate that asserts that inflation can be *reduced* by running *higher* government deficits

These outcomes will set the stage for the analysis of *Laffer Curves with Adaptive Expectations* that studies a version of the present model that uses a version of "adaptive expectations" instead of rational expectations.

That lecture will show that

- replacing rational expectations with adaptive expectations leaves the two stationary inflation rates unchanged, but that ...

- it reverses the perverse dynamics by making the *lower* stationary inflation rate the one to which the system typically converges

- a more plausible comparative dynamic outcome emerges in which now inflation can be *reduced* by running *lower* government deficits

30.2 The Model

Let

- m_t be the log of the money supply at the beginning of time t

- p_t be the log of the price level at time t

The demand function for money is

$$m_{t+1} - p_t = -\alpha(p_{t+1} - p_t) \tag{30.1}$$

where $\alpha \geq 0$.

The law of motion of the money supply is

$$\exp(m_{t+1}) - \exp(m_t) = g\exp(p_t) \tag{30.2}$$

where g is the part of government expenditures financed by printing money.

Remark: Please notice that while equation (**??**) is linear in logs of the money supply and price level, equation (**??**) is linear in levels. This will require adapting the equilibrium computation methods that we deployed in *Money Financed Government Deficits and Price Levels*.

30.3 Limiting Values of Inflation Rate

We can compute the two prospective limiting values for $\overline{\pi}$ by studying the steady-state Laffer curve.

Thus, in a *steady state*

$$m_{t+1} - m_t = p_{t+1} - p_t = x \quad \forall t,$$

where $x > 0$ is a common rate of growth of logarithms of the money supply and price level.

A few lines of algebra yields the following equation that x satisfies

$$\exp(-\alpha x) - \exp(-(1+\alpha)x) = g \tag{30.3}$$

where we require that

$$g \leq \max_{x \geq 0}\{\exp(-\alpha x) - \exp(-(1 + \alpha)x)\}, \tag{30.4}$$

so that it is feasible to finance g by printing money.

The left side of (**??**) is steady state revenue raised by printing money.

The right side of (**??**) is the quantity of time t goods that the government raises by printing money.

Soon we'll plot the left and right sides of equation (**??**).

But first we'll write code that computes a steady-state $\bar{\pi}$.

Let's start by importing some libraries

```python
from collections import namedtuple
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
from scipy.optimize import fsolve
```

Let's create a `namedtuple` to store the parameters of the model

```python
CaganLaffer = namedtuple('CaganLaffer',
                        ["m0",   # log of the money supply at t=0
                         "α",    # sensitivity of money demand
                         "λ",
                         "g" ])

# Create a Cagan Laffer model
def create_model(α=0.5, m0=np.log(100), g=0.35):
    return CaganLaffer(α=α, m0=m0, λ=α/(1+α), g=g)

model = create_model()
```

Now we write code that computes steady-state $\bar{\pi}$s.

```python
# Define formula for π_bar
def solve_π(x, α, g):
    return np.exp(-α * x) - np.exp(-(1 + α) * x) - g


def solve_π_bar(model, x0):
    π_bar = fsolve(solve_π, x0=x0, xtol=1e-10, args=(model.α, model.g))[0]
    return π_bar
```

```
# Solve for the two steady state of π
π_l = solve_π_bar(model, x0=0.6)
π_u = solve_π_bar(model, x0=3.0)
print(f'The two steady state of π are: {π_l, π_u}')
```

```
The two steady state of π are: (0.6737147075333032, 1.6930797322614812)
```

We find two steady state $\bar{\pi}$ values.

30.4 Steady State Laffer curve

The following figure plots the steady state Laffer curve together with the two stationary inflation rates.

```python
def compute_seign(x, α):
    return np.exp(-α * x) - np.exp(-(1 + α) * x)


def plot_laffer(model, πs):
    α, g = model.α, model.g

    # Generate π values
    x_values = np.linspace(0, 5, 1000)

    # Compute corresponding seigniorage values for the function
    y_values = compute_seign(x_values, α)

    # Plot the function
    plt.plot(x_values, y_values,
             label=f'Laffer curve')
    for π, label in zip(πs, ['$\pi_l$', '$\pi_u$']):
        plt.text(π, plt.gca().get_ylim()[0]*2,
                 label, horizontalalignment='center',
                 color='brown', size=10)
        plt.axvline(π, color='brown', linestyle='--')
    plt.axhline(g, color='red', linewidth=0.5,
                linestyle='--', label='g')
    plt.xlabel('$\pi$')
    plt.ylabel('seigniorage')
    plt.legend()
    plt.show()

# Steady state Laffer curve
```

```
plot_laffer(model, (π_l, π_u))
```
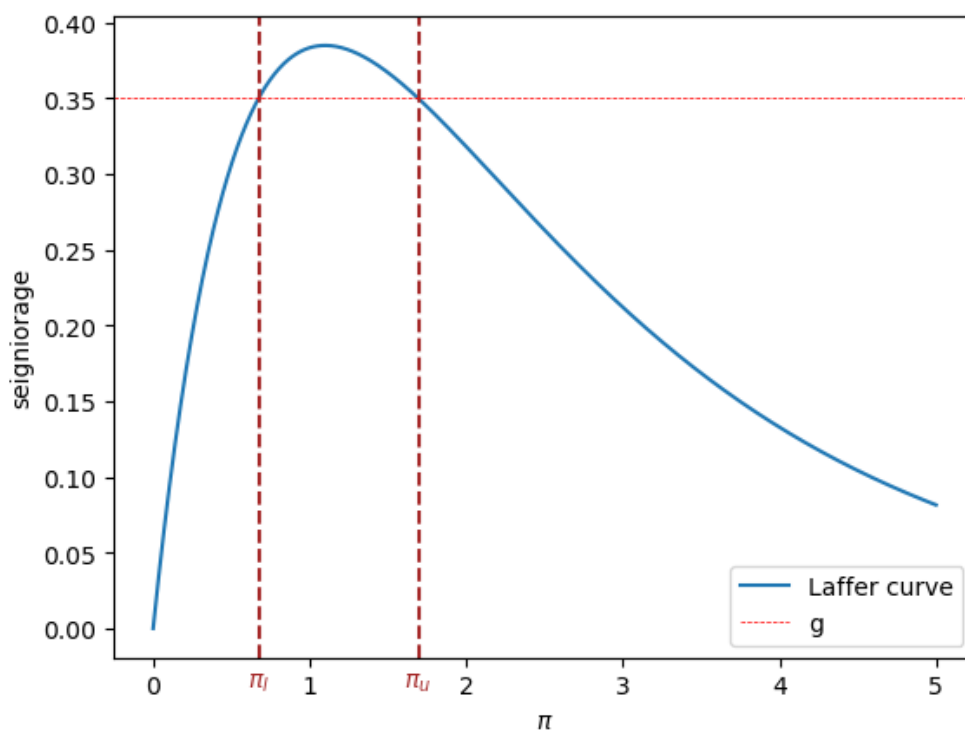

Fig. 30.1: Seigniorage as function of steady state inflation. The dashed brown lines indicate π_l and π_u.

30.5 Initial Price Levels

Now that we have our hands on the two possible steady states, we can compute two functions $\underline{p}(m_0)$ and $\overline{p}(m_0)$, which as initial conditions for p_t at time t, imply that $\pi_t = \bar{\pi}$ for all $t \geq 0$.

The function $\underline{p}(m_0)$ will be associated with π_l the lower steady-state inflation rate.

The function $\overline{p}(m_0)$ will be associated with π_u the lower steady-state inflation rate.

```
def solve_p0(p0, m0, α, g, π):
    return np.log(np.exp(m0) + g * np.exp(p0)) + α * π - p0

def solve_p0_bar(model, x0, π_bar):
    p0_bar = fsolve(solve_p0, x0=x0, xtol=1e-20, args=(model.m0,
                                                        model.α,
```

```
                                                   model.g,
                                                   π_bar))[0]
    return p0_bar


# Compute two initial price levels associated with π_l and π_u
p0_l = solve_p0_bar(model,
                    x0=np.log(220),
                    π_bar=π_l)
p0_u = solve_p0_bar(model,
                    x0=np.log(220),
                    π_bar=π_u)
print(f'Associated initial  p_0s  are: {p0_l, p0_u}')
```

```
Associated initial  p_0s  are: (5.615742247288047, 7.144789784380314)
```

30.5.1 Verification

To start, let's write some code to verify that if the initial log price level p_0 takes one of the two values we just calculated, the inflation rate π_t will be constant for all $t \geq 0$.

The following code verifies this.

```
# Implement pseudo-code above
def simulate_seq(p0, model, num_steps):
    λ, g = model.λ, model.g
    π_seq, μ_seq, m_seq, p_seq = [], [], [model.m0], [p0]

    for t in range(num_steps):

        m_seq.append(np.log(np.exp(m_seq[t]) + g * np.exp(p_seq[t])))
        p_seq.append(1/λ * p_seq[t] + (1 - 1/λ) * m_seq[t+1])

        μ_seq.append(m_seq[t+1]-m_seq[t])
        π_seq.append(p_seq[t+1]-p_seq[t])

    return π_seq, μ_seq, m_seq, p_seq
```

```
π_seq, μ_seq, m_seq, p_seq = simulate_seq(p0_l, model, 150)

# Check π and μ at steady state
print('π_bar == μ_bar:', π_seq[-1] == μ_seq[-1])
```

```
# Check steady state m_{t+1} - m_t and p_{t+1} - p_t
print('m_{t+1} - m_t:', m_seq[-1] - m_seq[-2])
print('p_{t+1} - p_t:', p_seq[-1] - p_seq[-2])

# Check if exp(-ax) - exp(-(1 + a)x) = g
eq_g = lambda x: np.exp(-model.a * x) - np.exp(-(1 + model.a) * x)

print('eq_g == g:', np.isclose(eq_g(m_seq[-1] - m_seq[-2]), model.g))
```

```
π_bar == μ_bar: True
m_{t+1} - m_t: 1.693079732261424
p_{t+1} - p_t: 1.693079732261424
eq_g == g: True
```

30.6 Computing an Equilibrium Sequence

We'll deploy a method similar to *Method 2* used in *Money Financed Government Deficits and Price Levels*.

We'll take the time t state vector to be the pair (m_t, p_t).

We'll treat m_t as a `natural state variable` and p_t as a `jump` variable.

Let

$$\lambda \equiv \frac{\alpha}{1 + \alpha}$$

Let's rewrite equation (**??**) as

$$p_t = (1 - \lambda)m_{t+1} + \lambda p_{t+1} \tag{30.5}$$

We'll summarize our algorithm with the following pseudo-code.

Pseudo-code

The heart of the pseudo-code iterates on the following mapping from state vector (m_t, p_t) at time t to state vector (m_{t+1}, p_{t+1}) at time $t + 1$.

- starting from a given pair (m_t, p_t) at time $t \geq 0$

  - solve (**??**) for m_{t+1}

  - solve (**??**) for $p_{t+1} = \lambda^{-1}p_t + (1 - \lambda^{-1})m_{t+1}$

  - compute the inflation rate $\pi_t = p_{t+1} - p_t$ and growth of money supply $\mu_t = m_{t+1} - m_t$

Next, compute the two functions $\underline{p}(m_0)$ and $\overline{p}(m_0)$ described above

Now initiate the algorithm as follows.

- set $m_0 > 0$

- set a value of $p_0 \in [\underline{p}(m_0), \overline{p}(m_0)]$ and form the pair (m_0, p_0) at time $t = 0$

Starting from (m_0, p_0) iterate on t to convergence of $\pi_t \to \overline{\pi}$ and $\mu_t \to \overline{\mu}$

It will turn out that

- if they exist, limiting values $\overline{\pi}$ and $\overline{\mu}$ will be equal

- if limiting values exist, there are two possible limiting values, one high, one low

- for almost all initial log price levels p_0, the limiting $\overline{\pi} = \overline{\mu}$ is the higher value

- for each of the two possible limiting values $\overline{\pi}$, there is a unique initial log price level p_0 that implies that $\pi_t = \mu_t = \overline{\mu}$ for all $t \geq 0$

    - this unique initial log price level solves $\log(\exp(m_0) + g \exp(p_0)) - p_0 = -\alpha\overline{\pi}$

    - the preceding equation for p_0 comes from $m_1 - p_0 = -\alpha\overline{\pi}$

30.7 Slippery Side of Laffer Curve Dynamics

We are now equipped to compute time series starting from different p_0 settings, like those in *Money Financed Government Deficits and Price Levels*.

```
# Generate a sequence from p0_l to p0_u
p0s = np.arange(p0_l, p0_u, 0.1)

line_params = {'lw': 1.5,
               'marker': 'o',
               'markersize': 3}

p0_bars = (p0_l, p0_u)

draw_iterations(p0s, model, line_params, p0_bars, num_steps=20)
```

Staring at the paths of price levels in Fig. **??** reveals that almost all paths converge to the *higher* inflation tax rate displayed in the stationary state Laffer curve. displayed in figure Fig. **??**.

Thus, we have reconfirmed what we have called the "perverse" dynamics under rational expectations in which the system converges to the higher of two possible stationary inflation tax rates.
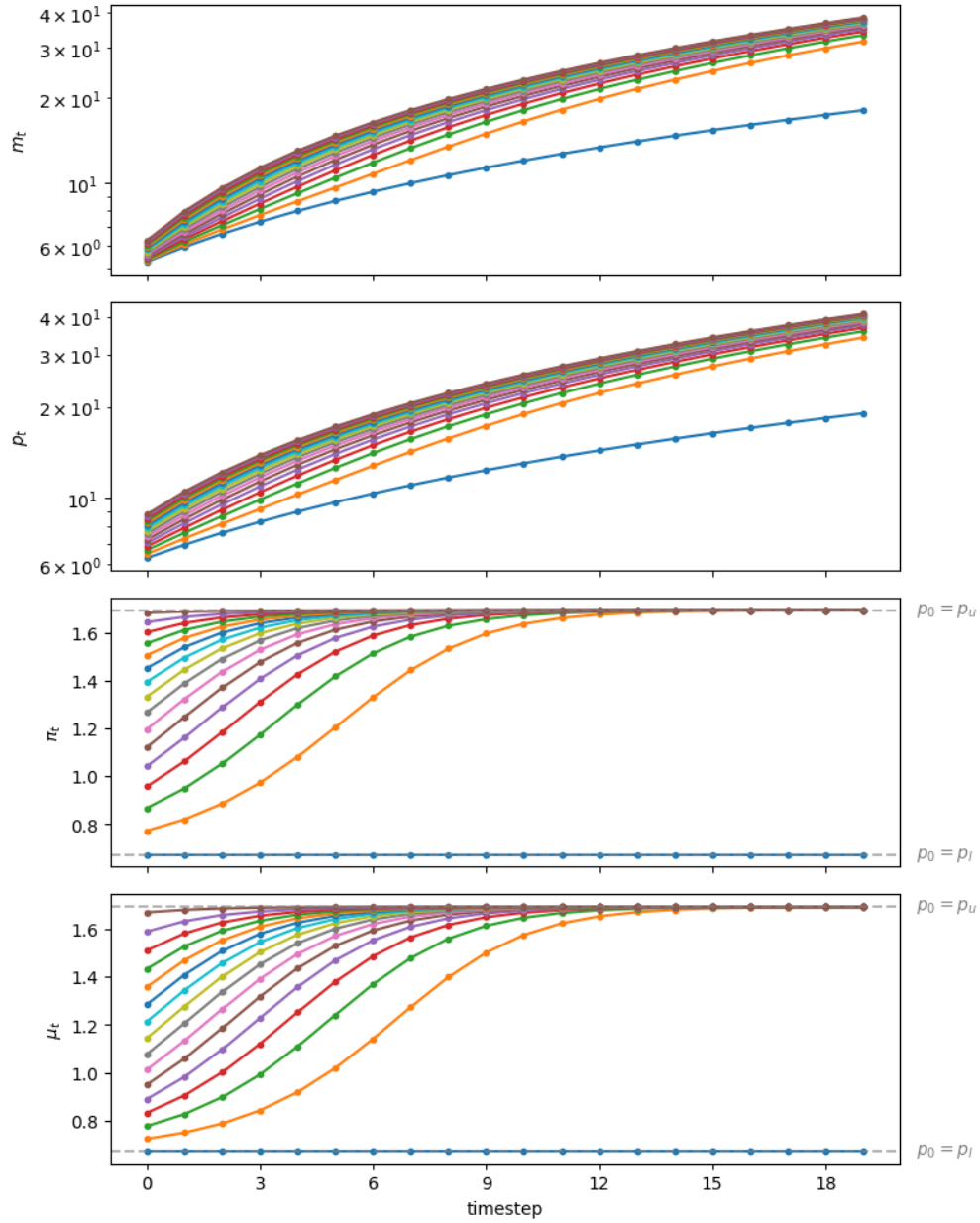
Fig. 30.2: Starting from different initial values of p_0, paths of m_t (top panel, log scale for m), p_t (second panel, log scale for p), π_t (third panel), and μ_t (bottom panel)

Those dynamics are "perverse" not only in the sense that they imply that the monetary and fiscal authorities that have chosen to finance government expenditures eventually impose a higher inflation tax than required to finance government expenditures, but because of the following "counterintuitive" situation that we can deduce by staring at the stationary state Laffer curve displayed in figure Fig. **??**:

- the figure indicates that inflation can be *reduced* by running *higher* government deficits, i.e., by raising more resources through printing money.

Note: The same qualitative outcomes prevail in *Money Financed Government Deficits and Price Levels* that studies a linear version of the model in this lecture.

We discovered that

- all but one of the equilibrium paths converge to limits in which the higher of two possible stationary inflation tax prevails

- there is a unique equilibrium path associated with "plausible" statements about how reductions in government deficits affect a stationary inflation rate

As in *Money Financed Government Deficits and Price Levels*, on grounds of plausibility, we again recommend selecting the unique equilibrium that converges to the lower stationary inflation tax rate.

As we shall see, we accepting this recommendation is a key ingredient of outcomes of the "unpleasant arithmetic" that we describe in *Some Unpleasant Monetarist Arithmetic*.

In *Laffer Curves with Adaptive Expectations*, we shall explore how [Bruno and Fischer, 1990] and others justified our equilibrium selection in other ways.

LAFFER CURVES WITH ADAPTIVE EXPECTATIONS

31.1 Overview

This lecture studies stationary and dynamic **Laffer curves** in the inflation tax rate in a non-linear version of the model studied in this lecture *Money Financed Government Deficits and Price Levels*.

As in the lecture *Money Financed Government Deficits and Price Levels*, this lecture uses the log-linear version of the demand function for money that [Cagan, 1956] used in his classic paper in place of the linear demand function used in this lecture *Money Financed Government Deficits and Price Levels*.

But now, instead of assuming "rational expectations"in the form of "perfect foresight", we'll adopt the "adaptive expectations"assumption used by [Cagan, 1956] and [Friedman, 1956].

This means that instead of assuming that expected inflation π_t^* is described by the "perfect foresight"or "rational expectations"hypothesis

$$\pi_t^* = p_{t+1} - p_t$$

that we adopted in lectures *Money Financed Government Deficits and Price Levels* and lectures *Inflation Rate Laffer Curves*, we'll now assume that π_t^* is determined by the adaptive expectations hypothesis described in equation (**??**) reported below.

We shall discover that changing our hypothesis about expectations formation in this way will change some our findings and leave others intact. In particular, we shall discover that

- replacing rational expectations with adaptive expectations leaves the two stationary inflation rates unchanged, but that ...

- it reverses the perverse dynamics by making the **lower** stationary inflation rate the one to which the system typically converges

- a more plausible comparative dynamic outcome emerges in which now inflation can be **reduced** by running **lower** government deficits

These more plausible comparative dynamics underlie the "old time religion"that states that "inflation is always and everywhere caused by government deficits".

These issues were studied by [Bruno and Fischer, 1990].

Their purpose was to reverse what they thought were counter intuitive predictions of their model under rational expectations (i.e., perfect foresight in this context) by dropping rational expectations and instead assuming that people form expectations about future inflation rates according to the "adaptive expectations" scheme (**??**) described below.

Note: [Marcet and Sargent, 1989] had studied another way of selecting stationary equilibrium that involved replacing rational expectations with a model of learning via least squares regression.
[Marcet and Nicolini, 2003] and [Sargent *et al.*, 2009] extended that work and applied it to study recurrent high-inflation episodes in Latin America.

31.2 The model

Let

- m_t be the log of the money supply at the beginning of time t

- p_t be the log of the price level at time t

- π_t^* be the public's expectation of the rate of inflation between t and $t+1$

The law of motion of the money supply is

$$\exp(m_{t+1}) - \exp(m_t) = g \exp(p_t) \tag{31.1}$$

where g is the part of government expenditures financed by printing money.

Notice that equation (**??**) implies that

$$m_{t+1} = \log[\exp(m_t) + g \exp(p_t)] \tag{31.2}$$

The demand function for money is

$$m_{t+1} - p_t = -\alpha \pi_t^* \tag{31.3}$$

where $\alpha \geq 0$.

Expectations of inflation are governed by

$$\pi_t^* = (1 - \delta)(p_t - p_{t-1}) + \delta \pi_{t-1}^* \tag{31.4}$$

where $\delta \in (0, 1)$

31.3 Computing an equilibrium sequence

Equation the expressions for m_{t+1} provided by (**??**) and (**??**) and use equation (**??**) to eliminate π_t^* to obtain the following equation for p_t:

$$\log[\exp(m_t) + g \exp(p_t)] - p_t = -\alpha[(1-\delta)(p_t - p_{t-1}) + \delta\pi_{t-1}^*] \tag{31.5}$$

Pseudo-code

Here is the pseudo-code for our algorithm.

Starting at time 0 with initial conditions $(m_0, \pi_{-1}^*, p_{-1})$, for each $t \geq 0$ deploy the following steps in order:

- solve (**??**) for p_t

- solve equation (**??**) for π_t^*

- solve equation (**??**) for m_{t+1}

This completes the algorithm.

31.4 Claims or conjectures

It will turn out that

- if they exist, limiting values $\bar{\pi}$ and $\bar{\mu}$ will be equal

- if limiting values exist, there are two possible limiting values, one high, one low

- unlike the outcome in lecture *Inflation Rate Laffer Curves*, for almost all initial log price levels and expected inflation rates p_0, π_t^*, the limiting $\bar{\pi} = \bar{\mu}$ is the **lower** steady state value

- for each of the two possible limiting values $\bar{\pi}$,there is a unique initial log price level p_0 that implies that $\pi_t = \mu_t = \bar{\mu}$ for all $t \geq 0$

    - this unique initial log price level solves $\log(\exp(m_0) + g \exp(p_0)) - p_0 = -\alpha\bar{\pi}$

    - the preceding equation for p_0 comes from $m_1 - p_0 = -\alpha\bar{\pi}$

31.5 Limiting values of inflation rate

As in our earlier lecture *Inflation Rate Laffer Curves*, we can compute the two prospective limiting values for $\bar{\pi}$ by studying the steady-state Laffer curve.

Thus, in a **steady state**

$$m_{t+1} - m_t = p_{t+1} - p_t = x \quad \forall t,$$

where $x > 0$ is a common rate of growth of logarithms of the money supply and price level.

A few lines of algebra yields the following equation that x satisfies

$$\exp(-\alpha x) - \exp(-(1+\alpha)x) = g \tag{31.6}$$

where we require that

$$g \leq \max_{x : x \geq 0} \exp(-\alpha x) - \exp(-(1+\alpha)x), \tag{31.7}$$

so that it is feasible to finance g by printing money.

The left side of (**??**) is steady state revenue raised by printing money.

The right side of (**??**) is the quantity of time t goods that the government raises by printing money.

Soon we'll plot the left and right sides of equation (**??**).

But first we'll write code that computes a steady-state $\bar{\pi}$.

Let's start by importing some libraries

```python
from collections import namedtuple
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
from matplotlib.cm import get_cmap
from matplotlib.colors import to_rgba
import matplotlib
from scipy.optimize import root, fsolve
```

Let's create a `namedtuple` to store the parameters of the model

```python
LafferAdaptive = namedtuple('LafferAdaptive',
                        ["m0",  # log of the money supply at t=0
                         "α",   # sensitivity of money demand
                         "g",   # government expenditure
                         "δ"])

# Create a Cagan Laffer model
def create_model(α=0.5, m0=np.log(100), g=0.35, δ=0.9):
    return LafferAdaptive(α=α, m0=m0, g=g, δ=δ)

model = create_model()
```

Now we write code that computes steady-state $\bar{\pi}$s.

```python
# Define formula for π_bar
def solve_π(x, α, g):
    return np.exp(-α * x) - np.exp(-(1 + α) * x) - g


def solve_π_bar(model, x0):
    π_bar = fsolve(solve_π, x0=x0, xtol=1e-10, args=(model.α, model.g))[0]
    return π_bar


# Solve for the two steady state of π
π_l = solve_π_bar(model, x0=0.6)
π_u = solve_π_bar(model, x0=3.0)
print(f'The two steady state of π are: {π_l, π_u}')
```

```
The two steady state of π are: (0.6737147075333032, 1.6930797322614812)
```

We find two steady state $\bar{\pi}$ values

31.6 Steady-state Laffer curve

The following figure plots the steady-state Laffer curve together with the two stationary inflation rates.

```python
def compute_seign(x, α):
    return np.exp(-α * x) - np.exp(-(1 + α) * x)


def plot_laffer(model, πs):
    α, g = model.α, model.g

    # Generate π values
    x_values = np.linspace(0, 5, 1000)

    # Compute corresponding seigniorage values for the function
    y_values = compute_seign(x_values, α)

    # Plot the function
    plt.plot(x_values, y_values,
            label=f'$exp((-{α})x) - exp(- (1- {α}) x)$')
    for π, label in zip(πs, ['$\pi_l$', '$\pi_u$']):
        plt.text(π, plt.gca().get_ylim()[0]*2,
                label, horizontalalignment='center',
                color='brown', size=10)
        plt.axvline(π, color='brown', linestyle='--')
```

(continues on next page)

```
    plt.axhline(g, color='red', linewidth=0.5,
                linestyle='--', label='g')
    plt.xlabel('$\pi$')
    plt.ylabel('seigniorage')
    plt.legend()
    plt.grid(True)
    plt.show()

# Steady state Laffer curve
plot_laffer(model, (π_l, π_u))
```
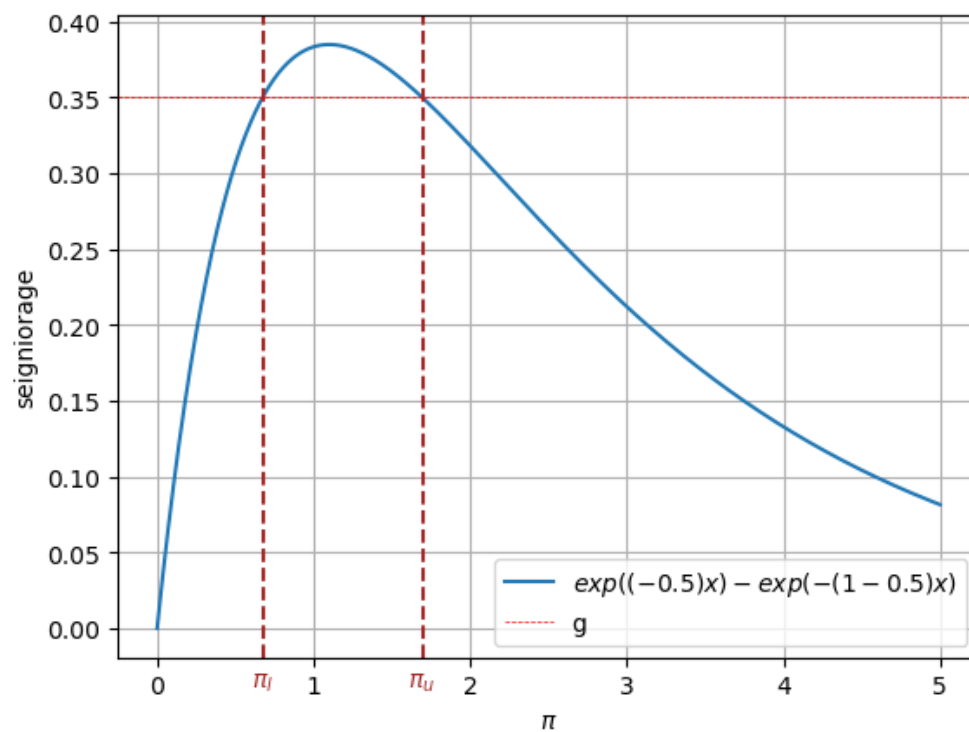

Fig. 31.1: Seigniorage as function of steady-state inflation. The dashed brown lines indicate π_l and π_u.

31.7 Associated initial price levels

Now that we have our hands on the two possible steady states, we can compute two initial log price levels p_{-1}, which as initial conditions, imply that $\pi_t = \bar{\pi}$ for all $t \geq 0$.

In particular, to initiate a fixed point of the dynamic Laffer curve dynamics, we set

$$p_{-1} = m_0 + \alpha \pi^*$$

```python
def solve_p_init(model, π_star):
    m0, α = model.m0, model.α
    return m0 + α*π_star


# Compute two initial price levels associated with π_l and π_u
p_l, p_u = map(lambda π: solve_p_init(model, π), (π_l, π_u))
print('Associated initial p_{-1}s', f'are: {p_l, p_u}')
```

```
Associated initial p_{-1}s are: (4.9420275397547435, 5.451710052118832)
```

31.7.1 Verification

To start, let's write some code to verify that if we initial π_{-1}^*, p_{-1} appropriately, the inflation rate π_t will be constant for all $t \geq 0$ (at either π_u or π_l depending on the initial condition)

The following code verifies this.

```python
def solve_laffer_adapt(p_init, π_init, model, num_steps):
    m0, α, δ, g = model.m0, model.α, model.δ, model.g

    m_seq = np.nan * np.ones(num_steps+1)
    π_seq = np.nan * np.ones(num_steps)
    p_seq = np.nan * np.ones(num_steps)
    μ_seq = np.nan * np.ones(num_steps)

    m_seq[1] = m0
    π_seq[0] = π_init
    p_seq[0] = p_init

    for t in range(1, num_steps):
        # Solve p_t
        def p_t(pt):
```

```
            return np.log(np.exp(m_seq[t]) + g * np.exp(pt)) \
                        - pt + α * ((1-δ)*(pt - p_seq[t-1]) + δ*π_seq[t-1])

        p_seq[t] = root(fun=p_t, x0=p_seq[t-1]).x[0]

        # Solve π_t
        π_seq[t] = (1-δ) * (p_seq[t]-p_seq[t-1]) + δ*π_seq[t-1]

        # Solve m_t
        m_seq[t+1] = np.log(np.exp(m_seq[t]) + g*np.exp(p_seq[t]))

        # Solve μ_t
        μ_seq[t] = m_seq[t+1] - m_seq[t]


    return π_seq, μ_seq, m_seq, p_seq
```

Compute limiting values starting from p_{-1} associated with π_l

```
π_seq, μ_seq, m_seq, p_seq = solve_laffer_adapt(p_l, π_l, model, 50)

# Check steady state m_{t+1} - m_t and p_{t+1} - p_t
print('m_{t+1} - m_t:', m_seq[-1] - m_seq[-2])
print('p_{t+1} - p_t:', p_seq[-1] - p_seq[-2])

# Check if exp(-ax) - exp(-(1 + a)x) = g
eq_g = lambda x: np.exp(-model.α * x) - np.exp(-(1 + model.α) * x)

print('eq_g == g:', np.isclose(eq_g(m_seq[-1] - m_seq[-2]), model.g))
```

```
m_{t+1} - m_t: 0.6737147075332999
p_{t+1} - p_t: 0.6737147075332928
eq_g == g: True
```

Compute limiting values starting from p_{-1} associated with π_u

```
π_seq, μ_seq, m_seq, p_seq = solve_laffer_adapt(p_u, π_u, model, 50)

# Check steady state m_{t+1} - m_t and p_{t+1} - p_t
print('m_{t+1} - m_t:', m_seq[-1] - m_seq[-2])
print('p_{t+1} - p_t:', p_seq[-1] - p_seq[-2])

# Check if exp(-ax) - exp(-(1 + a)x) = g
```

```
eq_g = lambda x: np.exp(-model.α * x) - np.exp(-(1 + model.α) * x)


print('eq_g == g:', np.isclose(eq_g(m_seq[-1] - m_seq[-2]), model.g))
```

```
m_{t+1} - m_t: 1.69307973225105
p_{t+1} - p_t: 1.6930797322506947
eq_g == g: True
```

31.8 Slippery side of Laffer curve dynamics

We are now equipped to compute time series starting from different p_{-1}, π_{-1}^* settings, analogous to those in this lecture *Money Financed Government Deficits and Price Levels* and this lecture *Inflation Rate Laffer Curves*.

Now we'll study how outcomes unfold when we start p_{-1}, π_{-1}^* away from a stationary point of the dynamic Laffer curve, i.e., away from either π_u or π_l.

To construct a perturbation pair $\check{p}_{-1}, \check{\pi}_{-1}^*$ we'll implement the following pseudo code:

- set $\check{\pi}_{-1}^*$ not equal to one of the stationary points π_u or π_l.

- set $\check{p}_{-1} = m_0 + \alpha\check{\pi}_{-1}^*$

Let's simulate the result generated by varying the initial π_{-1} and corresponding p_{-1}

```
πs = np.linspace(π_l, π_u, 10)


line_params = {'lw': 1.5,
               'marker': 'o',
               'markersize': 3}


π_bars = (π_l, π_u)
draw_iterations(πs, model, line_params, π_bars, num_steps=80)
```
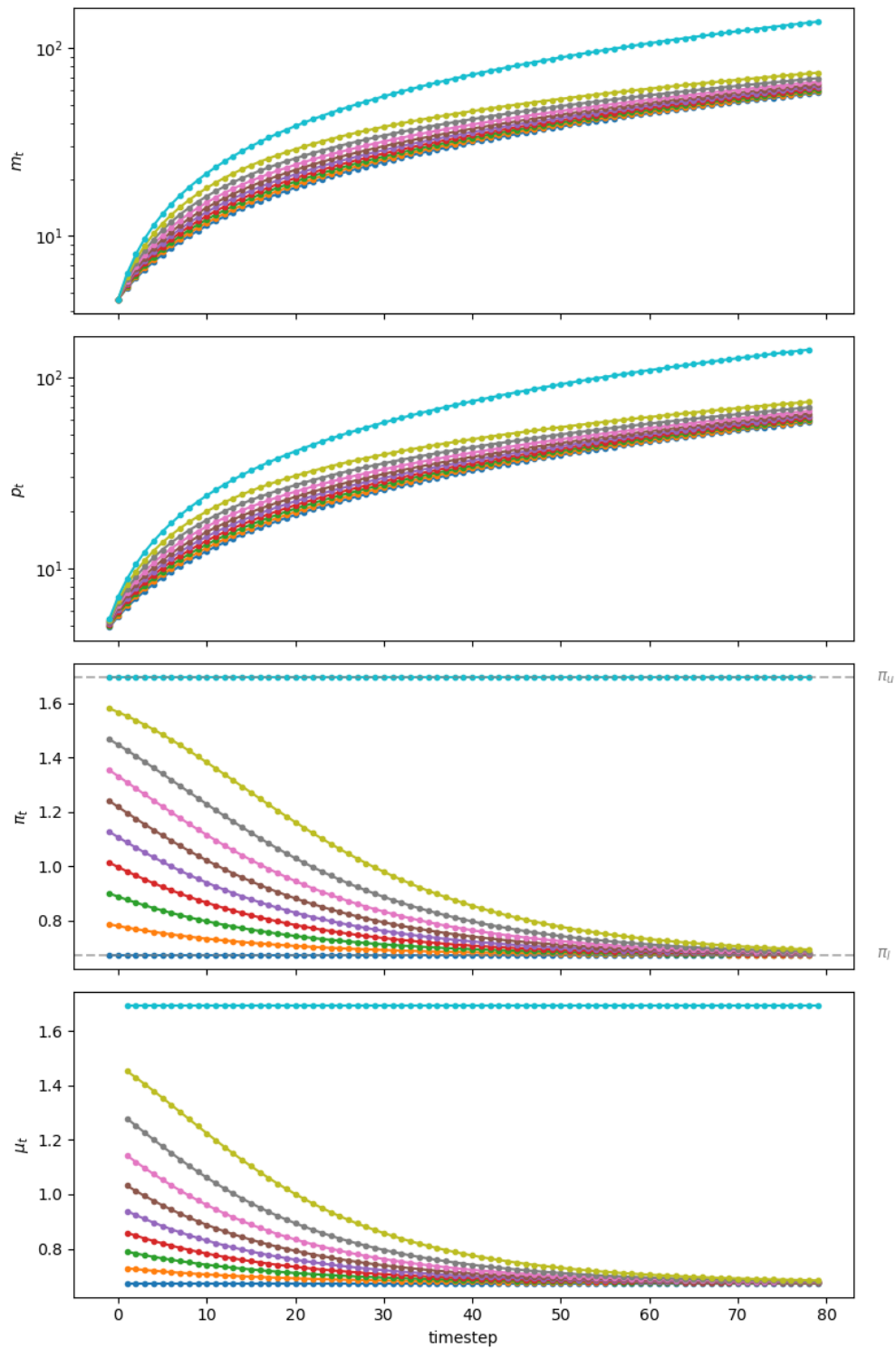
Fig. 31.2: Starting from different initial values of π_0, paths of m_t (top panel, log scale for m), p_t (second panel, log scale for p), π_t (third panel), and μ_t (bottom panel)

Part IX

Stochastic Dynamics

AR(1) PROCESSES

32.1 Overview

In this lecture we are going to study a very simple class of stochastic models called AR(1) processes.

These simple models are used again and again in economic research to represent the dynamics of series such as

- labor income

- dividends

- productivity, etc.

We are going to study AR(1) processes partly because they are useful and partly because they help us understand important concepts.

Let's start with some imports:

```python
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (11, 5)  #set default figure size
```

32.2 The AR(1) model

The **AR(1) model** (autoregressive model of order 1) takes the form

$$X_{t+1} = aX_t + b + cW_{t+1} \tag{32.1}$$

where a, b, c are scalar-valued parameters

(Equation (**??**) is sometimes called a **stochastic difference equation**.)

Example 32.2.1

For example, X_t might be

- the log of labor income for a given household, or

- the log of money demand in a given economy.

In either case, (**??**) shows that the current value evolves as a linear function of the previous value and an IID shock W_{t+1}.

(We use $t + 1$ for the subscript of W_{t+1} because this random variable is not observed at time t.)

The specification (**??**) generates a time series $\{X_t\}$ as soon as we specify an initial condition X_0.

To make things even simpler, we will assume that

- the process $\{W_t\}$ is *IID* and standard normal,

- the initial condition X_0 is drawn from the normal distribution $N(\mu_0, v_0)$ and

- the initial condition X_0 is independent of $\{W_t\}$.

32.2.1 Moving average representation

Iterating backwards from time t, we obtain

$$X_t = aX_{t-1} + b + cW_t = a^2 X_{t-2} + ab + acW_{t-1} + b + cW_t = a^3 X_{t-3} + a^2 b + a^2 cW_{t-2} + b + cW_t = \cdots$$

If we work all the way back to time zero, we get

$$X_t = a^t X_0 + b \sum_{j=0}^{t-1} a^j + c \sum_{j=0}^{t-1} a^j W_{t-j} \tag{32.2}$$

Equation (**??**) shows that X_t is a well defined random variable, the value of which depends on

- the parameters,

- the initial condition X_0 and

- the shocks $W_1, \dots W_t$ from time $t = 1$ to the present.

Throughout, the symbol ψ_t will be used to refer to the density of this random variable X_t.

32.2.2 Distribution dynamics

One of the nice things about this model is that it's so easy to trace out the sequence of distributions $\{\psi_t\}$ corresponding to the time series $\{X_t\}$.

To see this, we first note that X_t is normally distributed for each t.

This is immediate from (**??**), since linear combinations of independent normal random variables are normal.

Given that X_t is normally distributed, we will know the full distribution ψ_t if we can pin down its first two moments.

Let μ_t and v_t denote the mean and variance of X_t respectively.

We can pin down these values from (**??**) or we can use the following recursive expressions:

$$\mu_{t+1} = a\mu_t + b \quad \text{and} \quad v_{t+1} = a^2 v_t + c^2 \tag{32.3}$$

These expressions are obtained from (**??**) by taking, respectively, the expectation and variance of both sides of the equality.

In calculating the second expression, we are using the fact that X_t and W_{t+1} are independent.

(This follows from our assumptions and (**??**).)

Given the dynamics in (**??**) and initial conditions μ_0, v_0, we obtain μ_t, v_t and hence

$$\psi_t = N(\mu_t, v_t)$$

The following code uses these facts to track the sequence of marginal distributions $\{\psi_t\}$.

The parameters are

```
a, b, c = 0.9, 0.1, 0.5
```

```
mu, v = -3.0, 0.6  # initial conditions mu_0, v_0
```

Here's the sequence of distributions:

```
from scipy.stats import norm

sim_length = 10
grid = np.linspace(-5, 7, 120)

fig, ax = plt.subplots()

for t in range(sim_length):
    mu = a * mu + b
    v = a**2 * v + c**2
    ax.plot(grid, norm.pdf(grid, loc=mu, scale=np.sqrt(v)),
            label=f"$\psi_{t}$",
            alpha=0.7)

ax.legend(bbox_to_anchor=[1.05,1],loc=2,borderaxespad=1)

plt.show()
```
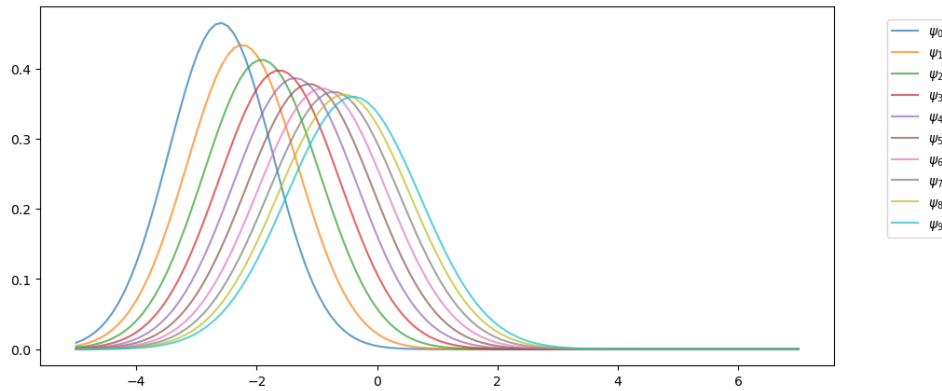
32.3 Stationarity and asymptotic stability

When we use models to study the real world, it is generally preferable that our models have clear, sharp predictions.

For dynamic problems, sharp predictions are related to stability.

For example, if a dynamic model predicts that inflation always converges to some kind of steady state, then the model gives a sharp prediction.

(The prediction might be wrong, but even this is helpful, because we can judge the quality of the model.)

Notice that, in the figure above, the sequence $\{\psi_t\}$ seems to be converging to a limiting distribution, suggesting some kind of stability.

This is even clearer if we project forward further into the future:

```python
def plot_density_seq(ax, mu_0=-3.0, v_0=0.6, sim_length=40):
    mu, v = mu_0, v_0
    for t in range(sim_length):
        mu = a * mu + b
        v = a**2 * v + c**2
        ax.plot(grid,
                norm.pdf(grid, loc=mu, scale=np.sqrt(v)),
                alpha=0.5)

fig, ax = plt.subplots()
plot_density_seq(ax)
plt.show()
```
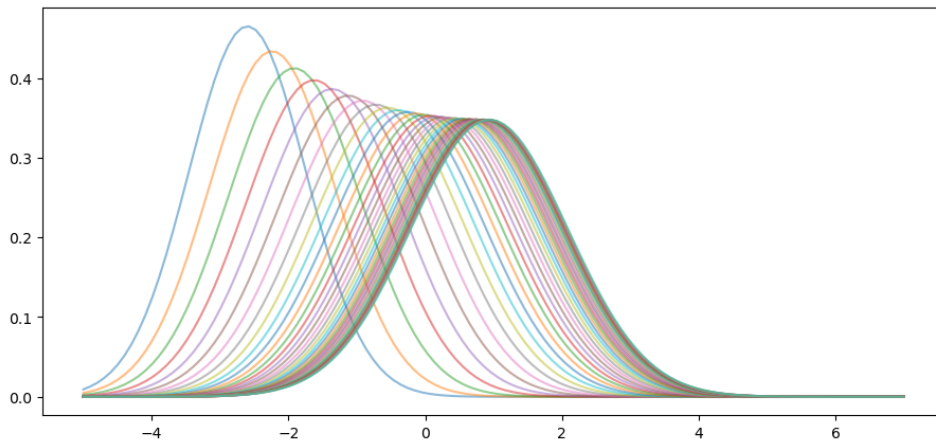
Moreover, the limit does not depend on the initial condition.

For example, this alternative density sequence also converges to the same limit.

```
fig, ax = plt.subplots()
plot_density_seq(ax, mu_0=4.0)
plt.show()
```
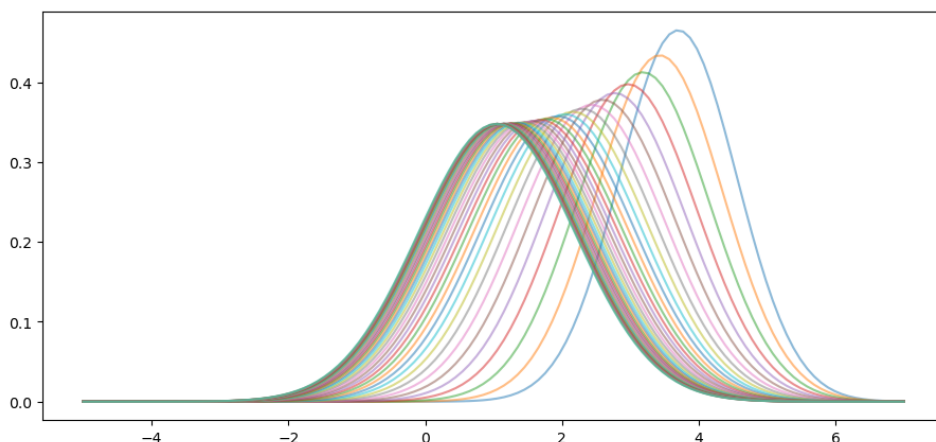

In fact it's easy to show that such convergence will occur, regardless of the initial condition, whenever $|a| < 1$.

To see this, we just have to look at the dynamics of the first two moments, as given in (**??**).

When $|a| < 1$, these sequences converge to the respective limits

$$\mu^* := \frac{b}{1-a} \quad \text{and} \quad v^* = \frac{c^2}{1-a^2} \tag{32.4}$$

(See our *lecture on one dimensional dynamics* for background on deterministic convergence.)

Hence

$$\psi_t \to \psi^* = N(\mu^*, v^*) \quad \text{as } t \to \infty \tag{32.5}$$

We can confirm this is valid for the sequence above using the following code.

32.3. Stationarity and asymptotic stability

```
fig, ax = plt.subplots()
plot_density_seq(ax, mu_0=4.0)

mu_star = b / (1 - a)
std_star = np.sqrt(c**2 / (1 - a**2))   # square root of v_star
psi_star = norm.pdf(grid, loc=mu_star, scale=std_star)
ax.plot(grid, psi_star, 'k-', lw=2, label="$\psi^*$")
ax.legend()

plt.show()
```
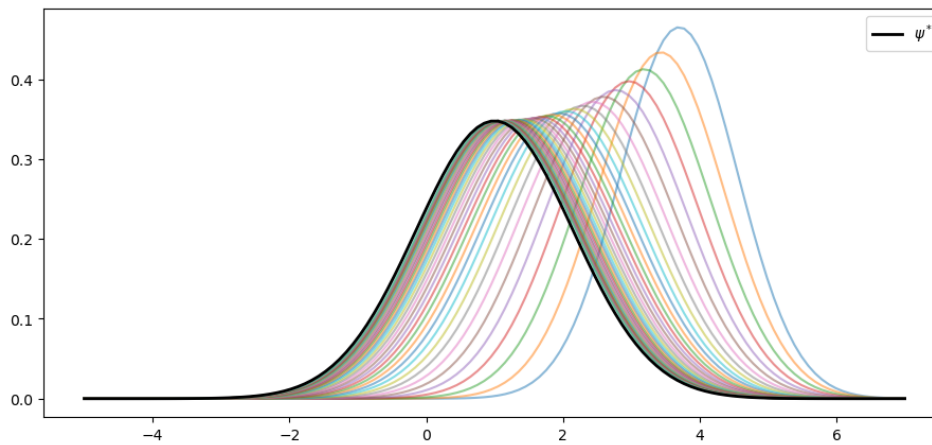

As claimed, the sequence $\{\psi_t\}$ converges to ψ^*.

We see that, at least for these parameters, the AR(1) model has strong stability properties.

32.3.1 Stationary distributions

Let's try to better understand the limiting distribution ψ^*.

A stationary distribution is a distribution that is a "fixed point" of the update rule for the AR(1) process.

In other words, if ψ_t is stationary, then $\psi_{t+j} = \psi_t$ for all j in $\mathbb{N}$.

A different way to put this, specialized to the current setting, is as follows: a density ψ on $\mathbb{R}$ is **stationary** for the AR(1) process if

$$X_t \sim \psi \quad \Longrightarrow \quad aX_t + b + cW_{t+1} \sim \psi$$

The distribution ψ^* in (**??**) has this property —checking this is an exercise.

(Of course, we are assuming that $|a| < 1$ so that ψ^* is well defined.)

In fact, it can be shown that no other distribution on $\mathbb{R}$ has this property.

Thus, when $|a| < 1$, the AR(1) model has exactly one stationary density and that density is given by ψ^*.

32.4 Ergodicity

The concept of ergodicity is used in different ways by different authors.

One way to understand it in the present setting is that a version of the law of large numbers is valid for $\{X_t\}$, even though it is not IID.

In particular, averages over time series converge to expectations under the stationary distribution.

Indeed, it can be proved that, whenever $|a| < 1$, we have

$$\frac{1}{m} \sum_{t=1}^{m} h(X_t) \to \int h(x)\psi^*(x)dx \quad \text{as } m \to \infty \tag{32.6}$$

whenever the integral on the right hand side is finite and well defined.

Notes:

- In (**??**), convergence holds with probability one.

- The textbook by [Meyn and Tweedie, 2009] is a classic reference on ergodicity.

Example 32.4.1

If we consider the identity function $h(x) = x$, we get

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

In other words, the time series sample mean converges to the mean of the stationary distribution.

Ergodicity is important for a range of reasons.

For example, (**??**) can be used to test theory.

In this equation, we can use observed data to evaluate the left hand side of (**??**).

And we can use a theoretical AR(1) model to calculate the right hand side.

If $\frac{1}{m}\sum_{t=1}^{m} X_t$ is not close to $\psi^(x)$, even for many observations, then our theory seems to be incorrect and we will need to revise it.

32.5 Exercises

Exercise 32.5.1

Let k be a natural number.

The k-th central moment of a random variable is defined as

$$M_k := \mathbb{E}[(X - \mathbb{E}X)^k]$$

When that random variable is $N(\mu, \sigma^2)$, it is known that

$$M_k = \begin{cases} 0 & \text{if } k \text{ is odd} \\ \sigma^k (k-1)!! & \text{if } k \text{ is even} \end{cases}$$

Here $n!!$ is the double factorial.

According to (**??**), we should have, for any $k \in \mathbb{N}$,

$$\frac{1}{m} \sum_{t=1}^{m} (X_t - \mu^*)^k \approx M_k$$

when m is large.

Confirm this by simulation at a range of k using the default parameters from the lecture.

Solution to Exercise 32.5.1

Here is one solution:

```python
from numba import njit
from scipy.special import factorial2


@njit
def sample_moments_ar1(k, m=100_000, mu_0=0.0, sigma_0=1.0, seed=1234):
    np.random.seed(seed)
    sample_sum = 0.0
    x = mu_0 + sigma_0 * np.random.randn()
    for t in range(m):
        sample_sum += (x - mu_star)**k
        x = a * x + b + c * np.random.randn()
    return sample_sum / m

def true_moments_ar1(k):
```

```
    if k % 2 == 0:
        return std_star**k * factorial2(k - 1)
    else:
        return 0


k_vals = np.arange(6) + 1
sample_moments = np.empty_like(k_vals)
true_moments = np.empty_like(k_vals)


for k_idx, k in enumerate(k_vals):
    sample_moments[k_idx] = sample_moments_ar1(k)
    true_moments[k_idx] = true_moments_ar1(k)


fig, ax = plt.subplots()
ax.plot(k_vals, true_moments, label="true moments")
ax.plot(k_vals, sample_moments, label="sample moments")
ax.legend()


plt.show()
```
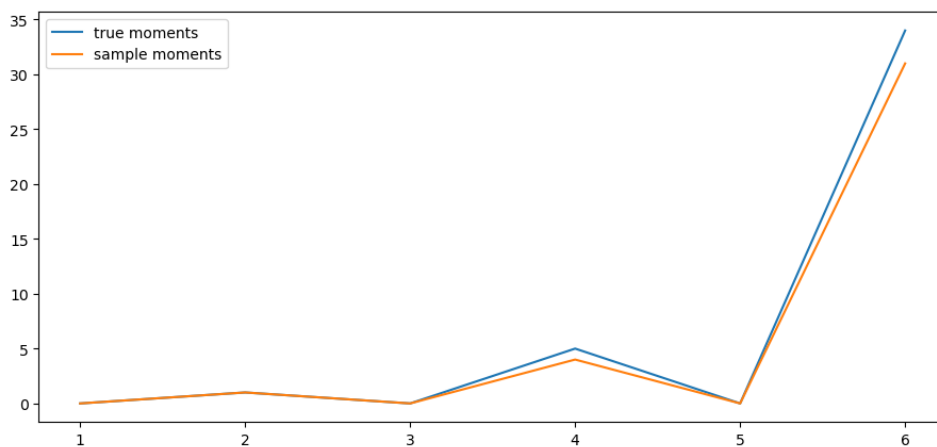

Exercise 32.5.2

Write your own version of a one dimensional kernel density estimator, which estimates a density from a sample.

Write it as a class that takes the data X and bandwidth h when initialized and provides a method f such that

$$f(x) = \frac{1}{hn} \sum_{i=1}^{n} K\left(\frac{x - X_i}{h}\right)$$

For K use the Gaussian kernel (K is the standard normal density).

Write the class so that the bandwidth defaults to Silverman's rule (see the "rule of thumb"discussion on this page). Test the class you have written by going through the steps

1. simulate data $X_1, \ldots, X_n$ from distribution ϕ

2. plot the kernel density estimate over a suitable range

3. plot the density of ϕ on the same figure

for distributions ϕ of the following types

- beta distribution with $\alpha = \beta = 2$

- beta distribution with $\alpha = 2$ and $\beta = 5$

- beta distribution with $\alpha = \beta = 0.5$

Use $n = 500$.

Make a comment on your results. (Do you think this is a good estimator of these distributions?)

Solution to Exercise 32.5.2

Here is one solution:

```python
K = norm.pdf

class KDE:

    def __init__(self, x_data, h=None):

        if h is None:
            c = x_data.std()
            n = len(x_data)
            h = 1.06 * c * n**(-1/5)
        self.h = h
        self.x_data = x_data

    def f(self, x):
        if np.isscalar(x):
            return K((x - self.x_data) / self.h).mean() * (1/self.h)
        else:
            y = np.empty_like(x)
            for i, x_val in enumerate(x):
                y[i] = K((x_val - self.x_data) / self.h).mean() * (1/self.h)
            return y
```

```
def plot_kde(ϕ, x_min=-0.2, x_max=1.2):
    x_data = ϕ.rvs(n)
    kde = KDE(x_data)

    x_grid = np.linspace(-0.2, 1.2, 100)
    fig, ax = plt.subplots()
    ax.plot(x_grid, kde.f(x_grid), label="estimate")
    ax.plot(x_grid, ϕ.pdf(x_grid), label="true density")
    ax.legend()
    plt.show()
```

```
from scipy.stats import beta

n = 500
parameter_pairs= (2, 2), (2, 5), (0.5, 0.5)
for α, β in parameter_pairs:
    plot_kde(beta(α, β))
```
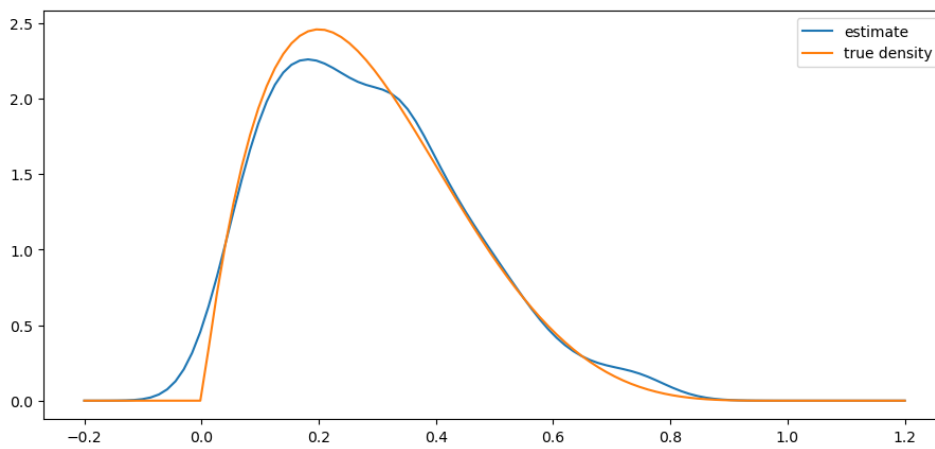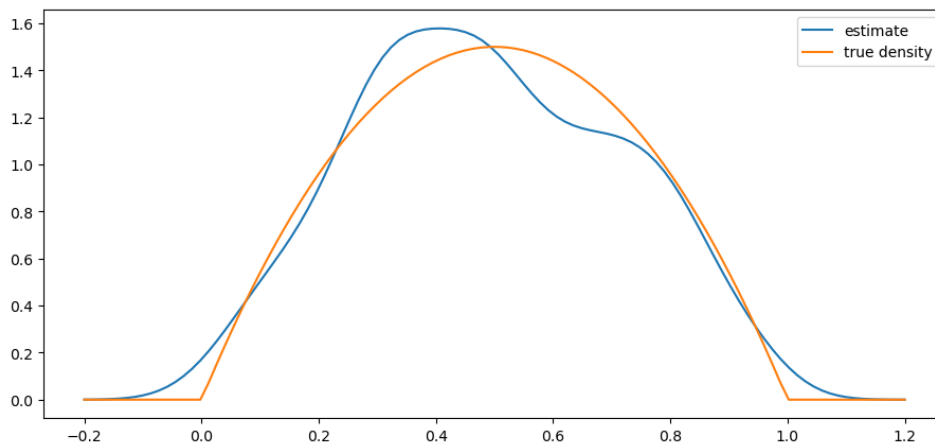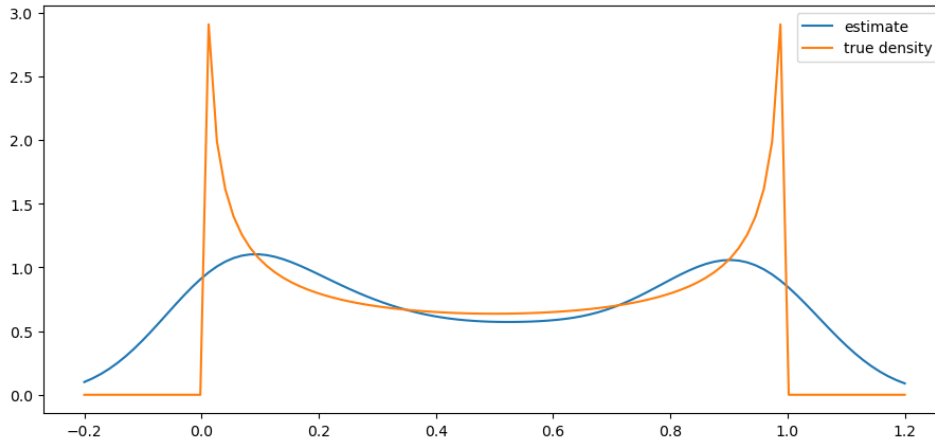
We see that the kernel density estimator is effective when the underlying distribution is smooth but less so otherwise.

Exercise 32.5.3

In the lecture we discussed the following fact: for the $AR(1)$ process

$$X_{t+1} = aX_t + b + cW_{t+1}$$

with $\{W_t\}$ iid and standard normal,

$$\psi_t = N(\mu, s^2) \implies \psi_{t+1} = N(a\mu + b, a^2 s^2 + c^2)$$

Confirm this, at least approximately, by simulation. Let

- $a = 0.9$

- $b = 0.0$

- $c = 0.1$

- $\mu = -3$

- $s = 0.2$

First, plot ψ_t and ψ_{t+1} using the true distributions described above.

Second, plot ψ_{t+1} on the same figure (in a different color) as follows:

1. Generate n draws of X_t from the $N(\mu, s^2)$ distribution

2. Update them all using the rule $X_{t+1} = aX_t + b + cW_{t+1}$

3. Use the resulting sample of X_{t+1} values to produce a density estimate via kernel density estimation.

Try this for $n = 2000$ and confirm that the simulation based estimate of ψ_{t+1} does converge to the theoretical distribution.

Solution to Exercise 32.5.3

Here is our solution

```
a = 0.9
b = 0.0
c = 0.1
μ = -3
s = 0.2
```

```
μ_next = a * μ + b
s_next = np.sqrt(a**2 * s**2 + c**2)
```

```
ψ = lambda x: K((x - μ) / s)
ψ_next = lambda x: K((x - μ_next) / s_next)
```

```
ψ = norm(μ, s)
ψ_next = norm(μ_next, s_next)
```

```
n = 2000
x_draws = ψ.rvs(n)
x_draws_next = a * x_draws + b + c * np.random.randn(n)
kde = KDE(x_draws_next)

x_grid = np.linspace(μ - 1, μ + 1, 100)
fig, ax = plt.subplots()

ax.plot(x_grid, ψ.pdf(x_grid), label="$\psi_t$")
ax.plot(x_grid, ψ_next.pdf(x_grid), label="$\psi_{t+1}$")
ax.plot(x_grid, kde.f(x_grid), label="estimate of $\psi_{t+1}$")

ax.legend()
plt.show()
```
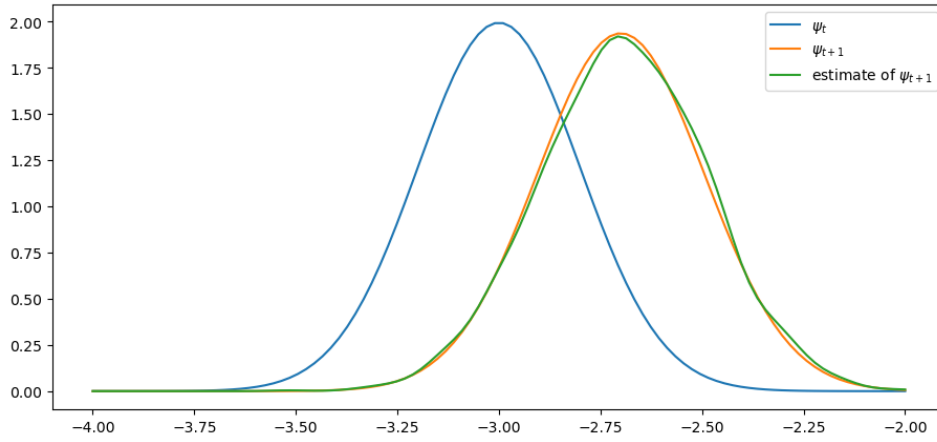
The simulated distribution approximately coincides with the theoretical distribution, as predicted.

MARKOV CHAINS: BASIC CONCEPTS

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

```
!pip install quantecon
```

33.1 Overview

Markov chains provide a way to model situations in which the past casts shadows on the future.

By this we mean that observing measurements about a present situation can help us forecast future situations.

This can be possible when there are statistical dependencies among measurements of something taken at different points of time.

For example,

- inflation next year might co-vary with inflation this year

- unemployment next month might co-vary with unemployment this month

Markov chains are a workhorse for economics and finance.

The theory of Markov chains is beautiful and provides many insights into probability and dynamics.

In this lecture, we will

- review some of the key ideas from the theory of Markov chains and

- show how Markov chains appear in some economic applications.

Let's start with some standard imports:

```
import matplotlib.pyplot as plt
import quantecon as qe
```

(continues on next page)

```python
import numpy as np
import networkx as nx
from matplotlib import cm
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.animation import FuncAnimation
from IPython.display import HTML
from matplotlib.patches import Polygon
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
```

```
OMP: Info #276: omp_set_nested routine deprecated, please use omp_set_max_active_
↪levels instead.
```

33.2 Definitions and examples

In this section we provide some definitions and elementary examples.

33.2.1 Stochastic matrices

Recall that a **probability mass function** over n possible outcomes is a nonnegative n-vector p that sums to one.

For example, $p = (0.2, 0.2, 0.6)$ is a probability mass function over 3 outcomes.

A **stochastic matrix** (or **Markov matrix**) is an $n \times n$ square matrix P such that each row of P is a probability mass function over n outcomes.

In other words,

1. each element of P is nonnegative, and

2. each row of P sums to one

If P is a stochastic matrix, then so is the k-th power P^k for all $k \in \mathbb{N}$.

You are asked to check this in an exercise below.

33.2.2 Markov chains

Now we can introduce Markov chains.

Before defining a Markov chain rigorously, we'll give some examples.

Example 1

From US unemployment data, Hamilton [Hamilton, 2005] estimated the following dynamics.
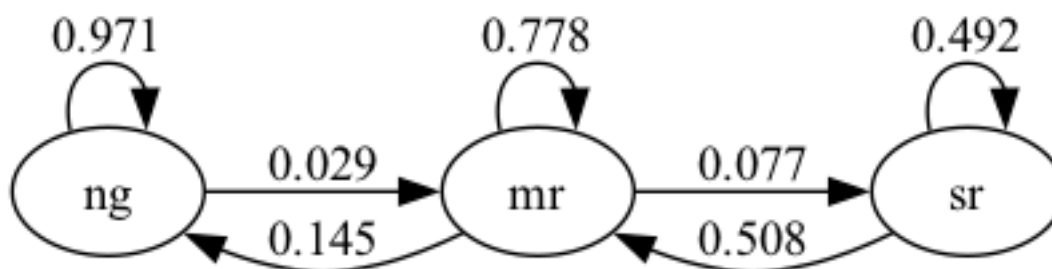
Here there are three **states**

- "ng" represents normal growth

- "mr" represents mild recession

- "sr" represents severe recession

The arrows represent transition probabilities over one month.

For example, the arrow from mild recession to normal growth has 0.145 next to it.

This tells us that, according to past data, there is a 14.5% probability of transitioning from mild recession to normal growth in one month.

The arrow from normal growth back to normal growth tells us that there is a 97% probability of transitioning from normal growth to normal growth (staying in the same state).

Note that these are conditional probabilities —the probability of transitioning from one state to another (or staying at the same one) conditional on the current state.

To make the problem easier to work with numerically, let's convert states to numbers.

In particular, we agree that

- state 0 represents normal growth

- state 1 represents mild recession

- state 2 represents severe recession

Let X_t record the value of the state at time t.

Now we can write the statement "there is a 14.5% probability of transitioning from mild recession to normal growth in one month" as

$$\mathbb{P}\{X_{t+1} = 0 \mid X_t = 1\} = 0.145$$

We can collect all of these conditional probabilities into a matrix, as follows

$$P = \begin{bmatrix} 0.971 & 0.029 & 0 \\ 0.145 & 0.778 & 0.077 \\ 0 & 0.508 & 0.492 \end{bmatrix}$$

Notice that P is a stochastic matrix.

Now we have the following relationship

$$P(i, j) = \mathbb{P}\{X_{t+1} = j \mid X_t = i\}$$

This holds for any i, j between 0 and 2.

In particular, $P(i, j)$ is the probability of transitioning from state i to state j in one month.

Example 2

Consider a worker who, at any given time t, is either unemployed (state 0) or employed (state 1).

Suppose that, over a one-month period,

1. the unemployed worker finds a job with probability $\alpha \in (0, 1)$.

2. the employed worker loses her job and becomes unemployed with probability $\beta \in (0, 1)$.

Given the above information, we can write out the transition probabilities in matrix form as

$$P = \begin{bmatrix} 1 - \alpha & \alpha \\ \beta & 1 - \beta \end{bmatrix} \tag{33.1}$$

For example,

$$P(0, 1) = \text{ probability of transitioning from state 0 to state 1 in one month}$$
$$= \text{ probability finding a job next month}$$
$$= \alpha$$

Suppose we can estimate the values α and β.

Then we can address a range of questions, such as

- What is the average duration of unemployment?

- Over the long-run, what fraction of the time does a worker find herself unemployed?

- Conditional on employment, what is the probability of becoming unemployed at least once over the next 12 months?

We'll cover some of these applications below.

Example 3

Imam and Temple [Imam and Temple, 2023] categorize political institutions into three types: democracy (D), autocracy (A), and an intermediate state called anocracy (N).

Each institution can have two potential development regimes: collapse (C) and growth (G). This results in six possible states: DG, DC, NG, NC, AG and AC.

Imam and Temple [Imam and Temple, 2023] estimate the following transition probabilities:

$$P := \begin{bmatrix} 0.86 & 0.11 & 0.03 & 0.00 & 0.00 & 0.00 \\ 0.52 & 0.33 & 0.13 & 0.02 & 0.00 & 0.00 \\ 0.12 & 0.03 & 0.70 & 0.11 & 0.03 & 0.01 \\ 0.13 & 0.02 & 0.35 & 0.36 & 0.10 & 0.04 \\ 0.00 & 0.00 & 0.09 & 0.11 & 0.55 & 0.25 \\ 0.00 & 0.00 & 0.09 & 0.15 & 0.26 & 0.50 \end{bmatrix}$$

```
nodes = ['DG', 'DC', 'NG', 'NC', 'AG', 'AC']
P = [[0.86, 0.11, 0.03, 0.00, 0.00, 0.00],
     [0.52, 0.33, 0.13, 0.02, 0.00, 0.00],
     [0.12, 0.03, 0.70, 0.11, 0.03, 0.01],
     [0.13, 0.02, 0.35, 0.36, 0.10, 0.04],
     [0.00, 0.00, 0.09, 0.11, 0.55, 0.25],
     [0.00, 0.00, 0.09, 0.15, 0.26, 0.50]]
```

Here is a visualization, with darker colors indicating higher probability.

```
/var/folders/zy/dpt7pvt11fvcm_yxjd7kmv880000gn/T/ipykernel_53712/2625744094.py:16:␣
 ↪DeprecationWarning: `alltrue` is deprecated as of NumPy 1.25.0, and will be␣
 ↪removed in NumPy 2.0. Please use `all` instead.
  edges = nx.draw_networkx_edges(G, pos, ax=ax, connectionstyle=f'arc3, rad = {arc_
 ↪rad}', edge_cmap=cm.Blues, width=2,
```
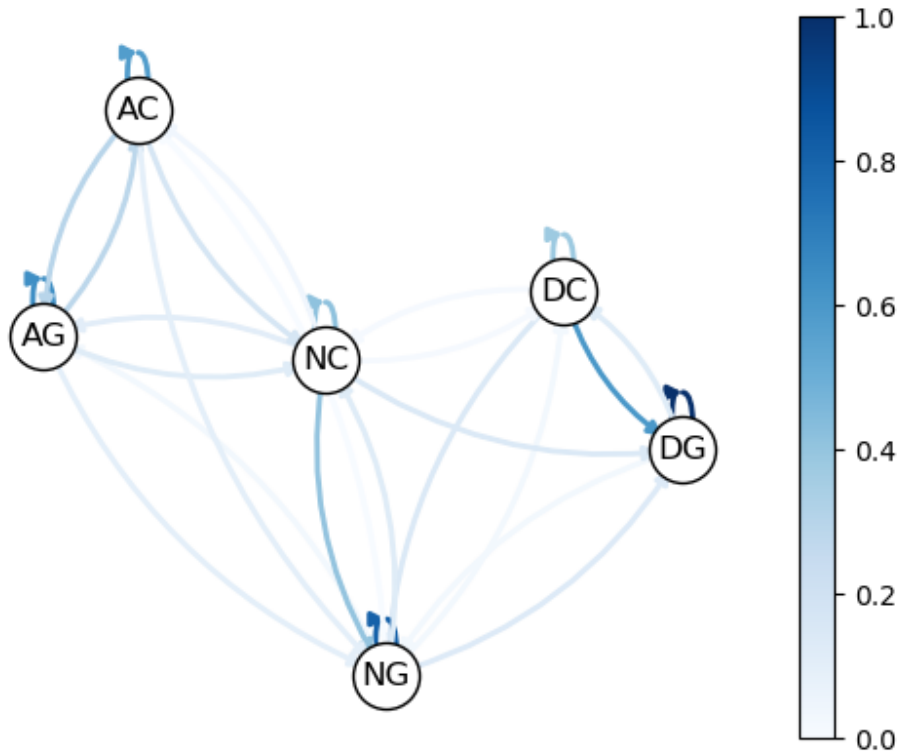
Looking at the data, we see that democracies tend to have longer-lasting growth regimes compared to autocracies (as indicated by the lower probability of transitioning from growth to growth in autocracies).

We can also find a higher probability from collapse to growth in democratic regimes.

33.2.3 Defining Markov chains

So far we've given examples of Markov chains but we haven't defined them.

Let's do that now.

To begin, let S be a finite set $\{x_1, \dots, x_n\}$ with n elements.

The set S is called the **state space** and $x_1, \dots, x_n$ are the **state values**.

A **distribution** ψ on S is a probability mass function of length n, where $\psi(i)$ is the amount of probability allocated to state x_i.

A **Markov chain** $\{X_t\}$ on S is a sequence of random variables taking values in S that have the **Markov property**.

This means that, for any date t and any state $y \in S$,

$$\mathbb{P}\{X_{t+1} = y \mid X_t\} = \mathbb{P}\{X_{t+1} = y \mid X_t, X_{t-1}, \dots\} \tag{33.2}$$

This means that once we know the current state X_t, adding knowledge of earlier states X_{t-1}, X_{t-2} provides no additional information about probabilities of *future* states.

Thus, the dynamics of a Markov chain are fully determined by the set of **conditional probabilities**

$$P(x, y) := \mathbb{P}\{X_{t+1} = y \mid X_t = x\} \qquad (x, y \in S) \tag{33.3}$$

By construction,

- $P(x, y)$ is the probability of going from x to y in one unit of time (one step)

- $P(x, \cdot)$ is the conditional distribution of X_{t+1} given $X_t = x$

We can view P as a stochastic matrix where

$$P_{ij} = P(x_i, x_j) \qquad 1 \leq i, j \leq n$$

Going the other way, if we take a stochastic matrix P, we can generate a Markov chain $\{X_t\}$ as follows:

- draw X_0 from a distribution ψ_0 on S

- for each $t = 0, 1, ...$, draw X_{t+1} from $P(X_t, \cdot)$

By construction, the resulting process satisfies (**??**).

33.3 Simulation

A good way to study Markov chains is to simulate them.

Let's start by doing this ourselves and then look at libraries that can help us.

In these exercises, we'll take the state space to be $S = 0, ..., n-1$.

(We start at 0 because Python arrays are indexed from 0.)

33.3.1 Writing our own simulation code

To simulate a Markov chain, we need

1. a stochastic matrix P and

2. a probability mass function ψ_0 of length n from which to draw an initial realization of X_0.

The Markov chain is then constructed as follows:

1. At time $t = 0$, draw a realization of X_0 from the distribution ψ_0.

2. At each subsequent time t, draw a realization of the new state X_{t+1} from $P(X_t, \cdot)$.

(That is, draw from row X_t of P.)

To implement this simulation procedure, we need a method for generating draws from a discrete distribution.

For this task, we'll use `random.draw` from QuantEcon.py.

To use `random.draw`, we first need to convert the probability mass function to a cumulative distribution

```
ψ_0 = (0.3, 0.7)              # probabilities over {0, 1}
cdf = np.cumsum(ψ_0)          # convert into cumulative distribution
qe.random.draw(cdf, 5)    # generate 5 independent draws from ψ
```

```
array([1, 1, 1, 1, 1])
```

We'll write our code as a function that accepts the following three arguments

- A stochastic matrix `P`.

- An initial distribution `ψ_0`.

- A positive integer `ts_length` representing the length of the time series the function should return.

```python
def mc_sample_path(P, ψ_0=None, ts_length=1_000):

    # set up
    P = np.asarray(P)
    X = np.empty(ts_length, dtype=int)

    # Convert each row of P into a cdf
    P_dist = np.cumsum(P, axis=1)   # Convert rows into cdfs

    # draw initial state, defaulting to 0
    if ψ_0 is not None:
        X_0 = qe.random.draw(np.cumsum(ψ_0))
    else:
        X_0 = 0

    # simulate
    X[0] = X_0
    for t in range(ts_length - 1):
        X[t+1] = qe.random.draw(P_dist[X[t], :])

    return X
```

Let's see how it works using the small matrix

```
P = [[0.4, 0.6],
     [0.2, 0.8]]
```

Here's a short time series.

```
mc_sample_path(P, ψ_0=(1.0, 0.0), ts_length=10)
```

```
array([0, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

It can be shown that for a long series drawn from P, the fraction of the sample that takes value 0 will be about 0.25.

(We will explain why *later*.)

Moreover, this is true regardless of the initial distribution from which X_0 is drawn.

The following code illustrates this

```
X = mc_sample_path(P, ψ_0=(0.1, 0.9), ts_length=1_000_000)
np.mean(X == 0)
```

```
0.250664
```

You can try changing the initial distribution to confirm that the output is always close to 0.25 (for the P matrix above).

33.3.2 Using QuantEcon's routines

QuantEcon.py has routines for handling Markov chains, including simulation.

Here's an illustration using the same P as the preceding example

```
mc = qe.MarkovChain(P)
X = mc.simulate(ts_length=1_000_000)
np.mean(X == 0)
```

```
0.250747
```

The simulate routine is faster (because it is JIT compiled).

```
%time mc_sample_path(P, ts_length=1_000_000) # Our homemade code version
```

```
CPU times: user 533 ms, sys: 3.58 ms, total: 537 ms
Wall time: 537 ms
```

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

```
%time mc.simulate(ts_length=1_000_000) # qe code version
```

```
CPU times: user 18 ms, sys: 1.36 ms, total: 19.3 ms
Wall time: 19.2 ms
```

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

Adding state values and initial conditions

If we wish to, we can provide a specification of state values to `MarkovChain`.

These state values can be integers, floats, or even strings.

The following code illustrates

```
mc = qe.MarkovChain(P, state_values=('unemployed', 'employed'))
mc.simulate(ts_length=4, init='employed')  # Start at employed initial state
```

```
array(['employed', 'employed', 'unemployed', 'employed'], dtype='<U10')
```

```
mc.simulate(ts_length=4, init='unemployed')  # Start at unemployed initial state
```

```
array(['unemployed', 'employed', 'employed', 'employed'], dtype='<U10')
```

```
mc.simulate(ts_length=4)  # Start at randomly chosen initial state
```

```
array(['employed', 'employed', 'employed', 'employed'], dtype='<U10')
```

If we want to see indices rather than state values as outputs as we can use

```
mc.simulate_indices(ts_length=4)
```

```
array([1, 0, 1, 1])
```

33.4 Distributions over time

We learned that

1. $\{X_t\}$ is a Markov chain with stochastic matrix P

2. the distribution of X_t is known to be ψ_t

What then is the distribution of X_{t+1}, or, more generally, of X_{t+m}?

To answer this, we let ψ_t be the distribution of X_t for $t = 0, 1, 2, ...$

Our first aim is to find ψ_{t+1} given ψ_t and P.

To begin, pick any $y \in S$.

To get the probability of being at y tomorrow (at $t+1$), we account for all ways this can happen and sum their probabilities.

This leads to

$$\mathbb{P}\{X_{t+1} = y\} = \sum_{x \in S} \mathbb{P}\{X_{t+1} = y \mid X_t = x\} \cdot \mathbb{P}\{X_t = x\}$$

(We are using the law of total probability.)

Rewriting this statement in terms of marginal and conditional probabilities gives

$$\psi_{t+1}(y) = \sum_{x \in S} P(x, y)\psi_t(x)$$

There are n such equations, one for each $y \in S$.

If we think of ψ_{t+1} and ψ_t as row vectors, these n equations are summarized by the matrix expression

$$\psi_{t+1} = \psi_t P \tag{33.4}$$

Thus, we postmultiply by P to move a distribution forward one unit of time.

By postmultiplying m times, we move a distribution forward m steps into the future.

Hence, iterating on (**??**), the expression $\psi_{t+m} = \psi_t P^m$ is also valid —here P^m is the m-th power of P.

As a special case, we see that if ψ_0 is the initial distribution from which X_0 is drawn, then $\psi_0 P^m$ is the distribution of X_m.

This is very important, so let's repeat it

$$X_0 \sim \psi_0 \quad \implies \quad X_m \sim \psi_0 P^m \tag{33.5}$$

The general rule is that postmultiplying a distribution by P^m shifts it forward m units of time.

Hence the following is also valid.

$$X_t \sim \psi_t \quad \implies \quad X_{t+m} \sim \psi_t P^m \tag{33.6}$$

33.4.1 Multiple step transition probabilities

We know that the probability of transitioning from x to y in one step is $P(x, y)$.

It turns out that the probability of transitioning from x to y in m steps is $P^m(x, y)$, the (x, y)-th element of the m-th power of P.

To see why, consider again (**??**), but now with a ψ_t that puts all probability on state x.

Then ψ_t is a vector with 1 in position x and zero elsewhere.

Inserting this into (**??**), we see that, conditional on $X_t = x$, the distribution of X_{t+m} is the x-th row of P^m.

In particular

$$\mathbb{P}\{X_{t+m} = y \,|\, X_t = x\} = P^m(x, y) = (x, y)\text{-th element of } P^m$$

33.4.2 Example: probability of recession

Recall the stochastic matrix P for recession and growth *considered above*.

Suppose that the current state is unknown —perhaps statistics are available only at the *end* of the current month.

We guess that the probability that the economy is in state x is $\psi_t(x)$ at time t.

The probability of being in recession (either mild or severe) in 6 months time is given by

$$(\psi_t P^6)(1) + (\psi_t P^6)(2)$$

33.4.3 Example 2: cross-sectional distributions

The distributions we have been studying can be viewed either

1. as probabilities or

2. as cross-sectional frequencies that the law of large numbers leads us to anticipate for large samples.

To illustrate, recall our model of employment/unemployment dynamics for a given worker *discussed above*.

Consider a large population of workers, each of whose lifetime experience is described by the specified dynamics, with each worker's outcomes being realizations of processes that are statistically independent of all other workers' processes.

Let ψ_t be the current *cross-sectional* distribution over $\{0, 1\}$.

The cross-sectional distribution records fractions of workers employed and unemployed at a given moment t.

- For example, $\psi_t(0)$ is the unemployment rate at time t.

What will the cross-sectional distribution be in 10 periods hence?

The answer is $\psi_t P^{10}$, where P is the stochastic matrix in (**??**).

This is because each worker's state evolves according to P, so $\psi_t P^{10}$ is a marginal distribution for a single randomly selected worker.

But when the sample is large, outcomes and probabilities are roughly equal (by an application of the law of large numbers).

So for a very large (tending to infinite) population, $\psi_t P^{10}$ also represents fractions of workers in each state.

This is exactly the cross-sectional distribution.

33.5 Stationary distributions

As seen in (**??**), we can shift a distribution forward one unit of time via postmultiplication by P.

Some distributions are invariant under this updating process —for example,

```
P = np.array([[0.4, 0.6],
              [0.2, 0.8]])
ψ = (0.25, 0.75)
ψ @ P
```

```
array([0.25, 0.75])
```

Notice that `ψ @ P` is the same as `ψ`.

Such distributions are called **stationary** or **invariant**.

Formally, a distribution ψ^* on S is called **stationary** for P if $\psi^* P = \psi^*$.

Notice that, postmultiplying by P, we have $\psi^* P^2 = \psi^* P = \psi^*$.

Continuing in the same way leads to $\psi^* = \psi^* P^t$ for all $t \geq 0$.

This tells us an important fact: If the distribution of ψ_0 is a stationary distribution, then ψ_t will have this same distribution for all $t \geq 0$.

The following theorem is proved in Chapter 4 of [Sargent and Stachurski, 2023] and numerous other sources.

Theorem 33.5.1

Every stochastic matrix P has at least one stationary distribution.

Note that there can be many stationary distributions corresponding to a given stochastic matrix P.

- For example, if P is the identity matrix, then all distributions on S are stationary.

To get uniqueness, we need the Markov chain to "mix around," so that the state doesn't get stuck in some part of the state space.

This gives some intuition for the following theorem.

Theorem 33.5.2

If P is everywhere positive, then P has exactly one stationary distribution.

We will come back to this when we introduce irreducibility in the *next lecture* on Markov chains.

33.5.1 Example

Recall our model of the employment/unemployment dynamics of a particular worker *discussed above*.

If $\alpha \in (0, 1)$ and $\beta \in (0, 1)$, then the transition matrix is everywhere positive.

Let $\psi^* = (p, 1 - p)$ be the stationary distribution, so that p corresponds to unemployment (state 0).

Using $\psi^* = \psi^* P$ and a bit of algebra yields

$$p = \frac{\beta}{\alpha + \beta}$$

This is, in some sense, a steady state probability of unemployment.

Not surprisingly it tends to zero as $\beta \to 0$, and to one as $\alpha \to 0$.

33.5.2 Calculating stationary distributions

A stable algorithm for computing stationary distributions is implemented in QuantEcon.py.

Here's an example

```
P = [[0.4, 0.6],
     [0.2, 0.8]]

mc = qe.MarkovChain(P)
mc.stationary_distributions  # Show all stationary distributions
```

```
array([[0.25, 0.75]])
```

33.5.3 Asymptotic stationarity

Consider an everywhere positive stochastic matrix with unique stationary distribution ψ^*.

Sometimes the distribution $\psi_t = \psi_0 P^t$ of X_t converges to ψ^* regardless of ψ_0.

For example, we have the following result

Theorem 33.5.3

If there exists an integer m such that all entries of P^m are strictly positive, then

$$\psi_0 P^t \to \psi^* \quad \text{as } t \to \infty$$

where ψ^* is the unique stationary distribution.

This situation is often referred to as **asymptotic stationarity** or **global stability**.

A proof of the theorem can be found in Chapter 4 of [Sargent and Stachurski, 2023], as well as many other sources.

Example: Hamilton's chain

Hamilton's chain satisfies the conditions of the theorem because P^2 is everywhere positive:

```
P = np.array([[0.971, 0.029, 0.000],
              [0.145, 0.778, 0.077],
              [0.000, 0.508, 0.492]])
P @ P
```

```
array([[0.947046, 0.050721, 0.002233],
       [0.253605, 0.648605, 0.09779 ],
       [0.07366 , 0.64516 , 0.28118 ]])
```

Let's pick an initial distribution ψ_1, ψ_2, ψ_3 and trace out the sequence of distributions $\psi_i P^t$ for $t = 0, 1, 2, ...$, for $i = 1, 2, 3$.

First, we write a function to iterate the sequence of distributions for `ts_length` period

```
def iterate_ψ(ψ_0, P, ts_length):
    n = len(P)
    ψ_t = np.empty((ts_length, n))
    ψ_t[0 ]= ψ_0
    for t in range(1, ts_length):
```

```
        ψ_t[t] = ψ_t[t-1] @ P
    return ψ_t
```

Now we plot the sequence

```
<IPython.core.display.HTML object>
```

Here

- P is the stochastic matrix for recession and growth *considered above*.

- The red, blue and green dots are initial marginal probability distributions ψ_1, ψ_2, ψ_3, each of which is represented as a vector in $\mathbb{R}^3$.

- The transparent dots are the marginal distributions $\psi_i P^t$ for $t = 1, 2, ...,$ for $i = 1, 2, 3..$

- The yellow dot is ψ^*.

You might like to try experimenting with different initial conditions.

Example: failure of convergence

Consider the periodic chain with stochastic matrix

$$P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

This matrix does not satisfy the conditions of strict_stationary because, as you can readily check,

- $P^m = P$ when m is odd and

- $P^m = I$, the identity matrix, when m is even.

Hence there is no m such that all elements of P^m are strictly positive.

Moreover, we can see that global stability does not hold.

For instance, if we start at $\psi_0 = (1, 0)$, then $\psi_m = \psi_0 P^m$ is $(1, 0)$ when m is even and $(0, 1)$ when m is odd.

We can see similar phenomena in higher dimensions.

The next figure illustrates this for a periodic Markov chain with three states.

```
<IPython.core.display.HTML object>
```

This animation demonstrates the behavior of an irreducible and periodic stochastic matrix.

The red, yellow, and green dots represent different initial probability distributions.

The blue dot represents the unique stationary distribution.

Unlike Hamilton's Markov chain, these initial distributions do not converge to the unique stationary distribution.

Instead, they cycle periodically around the probability simplex, illustrating that asymptotic stability fails.

33.6 Computing expectations

We sometimes want to compute mathematical expectations of functions of X_t of the form

$$\mathbb{E}[h(X_t)] \tag{33.7}$$

and conditional expectations such as

$$\mathbb{E}[h(X_{t+k}) \mid X_t = x] \tag{33.8}$$

where

- $\{X_t\}$ is a Markov chain generated by $n \times n$ stochastic matrix P.

- h is a given function, which, in terms of matrix algebra, we'll think of as the column vector

$$h = \begin{bmatrix} h(x_1) \\ \vdots \\ h(x_n) \end{bmatrix}.$$

Computing the unconditional expectation (**??**) is easy.

We just sum over the marginal distribution of X_t to get

$$\mathbb{E}[h(X_t)] = \sum_{x \in S} (\psi P^t)(x) h(x)$$

Here ψ is the distribution of X_0.

Since ψ and hence ψP^t are row vectors, we can also write this as

$$\mathbb{E}[h(X_t)] = \psi P^t h$$

For the conditional expectation (**??**), we need to sum over the conditional distribution of X_{t+k} given $X_t = x$.

We already know that this is $P^k(x, \cdot)$, so

$$\mathbb{E}[h(X_{t+k}) \mid X_t = x] = (P^k h)(x) \tag{33.9}$$

33.6.1 Expectations of geometric sums

Sometimes we want to compute the mathematical expectation of a geometric sum, such as $\sum_t \beta^t h(X_t)$.

In view of the preceding discussion, this is

$$\mathbb{E}\left[\sum_{j=0}^{\infty} \beta^j h(X_{t+j}) \mid X_t = x\right] = x + \beta(Ph)(x) + \beta^2(P^2h)(x) + \cdots$$

By the *Neumann series lemma*, this sum can be calculated using

$$I + \beta P + \beta^2 P^2 + \cdots = (I - \beta P)^{-1}$$

The vector $P^k h$ stores the conditional expectation $\mathbb{E}[h(X_{t+k}) \mid X_t = x]$ over all x.

Exercise 33.6.1

Imam and Temple [Imam and Temple, 2023] used a three-state transition matrix to describe the transition of three states of a regime: growth, stagnation, and collapse

$$P := \begin{bmatrix} 0.68 & 0.12 & 0.20 \\ 0.50 & 0.24 & 0.26 \\ 0.36 & 0.18 & 0.46 \end{bmatrix}$$

where rows, from top to down, correspond to growth, stagnation, and collapse.

In this exercise,

1. visualize the transition matrix and show this process is asymptotically stationary

2. calculate the stationary distribution using simulations

3. visualize the dynamics of $(\psi_0 P^t)(i)$ where $t \in 0, ..., 25$ and compare the convergent path with the previous transition matrix

Compare your solution to the paper.

Solution to Exercise 33.6.1

Solution 1:
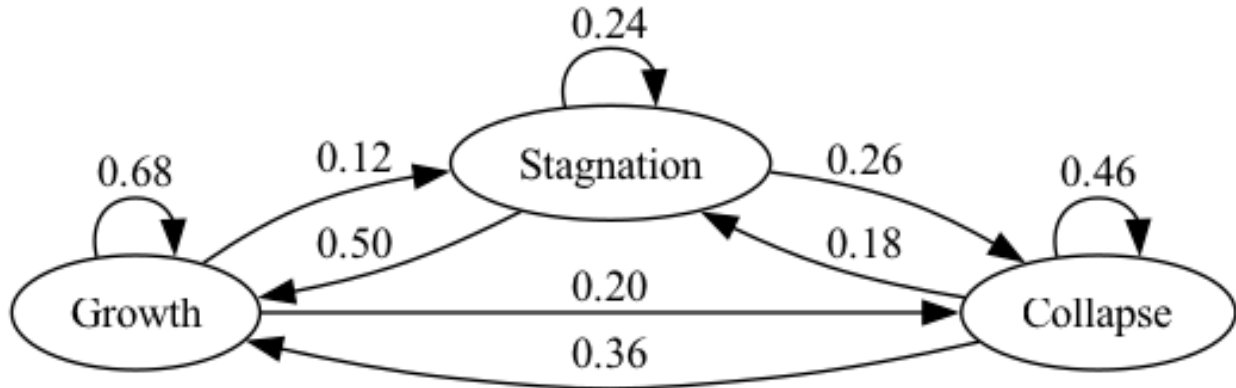
Since the matrix is everywhere positive, there is a unique stationary distribution ψ^* such that $\psi_t \to \psi^*$ as $t \to \infty$.

Solution 2:

One simple way to calculate the stationary distribution is to take the power of the transition matrix as we have shown before

```
P = np.array([[0.68, 0.12, 0.20],
              [0.50, 0.24, 0.26],
              [0.36, 0.18, 0.46]])
P_power = np.linalg.matrix_power(P, 20)
P_power
```

```
array([[0.56145769, 0.15565164, 0.28289067],
       [0.56145769, 0.15565164, 0.28289067],
       [0.56145769, 0.15565164, 0.28289067]])
```

Note that rows of the transition matrix converge to the stationary distribution.

```
ψ_star_p = P_power[0]
ψ_star_p
```

```
array([0.56145769, 0.15565164, 0.28289067])
```

```
mc = qe.MarkovChain(P)
ψ_star = mc.stationary_distributions[0]
ψ_star
```

```
array([0.56145769, 0.15565164, 0.28289067])
```

Exercise 33.6.2

We discussed the six-state transition matrix estimated by Imam & Temple [Imam and Temple, 2023] *before*.

```
nodes = ['DG', 'DC', 'NG', 'NC', 'AG', 'AC']
P = [[0.86, 0.11, 0.03, 0.00, 0.00, 0.00],
     [0.52, 0.33, 0.13, 0.02, 0.00, 0.00],
     [0.12, 0.03, 0.70, 0.11, 0.03, 0.01],
     [0.13, 0.02, 0.35, 0.36, 0.10, 0.04],
     [0.00, 0.00, 0.09, 0.11, 0.55, 0.25],
     [0.00, 0.00, 0.09, 0.15, 0.26, 0.50]]
```

In this exercise,

1. show this process is asymptotically stationary without simulation

2. simulate and visualize the dynamics starting with a uniform distribution across states (each state will have a probability of 1/6)

3. change the initial distribution to P(DG) = 1, while all other states have a probability of 0

Solution to Exercise 33.6.2

Solution 1:

Although P is not every positive, P^m when $m = 3$ is everywhere positive.

```
P = np.array([[0.86, 0.11, 0.03, 0.00, 0.00, 0.00],
              [0.52, 0.33, 0.13, 0.02, 0.00, 0.00],
              [0.12, 0.03, 0.70, 0.11, 0.03, 0.01],
              [0.13, 0.02, 0.35, 0.36, 0.10, 0.04],
              [0.00, 0.00, 0.09, 0.11, 0.55, 0.25],
              [0.00, 0.00, 0.09, 0.15, 0.26, 0.50]])

np.linalg.matrix_power(P,3)
```

```
array([[0.764927, 0.133481, 0.085949, 0.011481, 0.002956, 0.001206],
       [0.658861, 0.131559, 0.161367, 0.031703, 0.011296, 0.005214],
       [0.291394, 0.057788, 0.439702, 0.113408, 0.062707, 0.035001],
       [0.272459, 0.051361, 0.365075, 0.132207, 0.108152, 0.070746],
       [0.064129, 0.012533, 0.232875, 0.154385, 0.299243, 0.236835],
       [0.072865, 0.014081, 0.244139, 0.160905, 0.265846, 0.242164]])
```

So it satisfies the requirement.

Solution 2:

We find the distribution ψ converges to the stationary distribution quickly regardless of the initial distributions

```python
ts_length = 30
num_distributions = 20
nodes = ['DG', 'DC', 'NG', 'NC', 'AG', 'AC']

# Get parameters of transition matrix
n = len(P)
mc = qe.MarkovChain(P)
ψ_star = mc.stationary_distributions[0]
ψ_0 = np.array([[1/6 for i in range(6)],
                [0 if i != 0 else 1 for i in range(6)]])
## Draw the plot
fig, axes = plt.subplots(ncols=2)
plt.subplots_adjust(wspace=0.35)
for idx in range(2):
    ψ_t = iterate_ψ(ψ_0[idx], P, ts_length)
    for i in range(n):
        axes[idx].plot(ψ_t[:, i] - ψ_star[i], alpha=0.5, label=fr'$\psi_t({i+1})$')
        axes[idx].set_ylim([-0.3, 0.3])
        axes[idx].set_xlabel('t')
        axes[idx].set_ylabel(fr'$\psi_t$')
        axes[idx].legend()
        axes[idx].axhline(0, linestyle='dashed', lw=1, color = 'black')

plt.show()
```
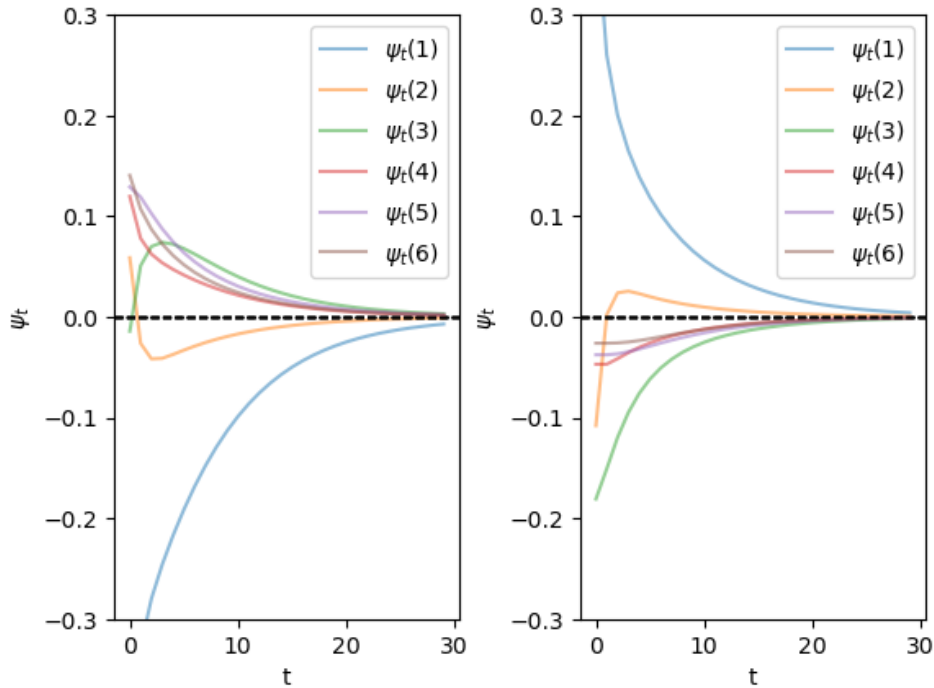
Exercise 33.6.3

Prove the following: If P is a stochastic matrix, then so is the k-th power P^k for all $k \in \mathbb{N}$.

Solution to Exercise 33.6.3

Suppose that P is stochastic and, moreover, that P^k is stochastic for some integer k.

We will prove that $P^{k+1} = PP^k$ is also stochastic.

(We are doing proof by induction —we assume the claim is true at k and now prove it is true at $k + 1$.)

To see this, observe that, since P^k is stochastic and the product of nonnegative matrices is nonnegative, $P^{k+1} = PP^k$ is nonnegative.

Also, if $\mathbf{1}$ is a column vector of ones, then, since P^k is stochastic we have $P^k \mathbf{1} = \mathbf{1}$ (rows sum to one).

Therefore $P^{k+1} \mathbf{1} = PP^k \mathbf{1} = P\mathbf{1} = \mathbf{1}$

The proof is done.

MARKOV CHAINS: IRREDUCIBILITY AND ERGODICITY

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

```
!pip install quantecon
```

34.1 Overview

This lecture continues on from our *earlier lecture on Markov chains*.

Specifically, we will introduce the concepts of irreducibility and ergodicity, and see how they connect to stationarity.

Irreducibility describes the ability of a Markov chain to move between any two states in the system.

Ergodicity is a sample path property that describes the behavior of the system over long periods of time.

As we will see,

- an irreducible Markov chain guarantees the existence of a unique stationary distribution, while

- an ergodic Markov chain generates time series that satisfy a version of the law of large numbers.

Together, these concepts provide a foundation for understanding the long-term behavior of Markov chains.

Let's start with some standard imports:

```python
import matplotlib.pyplot as plt
import quantecon as qe
import numpy as np
```

```
OMP: Info #276: omp_set_nested routine deprecated, please use omp_set_max_active_
↪levels instead.
```

34.2 Irreducibility

To explain irreducibility, let's take P to be a fixed stochastic matrix.

State x is called **accessible** (or **reachable**) from state y if $P^t(x, y) > 0$ for some integer $t \geq 0$.

Two states, x and y, are said to **communicate** if x and y are accessible from each other.

In view of our discussion *above*, this means precisely that

- state x can eventually be reached from state y, and

- state y can eventually be reached from state x

The stochastic matrix P is called **irreducible** if all states communicate; that is, if x and y communicate for all (x, y) in $S \times S$.

Example 34.2.1

For example, consider the following transition probabilities for wealth of a fictitious set of households
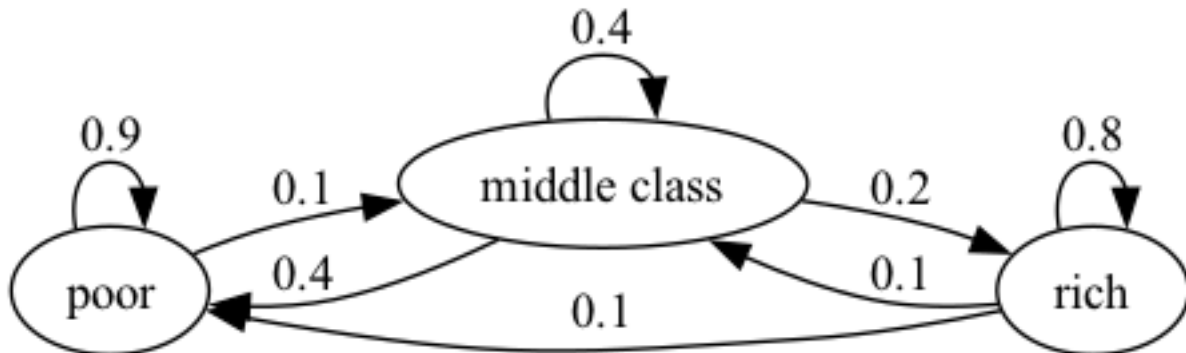
We can translate this into a stochastic matrix, putting zeros where there's no edge between nodes

$$P := \begin{bmatrix} 0.9 & 0.1 & 0 \\ 0.4 & 0.4 & 0.2 \\ 0.1 & 0.1 & 0.8 \end{bmatrix}$$

It's clear from the graph that this stochastic matrix is irreducible: we can eventually reach any state from any other state.

```
We can also test this using [QuantEcon.py](http://quantecon.org/quantecon-py)'s␣
␣MarkovChain class

```{code-cell} ipython3
```

```
P = [[0.9, 0.1, 0.0],
     [0.4, 0.4, 0.2],
     [0.1, 0.1, 0.8]]

mc = qe.MarkovChain(P, ('poor', 'middle', 'rich'))
mc.is_irreducible
```

Example 34.2.2

Here's a more pessimistic scenario in which poor people remain poor forever
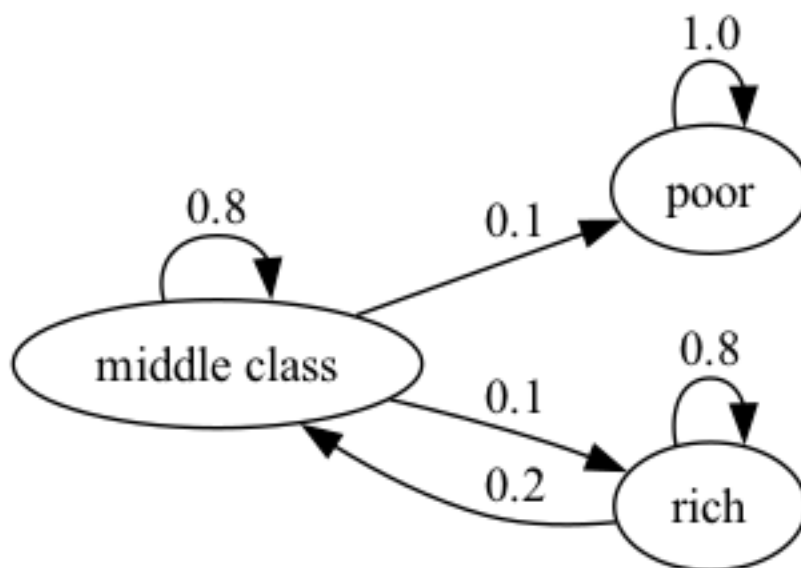
This stochastic matrix is not irreducible since, for example, rich is not accessible from poor.

```
Let's confirm this

```{code-cell} ipython3
P = [[1.0, 0.0, 0.0],
     [0.1, 0.8, 0.1],
     [0.0, 0.2, 0.8]]

mc = qe.MarkovChain(P, ('poor', 'middle', 'rich'))
mc.is_irreducible
```

It might be clear to you already that irreducibility is going to be important in terms of long-run outcomes.

For example, poverty is a life sentence in the second graph but not the first.

We'll come back to this a bit later.

34.2.1 Irreducibility and stationarity

We discussed uniqueness of stationary distributions in our earlier lecture *Markov Chains: Basic Concepts*.

There we *stated* that uniqueness holds when the transition matrix is everywhere positive.

In fact irreducibility is sufficient:

Theorem 34.2.1

If P is irreducible, then P has exactly one stationary distribution.

For proof, see Chapter 4 of [Sargent and Stachurski, 2023] or Theorem 5.2 of [Häggström, 2002].

34.3 Ergodicity

Under irreducibility, yet another important result obtains:

Theorem 34.3.1

If P is irreducible and ψ^* is the unique stationary distribution, then, for all $x \in S$,

$$\frac{1}{m} \sum_{t=1}^{m} \mathbb{1}\{X_t = x\} \to \psi^*(x) \quad \text{as } m \to \infty \tag{34.1}$$

Here

- $\{X_t\}$ is a Markov chain with stochastic matrix P and initial distribution ψ_0

- $\mathbb{1}\{X_t = x\} = 1$ if $X_t = x$ and zero otherwise.

The result in *(34.1)* is sometimes called **ergodicity**.

The theorem tells us that the fraction of time the chain spends at state x converges to $\psi^*(x)$ as time goes to infinity.

This gives us another way to interpret the stationary distribution (provided irreducibility holds).

Importantly, the result is valid for any choice of ψ_0.

The theorem is related to *the law of large numbers*.

It tells us that, in some settings, the law of large numbers sometimes holds even when the sequence of random variables is *not IID*.

34.3.1 Example: ergodicity and unemployment

Recall our cross-sectional interpretation of the employment/unemployment model *discussed before*.

Assume that $\alpha \in (0, 1)$ and $\beta \in (0, 1)$, so that irreducibility holds.

We saw that the stationary distribution is $(p, 1 - p)$, where

$$p = \frac{\beta}{\alpha + \beta}$$

In the cross-sectional interpretation, this is the fraction of people unemployed.

In view of our latest (ergodicity) result, it is also the fraction of time that a single worker can expect to spend unemployed.

Thus, in the long run, cross-sectional averages for a population and time-series averages for a given person coincide.

This is one aspect of the concept of ergodicity.

34.3.2 Example: Hamilton dynamics

Another example is the Hamilton dynamics we *discussed before*.

Let $\{X_t\}$ be a sample path generated by these dynamics.

Let's denote the fraction of time spent in state x over the period $t = 1, \dots, n$ by $\hat{p}_n(x)$, so that

$$\hat{p}_n(x) := \frac{1}{n} \sum_{t=1}^{n} \mathbb{1}\{X_t = x\} \qquad (x \in \{0, 1, 2\})$$

The *graph* of the Markov chain shows it is irreducible, so ergodicity holds.

Hence we expect that $\hat{p}_n(x) \approx \psi^*(x)$ when n is large.

The next figure shows convergence of $\hat{p}_n(x)$ to $\psi^*(x)$ when $x = 1$ and X_0 is either $0, 1$ or 2.

```
P = np.array([[0.971, 0.029, 0.000],
              [0.145, 0.778, 0.077],
              [0.000, 0.508, 0.492]])
ts_length = 10_000
mc = qe.MarkovChain(P)
ψ_star = mc.stationary_distributions[0]
x = 1   # We study convergence to psi^*(x)

fig, ax = plt.subplots()
```

(continues on next page)

```
ax.axhline(ψ_star[x], linestyle='dashed', color='black',
                    label = fr'$\psi^*({x})$')
# Compute the fraction of time spent in state 0, starting from different x_0s
for x0 in range(len(P)):
    X = mc.simulate(ts_length, init=x0)
    p_hat = (X == x).cumsum() / np.arange(1, ts_length+1)
    ax.plot(p_hat, label=fr'$\hat p_n({x})$ when $X_0 = \, {x0}$')
ax.set_xlabel('t')
ax.set_ylabel(fr'$\hat p_n({x})$')
ax.legend()
plt.show()
```
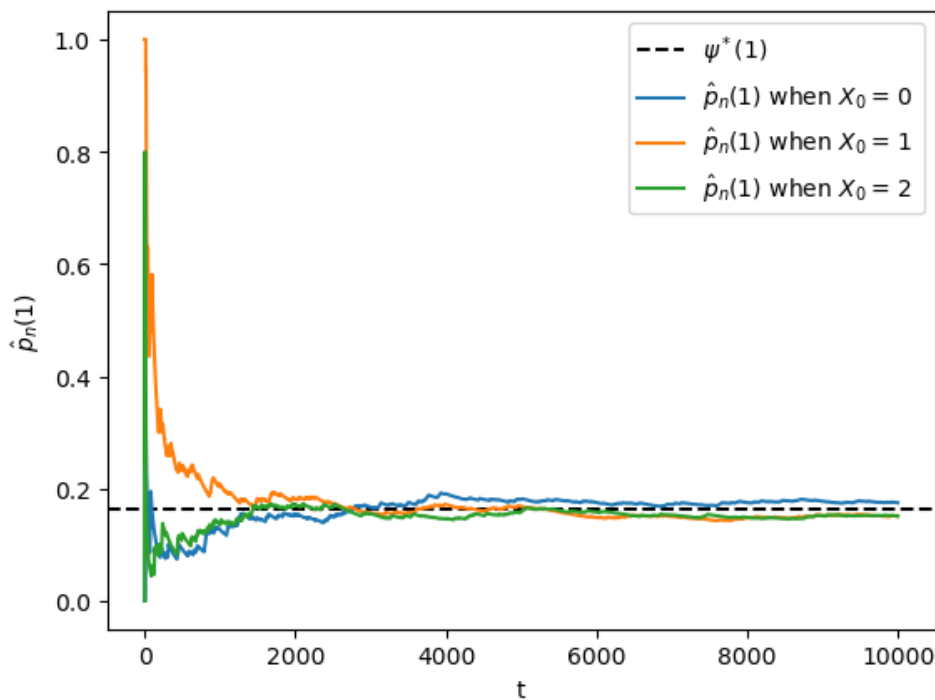

You might like to try changing $x = 1$ to either $x = 0$ or $x = 2$.

In any of these cases, ergodicity will hold.

34.3.3 Example: a periodic chain

Example 34.3.1

Let's look at the following example with states 0 and 1:

$$P := \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

The transition graph shows that this model is irreducible.
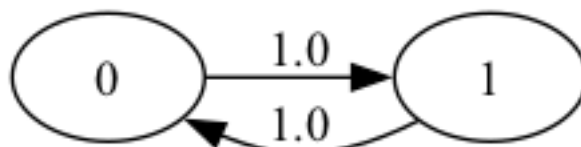
Notice that there is a periodic cycle —the state cycles between the two states in a regular way.

```
Not surprisingly, this property
is called [periodicity](https://stats.libretexts.org/Bookshelves/Probability_Theory/
 ↪Probability_Mathematical_Statistics_and_Stochastic_Processes_(Siegrist)/16%3A_
 ↪Markov_Processes/16.05%3A_Periodicity_of_Discrete-Time_Chains).


Nonetheless, the model is irreducible, so ergodicity holds.


The following figure illustrates

```{code-cell} ipython3
P = np.array([[0, 1],
              [1, 0]])
ts_length = 100
mc = qe.MarkovChain(P)
n = len(P)
fig, axes = plt.subplots(nrows=1, ncols=n)
ψ_star = mc.stationary_distributions[0]

for i in range(n):
    axes[i].axhline(ψ_star[i], linestyle='dashed', lw=2, color='black',
                    label = fr'$\psi^*({i})$')
    axes[i].set_xlabel('t')
    axes[i].set_ylabel(fr'$\hat p_n({i})$')

    # Compute the fraction of time spent, for each x
```

```
    for x0 in range(n):
        # Generate time series starting at different x_0
        X = mc.simulate(ts_length, init=x0)
        p_hat = (X == i).cumsum() / np.arange(1, ts_length+1)
        axes[i].plot(p_hat, label=f'$x_0 = \, {x0} $')


    axes[i].legend()
plt.tight_layout()
plt.show()
```

This example helps to emphasize that asymptotic stationarity is about the distribution, while ergodicity is about the sample path.

The proportion of time spent in a state can converge to the stationary distribution with periodic chains.

However, the distribution at each state does not.

34.3.4 Example: political institutions

Let's go back to the political institutions model with six states discussed *in a previous lecture* and study ergodicity.

Here's the transition matrix.

$$P := \begin{bmatrix} 0.86 & 0.11 & 0.03 & 0.00 & 0.00 & 0.00 \\ 0.52 & 0.33 & 0.13 & 0.02 & 0.00 & 0.00 \\ 0.12 & 0.03 & 0.70 & 0.11 & 0.03 & 0.01 \\ 0.13 & 0.02 & 0.35 & 0.36 & 0.10 & 0.04 \\ 0.00 & 0.00 & 0.09 & 0.11 & 0.55 & 0.25 \\ 0.00 & 0.00 & 0.09 & 0.15 & 0.26 & 0.50 \end{bmatrix}$$

The *graph* for the chain shows all states are reachable, indicating that this chain is irreducible.

In the next figure, we visualize the difference $\hat{p}_n(x) - \psi^*(x)$ for each state x.

Unlike the previous figure, X_0 is held fixed.

```
P = [[0.86, 0.11, 0.03, 0.00, 0.00, 0.00],
     [0.52, 0.33, 0.13, 0.02, 0.00, 0.00],
     [0.12, 0.03, 0.70, 0.11, 0.03, 0.01],
     [0.13, 0.02, 0.35, 0.36, 0.10, 0.04],
     [0.00, 0.00, 0.09, 0.11, 0.55, 0.25],
     [0.00, 0.00, 0.09, 0.15, 0.26, 0.50]]

ts_length = 2500
```

```python
mc = qe.MarkovChain(P)
ψ_star = mc.stationary_distributions[0]
fig, ax = plt.subplots()
X = mc.simulate(ts_length, random_state=1)
# Center the plot at 0
ax.axhline(linestyle='dashed', lw=2, color='black')


for x0 in range(len(P)):
    # Calculate the fraction of time for each state
    p_hat = (X == x0).cumsum() / np.arange(1, ts_length+1)
    ax.plot(p_hat - ψ_star[x0], label=f'$x = {x0+1} $')
    ax.set_xlabel('t')
    ax.set_ylabel(r'$\hat p_n(x) - \psi^* (x)$')

ax.legend()
plt.show()
```
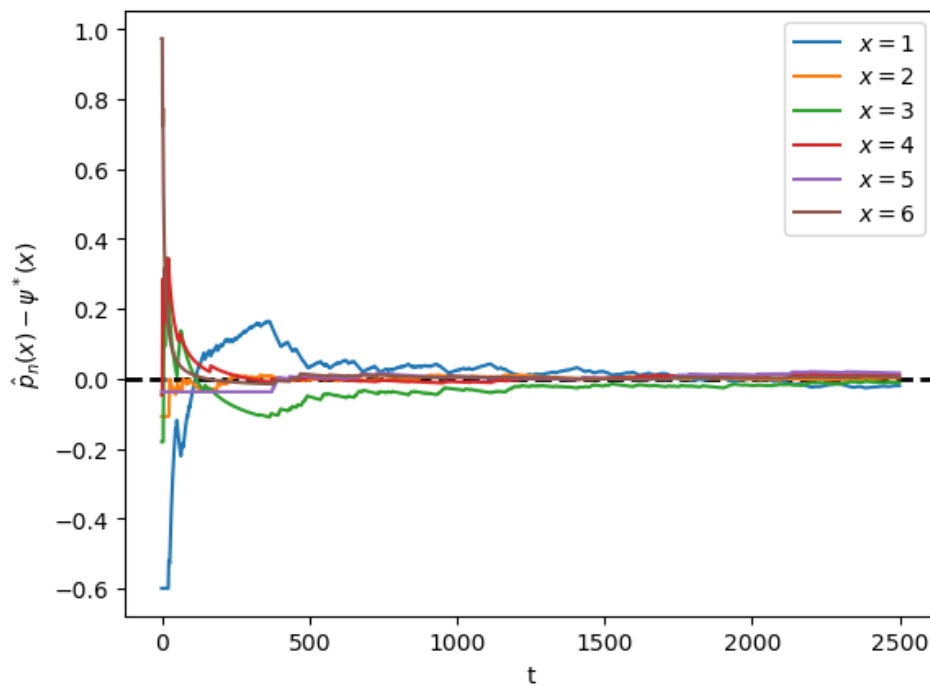
34.4 Exercises

Exercise 34.4.1

Benhabib et al. [Benhabib *et al.*, 2019] estimated that the transition matrix for social mobility as the following

$$
P := \begin{bmatrix}
0.222 & 0.222 & 0.215 & 0.187 & 0.081 & 0.038 & 0.029 & 0.006 \\
0.221 & 0.22 & 0.215 & 0.188 & 0.082 & 0.039 & 0.029 & 0.006 \\
0.207 & 0.209 & 0.21 & 0.194 & 0.09 & 0.046 & 0.036 & 0.008 \\
0.198 & 0.201 & 0.207 & 0.198 & 0.095 & 0.052 & 0.04 & 0.009 \\
0.175 & 0.178 & 0.197 & 0.207 & 0.11 & 0.067 & 0.054 & 0.012 \\
0.182 & 0.184 & 0.2 & 0.205 & 0.106 & 0.062 & 0.05 & 0.011 \\
0.123 & 0.125 & 0.166 & 0.216 & 0.141 & 0.114 & 0.094 & 0.021 \\
0.084 & 0.084 & 0.142 & 0.228 & 0.17 & 0.143 & 0.121 & 0.028
\end{bmatrix}
$$

where each state 1 to 8 corresponds to a percentile of wealth shares

$$0 - 20\%, 20 - 40\%, 40 - 60\%, 60 - 80\%, 80 - 90\%, 90 - 95\%, 95 - 99\%, 99 - 100\%$$

The matrix is recorded as `P` below

```
P = [
    [0.222, 0.222, 0.215, 0.187, 0.081, 0.038, 0.029, 0.006],
    [0.221, 0.22,  0.215, 0.188, 0.082, 0.039, 0.029, 0.006],
    [0.207, 0.209, 0.21,  0.194, 0.09,  0.046, 0.036, 0.008],
    [0.198, 0.201, 0.207, 0.198, 0.095, 0.052, 0.04,  0.009],
    [0.175, 0.178, 0.197, 0.207, 0.11,  0.067, 0.054, 0.012],
    [0.182, 0.184, 0.2,   0.205, 0.106, 0.062, 0.05,  0.011],
    [0.123, 0.125, 0.166, 0.216, 0.141, 0.114, 0.094, 0.021],
    [0.084, 0.084, 0.142, 0.228, 0.17,  0.143, 0.121, 0.028]
    ]

P = np.array(P)
codes_B = ('1','2','3','4','5','6','7','8')
```

1. Show this process is asymptotically stationary and calculate an approximation to the stationary distribution.

2. Use simulations to illustrate ergodicity.

Solution to Exercise 34.4.1

Part 1:

One option is to take the power of the transition matrix.

```
P = [[0.222, 0.222, 0.215, 0.187, 0.081, 0.038, 0.029, 0.006],
     [0.221, 0.22,  0.215, 0.188, 0.082, 0.039, 0.029, 0.006],
     [0.207, 0.209, 0.21,  0.194, 0.09,  0.046, 0.036, 0.008],
     [0.198, 0.201, 0.207, 0.198, 0.095, 0.052, 0.04,  0.009],
     [0.175, 0.178, 0.197, 0.207, 0.11,  0.067, 0.054, 0.012],
     [0.182, 0.184, 0.2,   0.205, 0.106, 0.062, 0.05,  0.011],
     [0.123, 0.125, 0.166, 0.216, 0.141, 0.114, 0.094, 0.021],
     [0.084, 0.084, 0.142, 0.228, 0.17,  0.143, 0.121, 0.028]]

P = np.array(P)
codes_B = ('1','2','3','4','5','6','7','8')

np.linalg.matrix_power(P, 10)
```

```
array([[0.20254451, 0.20379879, 0.20742102, 0.19505842, 0.09287832,
        0.0503871 , 0.03932382, 0.00858802],
       [0.20254451, 0.20379879, 0.20742102, 0.19505842, 0.09287832,
        0.0503871 , 0.03932382, 0.00858802],
       [0.20254451, 0.20379879, 0.20742102, 0.19505842, 0.09287832,
        0.0503871 , 0.03932382, 0.00858802],
       [0.20254451, 0.20379879, 0.20742102, 0.19505842, 0.09287832,
        0.0503871 , 0.03932382, 0.00858802],
       [0.20254451, 0.20379879, 0.20742102, 0.19505842, 0.09287832,
        0.0503871 , 0.03932382, 0.00858802],
       [0.20254451, 0.20379879, 0.20742102, 0.19505842, 0.09287832,
        0.0503871 , 0.03932382, 0.00858802],
       [0.20254451, 0.20379879, 0.20742102, 0.19505842, 0.09287832,
        0.0503871 , 0.03932382, 0.00858802],
       [0.20254451, 0.20379879, 0.20742102, 0.19505842, 0.09287832,
        0.0503871 , 0.03932382, 0.00858802]])
```

For this model, rows of P^n converge to the stationary distribution as $n \to \infty$:

```
mc = qe.MarkovChain(P)
ψ_star = mc.stationary_distributions[0]
ψ_star
```

```
array([0.20254451, 0.20379879, 0.20742102, 0.19505842, 0.09287832,
       0.0503871 , 0.03932382, 0.00858802])
```

Part 2:

```
ts_length = 1000
mc = qe.MarkovChain(P)
fig, ax = plt.subplots()
X = mc.simulate(ts_length, random_state=1)
ax.axhline(linestyle='dashed', lw=2, color='black')

for x0 in range(len(P)):
    # Calculate the fraction of time for each worker
    p_hat = (X == x0).cumsum() / np.arange(1, ts_length+1)
    ax.plot(p_hat - ψ_star[x0], label=f'$x = {x0+1} $')
    ax.set_xlabel('t')
    ax.set_ylabel(r'$\hat p_n(x) - \psi^* (x)$')

ax.legend()
plt.show()
```
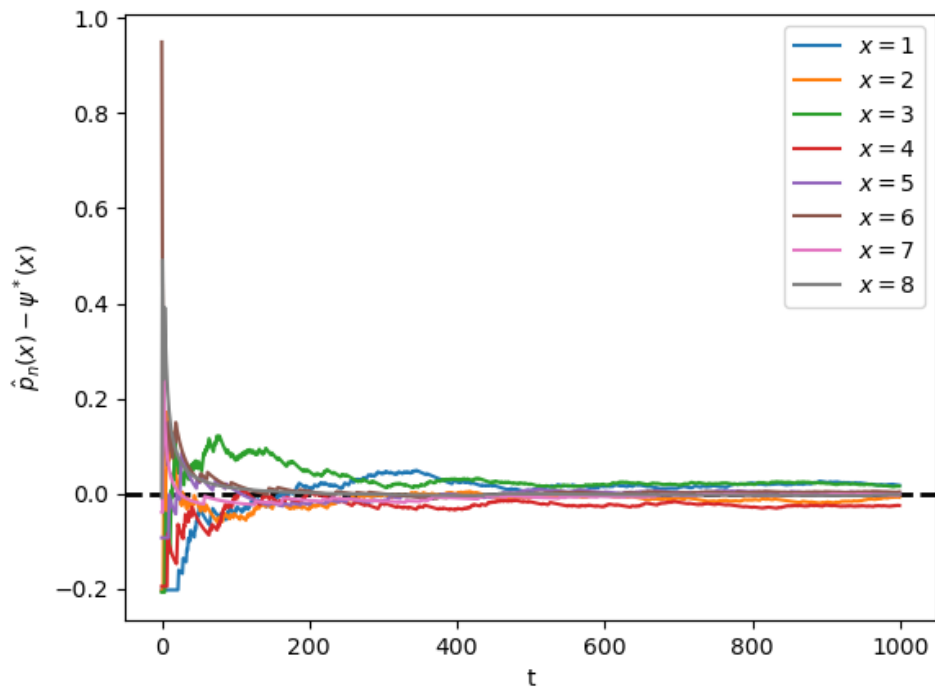

Note that the fraction of time spent at each state converges to the probability assigned to that state by the stationary distribution.

Exercise 34.4.2

According to the discussion *above*, if a worker's employment dynamics obey the stochastic matrix

$$P := \begin{bmatrix} 1 - \alpha & \alpha \\ \beta & 1 - \beta \end{bmatrix}$$

with $\alpha \in (0, 1)$ and $\beta \in (0, 1)$, then, in the long run, the fraction of time spent unemployed will be

$$p := \frac{\beta}{\alpha + \beta}$$

In other words, if $\{X_t\}$ represents the Markov chain for employment, then $\bar{X}_m \to p$ as $m \to \infty$, where

$$\bar{X}_m := \frac{1}{m} \sum_{t=1}^{m} \mathbb{1}\{X_t = 0\}$$

This exercise asks you to illustrate convergence by computing $\bar{X}_m$ for large m and checking that it is close to p.

You will see that this statement is true regardless of the choice of initial condition or the values of α, β, provided both lie in $(0, 1)$.

The result should be similar to the plot we plotted *here*

Solution to Exercise 34.4.2

We will address this exercise graphically.

The plots show the time series of $\bar{X}_m - p$ for two initial conditions.

As m gets large, both series converge to zero.

```
α = β = 0.1
ts_length = 3000
p = β / (α + β)

P = ((1 - α,       α),              # Careful: P and p are distinct
     (    β,   1 - β))
mc = qe.MarkovChain(P)

fig, ax = plt.subplots()
ax.axhline(linestyle='dashed', lw=2, color='black')

for x0 in range(len(P)):
    # Generate time series for worker that starts at x0
    X = mc.simulate(ts_length, init=x0)
    # Compute fraction of time spent unemployed, for each n
    X_bar = (X == 0).cumsum() / np.arange(1, ts_length+1)
    # Plot
```

```
    ax.plot(X_bar - p, label=f'$x_0 = \, {x0} $')
    ax.set_xlabel('t')
    ax.set_ylabel(r'$\bar X_m - \psi^* (x)$')

ax.legend()
plt.show()
```
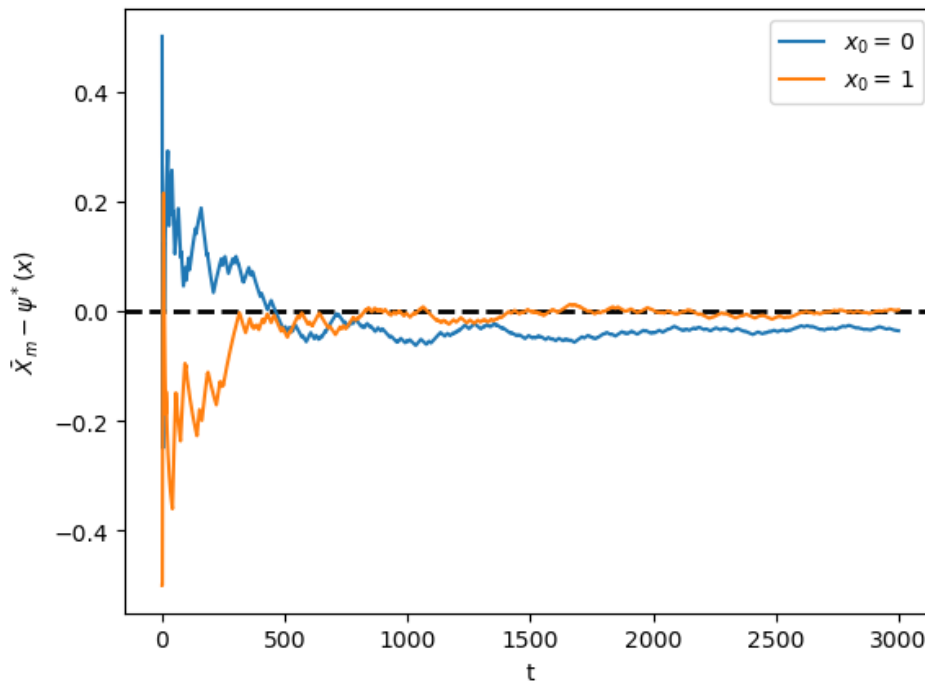
Exercise 34.4.3

In `quantecon` library, irreducibility is tested by checking whether the chain forms a strongly connected component.

Another way to test irreducibility is via the following statement:

The $n \times n$ matrix A is irreducible if and only if $\sum_{k=0}^{n-1} A^k$ is a strictly positive matrix.

(see, e.g., [Zhao, 2012] and this StackExchange post)

Based on this claim, write a function to test irreducibility.

Solution to Exercise 34.4.3

```python
def is_irreducible(P):
    n = len(P)
    result = np.zeros((n, n))
    for i in range(n):
        result += np.linalg.matrix_power(P, i)
    return np.all(result > 0)
```

Let's try it.

```python
P1 = np.array([[0, 1],
               [1, 0]])
P2 = np.array([[1.0, 0.0, 0.0],
               [0.1, 0.8, 0.1],
               [0.0, 0.2, 0.8]])
P3 = np.array([[0.971, 0.029, 0.000],
               [0.145, 0.778, 0.077],
               [0.000, 0.508, 0.492]])

for P in (P1, P2, P3):
    result = lambda P: 'irreducible' if is_irreducible(P) else 'reducible'
    print(f'{P}: {result(P)}')
```

```
[[0 1]
 [1 0]]: irreducible
[[1.  0.  0. ]
 [0.1 0.8 0.1]
 [0.  0.2 0.8]]: reducible
[[0.971 0.029 0.   ]
 [0.145 0.778 0.077]
 [0.    0.508 0.492]]: irreducible
```

UNIVARIATE TIME SERIES WITH MATRIX ALGEBRA

35.1 Overview

This lecture uses matrices to solve some linear difference equations.

As a running example, we'll study a **second-order linear difference equation** that was the key technical tool in Paul Samuelson's 1939 article [Samuelson, 1939] that introduced the *multiplier-accelerator model*.

This model became the workhorse that powered early econometric versions of Keynesian macroeconomic models in the United States.

You can read about the details of that model in Samuelson Multiplier-Accelerator.

(That lecture also describes some technicalities about second-order linear difference equations.)

In this lecture, we'll also learn about an **autoregressive** representation and a **moving average** representation of a non-stationary univariate time series $\{y_t\}_{t=0}^{T}$.

We'll also study a "perfect foresight" model of stock prices that involves solving a "forward-looking" linear difference equation.

We will use the following imports:

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm

# Custom figsize for this lecture
plt.rcParams["figure.figsize"] = (11, 5)

# Set decimal printing to 3 decimal places
np.set_printoptions(precision=3, suppress=True)
```

35.2 Samuelson's model

Let $t = 0, \pm 1, \pm 2, \dots$ index time.

For $t = 1, 2, 3, \dots, T$ suppose that

$$y_t = \alpha_0 + \alpha_1 y_{t-1} + \alpha_2 y_{t-2} \tag{35.1}$$

where we assume that y_0 and y_{-1} are given numbers that we take as *initial conditions*.

In Samuelson's model, y_t stood for **national income** or perhaps a different measure of aggregate activity called **gross domestic product** (GDP) at time t.

Equation (**??**) is called a *second-order linear difference equation*. It is called second order because it depends on two lags.

But actually, it is a collection of T simultaneous linear equations in the T variables $y_1, y_2, \dots, y_T$.

Note: To be able to solve a second-order linear difference equation, we require two *boundary conditions* that can take the form either of two *initial conditions*, two *terminal conditions* or possibly one of each.

Let's write our equations as a stacked system

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -\alpha_1 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -\alpha_2 & -\alpha_1 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & -\alpha_2 & -\alpha_1 & 1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & -\alpha_2 & -\alpha_1 & 1 \end{bmatrix}}_{\equiv A} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_T \end{bmatrix} = \underbrace{\begin{bmatrix} \alpha_0 + \alpha_1 y_0 + \alpha_2 y_{-1} \\ \alpha_0 + \alpha_2 y_0 \\ \alpha_0 \\ \alpha_0 \\ \vdots \\ \alpha_0 \end{bmatrix}}_{\equiv b}$$

or

$$Ay = b$$

where

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_T \end{bmatrix}$$

Evidently y can be computed from

$$y = A^{-1}b$$

The vector y is a complete time path $\{y_t\}_{t=1}^T$.

Let's put Python to work on an example that captures the flavor of Samuelson's multiplier-accelerator model.

We'll set parameters equal to the same values we used in Samuelson Multiplier-Accelerator.

```
T = 80

# parameters
α_0 = 10.0
α_1 = 1.53
α_2 = -.9

y_neg1 = 28.0 # y_{-1}
y_0 = 24.0
```

Now we construct A and b.

```
A = np.identity(T)   # The T x T identity matrix

for i in range(T):

    if i-1 >= 0:
        A[i, i-1] = -α_1

    if i-2 >= 0:
        A[i, i-2] = -α_2

b = np.full(T, α_0)
b[0] = α_0 + α_1 * y_0 + α_2 * y_neg1
b[1] = α_0 + α_2 * y_0
```

Let's look at the matrix A and the vector b for our example.

```
A, b
```

```
(array([[ 1.  ,  0.  ,  0.  , ...,  0.  ,  0.  ,  0.  ],
        [-1.53,  1.  ,  0.  , ...,  0.  ,  0.  ,  0.  ],
        [ 0.9 , -1.53,  1.  , ...,  0.  ,  0.  ,  0.  ],
        ...,
        [ 0.  ,  0.  ,  0.  , ...,  1.  ,  0.  ,  0.  ],
        [ 0.  ,  0.  ,  0.  , ..., -1.53,  1.  ,  0.  ],
        [ 0.  ,  0.  ,  0.  , ...,  0.9 , -1.53,  1.  ]]),
 array([ 21.52, -11.6 ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,
         10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,
         10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,
         10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,
         10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,
         10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,
```

```
        10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,
        10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,
        10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,
        10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ,  10.  ]))
```

Now let's solve for the path of y.

If y_t is GNP at time t, then we have a version of Samuelson's model of the dynamics for GNP.

To solve $y = A^{-1}b$ we can either invert A directly, as in

```
A_inv = np.linalg.inv(A)

y = A_inv @ b
```

or we can use `np.linalg.solve`:

```
y_second_method = np.linalg.solve(A, b)
```

Here make sure the two methods give the same result, at least up to floating point precision:

```
np.allclose(y, y_second_method)
```

```
True
```

A is invertible as it is lower triangular and its diagonal entries are non-zero

```
# Check if A is lower triangular
np.allclose(A, np.tril(A))
```

```
True
```

Note: In general, `np.linalg.solve` is more numerically stable than using `np.linalg.inv` directly. However, stability is not an issue for this small example. Moreover, we will repeatedly use `A_inv` in what follows, so there is added value in computing it directly.

Now we can plot.

```
plt.plot(np.arange(T)+1, y)
plt.xlabel('t')
```

```
plt.ylabel('y')

plt.show()
```
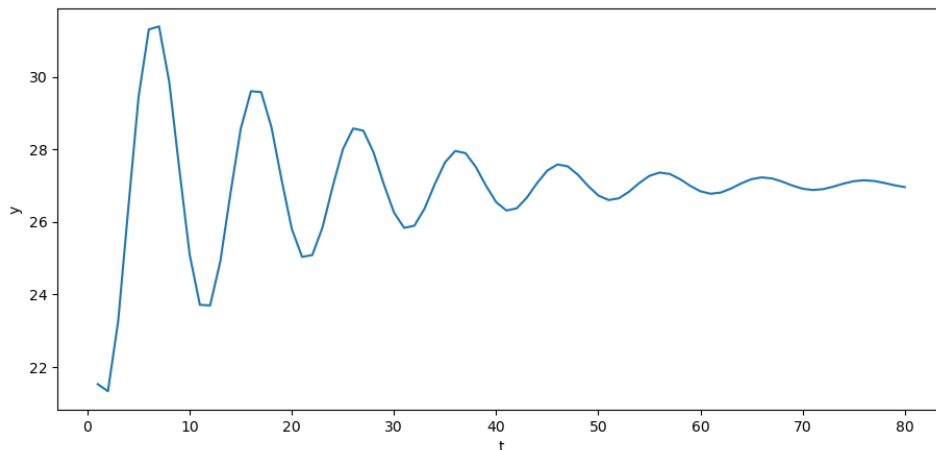

The *steady state* value y^* of y_t is obtained by setting $y_t = y_{t-1} = y_{t-2} = y^*$ in (**??**), which yields

$$y^* = \frac{\alpha_0}{1 - \alpha_1 - \alpha_2}$$

If we set the initial values to $y_0 = y_{-1} = y^*$, then y_t will be constant:

```
y_star = α_0 / (1 - α_1 - α_2)
y_neg1_steady = y_star # y_{-1}
y_0_steady = y_star

b_steady = np.full(T, α_0)
b_steady[0] = α_0 + α_1 * y_0_steady + α_2 * y_neg1_steady
b_steady[1] = α_0 + α_2 * y_0_steady
```

```
y_steady = A_inv @ b_steady
```

```
plt.plot(np.arange(T)+1, y_steady)
plt.xlabel('t')
plt.ylabel('y')

plt.show()
```
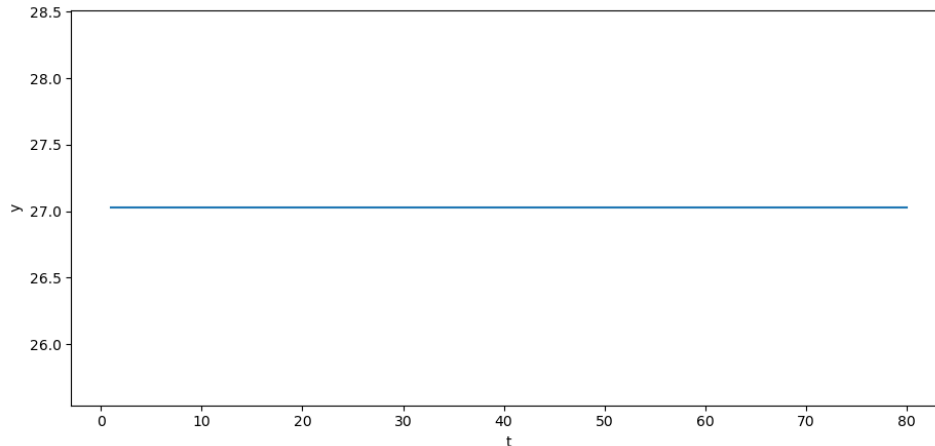
35.3 Adding a random term

To generate some excitement, we'll follow in the spirit of the great economists [Eugen Slutsky](#) and [Ragnar Frisch](#) and replace our original second-order difference equation with the following **second-order stochastic linear difference equation**:

$$y_t = \alpha_0 + \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + u_t \tag{35.2}$$

where $u_t \sim N\left(0, \sigma_u^2\right)$ and is *IID*, meaning independent and identically distributed.

We'll stack these T equations into a system cast in terms of matrix algebra.

Let's define the random vector

$$u = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_T \end{bmatrix}$$

Where A, b, y are defined as above, now assume that y is governed by the system

$$Ay = b + u \tag{35.3}$$

The solution for y becomes

$$y = A^{-1}\left(b + u\right) \tag{35.4}$$

Let's try it out in Python.

```
σ_u = 2.
u = np.random.normal(0, σ_u, size=T)
y = A_inv @ (b + u)
```

```
plt.plot(np.arange(T)+1, y)
plt.xlabel('t')
plt.ylabel('y')

plt.show()
```
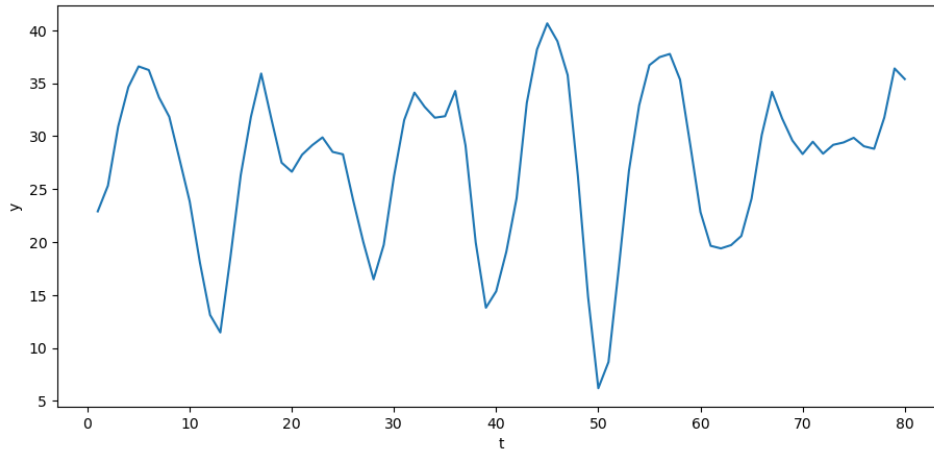

The above time series looks a lot like (detrended) GDP series for a number of advanced countries in recent decades.

We can simulate N paths.

```
N = 100

for i in range(N):
    col = cm.viridis(np.random.rand())  # Choose a random color from viridis
    u = np.random.normal(0, σ_u, size=T)
    y = A_inv @ (b + u)
    plt.plot(np.arange(T)+1, y, lw=0.5, color=col)

plt.xlabel('t')
plt.ylabel('y')

plt.show()
```
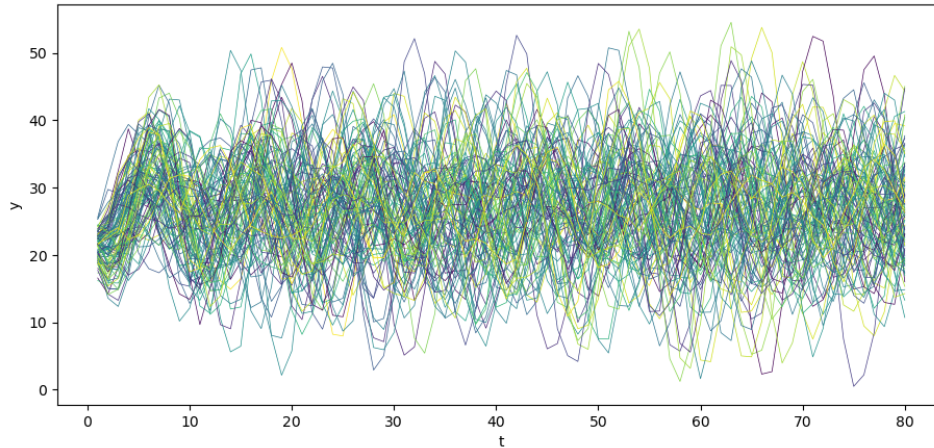
Also consider the case when y_0 and y_{-1} are at steady state.

```python
N = 100

for i in range(N):
    col = cm.viridis(np.random.rand())    # Choose a random color from viridis
    u = np.random.normal(0, σ_u, size=T)
    y_steady = A_inv @ (b_steady + u)
    plt.plot(np.arange(T)+1, y_steady, lw=0.5, color=col)

plt.xlabel('t')
plt.ylabel('y')

plt.show()
```
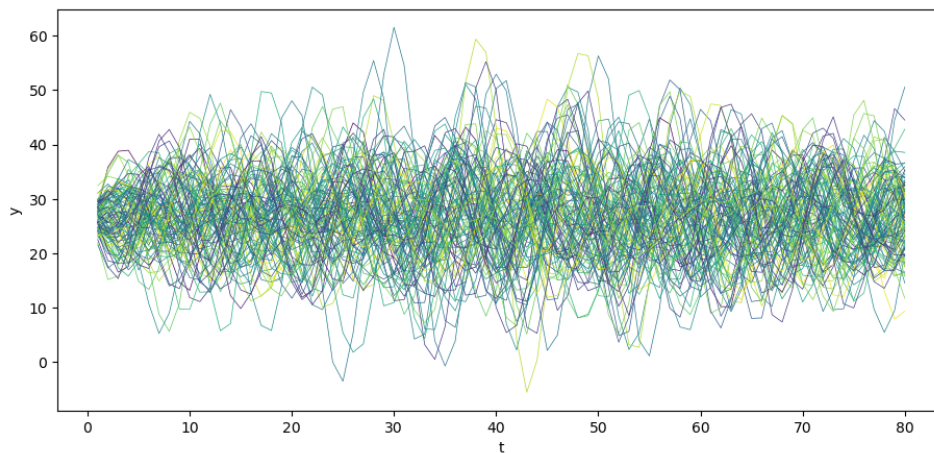
35.4 Computing population moments

We can apply standard formulas for multivariate normal distributions to compute the mean vector and covariance matrix for our time series model

$$y = A^{-1}(b + u).$$

You can read about multivariate normal distributions in this lecture Multivariate Normal Distribution.

Let's write our model as

$$y = \tilde{A}(b + u)$$

where $\tilde{A} = A^{-1}$.

Because linear combinations of normal random variables are normal, we know that

$$y \sim \mathcal{N}(\mu_y, \Sigma_y)$$

where

$$\mu_y = \tilde{A}b$$

and

$$\Sigma_y = \tilde{A}(\sigma_u^2 I_{T \times T})\tilde{A}^T$$

Let's write a Python class that computes the mean vector μ_y and covariance matrix Σ_y.

```python
class population_moments:
    """
    Compute population moments μ_y, Σ_y.
    ---------
    Parameters:
    a_0, a_1, a_2, T, y_neg1, y_0
    """
    def __init__(self, a_0=10.0,
                       a_1=1.53,
                       a_2=-.9,
                       T=80,
                       y_neg1=28.0,
                       y_0=24.0,
                       σ_u=1):

        # compute A
        A = np.identity(T)
```

```python
        for i in range(T):
            if i-1 >= 0:
                A[i, i-1] = -α_1

            if i-2 >= 0:
                A[i, i-2] = -α_2

        # compute b
        b = np.full(T, α_0)
        b[0] = α_0 + α_1 * y_0 + α_2 * y_neg1
        b[1] = α_0 + α_2 * y_0

        # compute A inverse
        A_inv = np.linalg.inv(A)

        self.A, self.b, self.A_inv, self.σ_u, self.T = A, b, A_inv, σ_u, T

    def sample_y(self, n):
        """
        Give a sample of size n of y.
        """
        A_inv, σ_u, b, T = self.A_inv, self.σ_u, self.b, self.T
        us = np.random.normal(0, σ_u, size=[n, T])
        ys = np.vstack([A_inv @ (b + u) for u in us])

        return ys

    def get_moments(self):
        """
        Compute the population moments of y.
        """
        A_inv, σ_u, b = self.A_inv, self.σ_u, self.b

        # compute μ_y
        self.μ_y = A_inv @ b
        self.Σ_y = σ_u**2 * (A_inv @ A_inv.T)

        return self.μ_y, self.Σ_y


series_process = population_moments()
```

```
μ_y, Σ_y = series_process.get_moments()
A_inv = series_process.A_inv
```

It is enlightening to study the μ_y, Σ_y's implied by various parameter values.

Among other things, we can use the class to exhibit how **statistical stationarity** of y prevails only for very special initial conditions.

Let's begin by generating N time realizations of y plotting them together with population mean μ_y.

```
# Plot mean
N = 100

for i in range(N):
    col = cm.viridis(np.random.rand())  # Choose a random color from viridis
    ys = series_process.sample_y(N)
    plt.plot(ys[i,:], lw=0.5, color=col)
    plt.plot(μ_y, color='red')

plt.xlabel('t')
plt.ylabel('y')

plt.show()
```
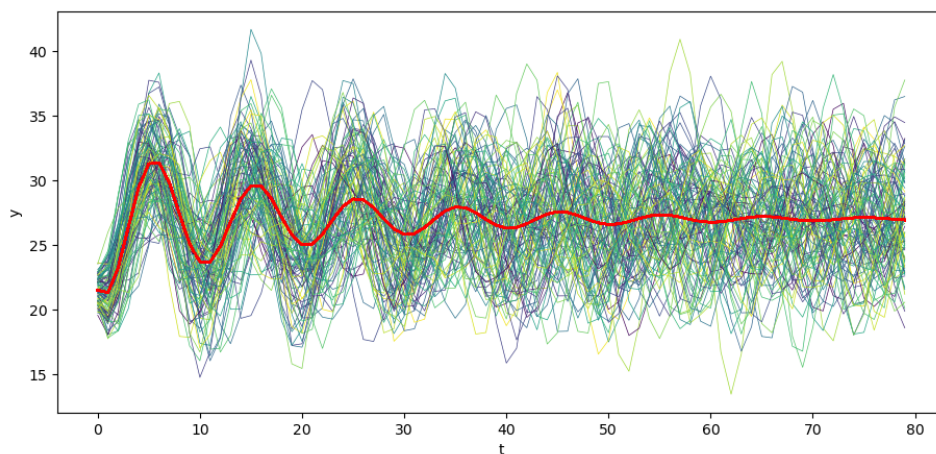

Visually, notice how the variance across realizations of y_t decreases as t increases.

Let's plot the population variance of y_t against t.

```
# Plot variance
plt.plot(Σ_y.diagonal())
plt.show()
```
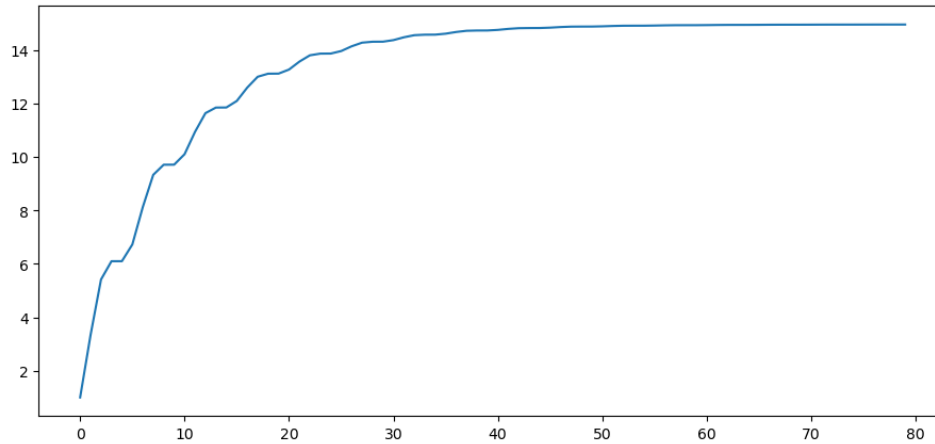
Notice how the population variance increases and asymptotes.

Let's print out the covariance matrix Σ_y for a time series y.

```
series_process = population_moments(α_0=0,
                                     α_1=.8,
                                     α_2=0,
                                     T=6,
                                     y_neg1=0.,
                                     y_0=0.,
                                     σ_u=1)

μ_y, Σ_y = series_process.get_moments()
print("μ_y = ", μ_y)
print("Σ_y = \n", Σ_y)
```

```
μ_y =  [0. 0. 0. 0. 0. 0.]
Σ_y =
 [[1.    0.8   0.64  0.512 0.41  0.328]
 [0.8   1.64  1.312 1.05  0.84  0.672]
 [0.64  1.312 2.05  1.64  1.312 1.049]
 [0.512 1.05  1.64  2.312 1.849 1.48 ]
 [0.41  0.84  1.312 1.849 2.48  1.984]
 [0.328 0.672 1.049 1.48  1.984 2.587]]
```

Notice that the covariance between y_t and y_{t-1} –the elements on the superdiagonal –are *not* identical.

This is an indication that the time series represented by our y vector is not **stationary**.

To make it stationary, we'd have to alter our system so that our *initial conditions* (y_0, y_{-1}) are not fixed numbers but instead a jointly normally distributed random vector with a particular mean and covariance matrix.

We describe how to do that in Linear State Space Models.

But just to set the stage for that analysis, let's print out the bottom right corner of Σ_y.

```
series_process = population_moments()
μ_y, Σ_y = series_process.get_moments()

print("bottom right corner of Σ_y = \n", Σ_y[72:,72:])
```

```
bottom right corner of Σ_y =
 [[ 14.965  12.051   4.969  -3.243  -9.434 -11.515  -9.128  -3.602]
  [ 12.051  14.965  12.051   4.969  -3.243  -9.434 -11.515  -9.128]
  [  4.969  12.051  14.966  12.051   4.97   -3.243  -9.434 -11.516]
  [ -3.243   4.969  12.051  14.966  12.052   4.97   -3.243  -9.434]
  [ -9.434  -3.243   4.97   12.052  14.967  12.053   4.97   -3.243]
  [-11.515  -9.434  -3.243   4.97   12.053  14.968  12.053   4.97 ]
  [ -9.128 -11.515  -9.434  -3.243   4.97   12.053  14.968  12.053]
  [ -3.602  -9.128 -11.516  -9.434  -3.243   4.97   12.053  14.968]]
```

Please notice how the subdiagonal and superdiagonal elements seem to have converged.

This is an indication that our process is asymptotically stationary.

You can read about stationarity of more general linear time series models in this lecture Linear State Space Models.

There is a lot to be learned about the process by staring at the off diagonal elements of Σ_y corresponding to different time periods t, but we resist the temptation to do so here.

35.5 Moving average representation

Let's print out A^{-1} and stare at its structure

- is it triangular or almost triangular or ... ?

To study the structure of A^{-1}, we shall print just up to 3 decimals.

Let's begin by printing out just the upper left hand corner of A^{-1}.

```
print(A_inv[0:7,0:7])
```

```
[[ 1.     0.    -0.    -0.     0.    -0.    -0.   ]
 [ 1.53   1.    -0.    -0.     0.    -0.    -0.   ]
 [ 1.441  1.53   1.     0.     0.     0.     0.   ]
 [ 0.828  1.441  1.53   1.     0.     0.     0.   ]
 [-0.031  0.828  1.441  1.53   1.    -0.    -0.   ]
```

```
    [-0.792 -0.031  0.828  1.441  1.53   1.     0.   ]
    [-1.184 -0.792 -0.031  0.828  1.441  1.53   1.   ]]
```

Evidently, A^{-1} is a lower triangular matrix.

Notice how every row ends with the previous row's pre-diagonal entries.

Since A^{-1} is lower triangular, each row represents y_t for a particular t as the sum of

- a time-dependent function $A^{-1}b$ of the initial conditions incorporated in b, and

- a weighted sum of current and past values of the IID shocks $\{u_t\}$.

Thus, let $\tilde{A} = A^{-1}$.

Evidently, for $t \geq 0$,

$$y_{t+1} = \sum_{i=1}^{t+1} \tilde{A}_{t+1,i} b_i + \sum_{i=1}^{t} \tilde{A}_{t+1,i} u_i + u_{t+1}$$

This is a **moving average** representation with time-varying coefficients.

Just as system (**??**) constitutes a **moving average** representation for y, system (**??**) constitutes an **autoregressive** representation for y.

35.6 A forward looking model

Samuelson's model is *backward looking* in the sense that we give it *initial conditions* and let it run.

Let's now turn to model that is *forward looking*.

We apply similar linear algebra machinery to study a *perfect foresight* model widely used as a benchmark in macroeconomics and finance.

As an example, we suppose that p_t is the price of a stock and that y_t is its dividend.

We assume that y_t is determined by second-order difference equation that we analyzed just above, so that

$$y = A^{-1} (b + u)$$

Our *perfect foresight* model of stock prices is

$$p_t = \sum_{j=0}^{T-t} \beta^j y_{t+j}, \quad \beta \in (0, 1)$$

where β is a discount factor.

The model asserts that the price of the stock at t equals the discounted present values of the (perfectly foreseen) future dividends.

Form

$$
\underbrace{\begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ \vdots \\ p_T \end{bmatrix}}_{\equiv p} = \underbrace{\begin{bmatrix} 1 & \beta & \beta^2 & \cdots & \beta^{T-1} \\ 0 & 1 & \beta & \cdots & \beta^{T-2} \\ 0 & 0 & 1 & \cdots & \beta^{T-3} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}}_{\equiv B} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_T \end{bmatrix}
$$

```
β = .96
```

```
# construct B
B = np.zeros((T, T))

for i in range(T):
    B[i, i:] = β ** np.arange(0, T-i)
```

```
print(B)
```

```
[[1.    0.96  0.922 ... 0.043 0.041 0.04 ]
 [0.    1.    0.96  ... 0.045 0.043 0.041]
 [0.    0.    1.    ... 0.047 0.045 0.043]
 ...
 [0.    0.    0.    ... 1.    0.96  0.922]
 [0.    0.    0.    ... 0.    1.    0.96 ]
 [0.    0.    0.    ... 0.    0.    1.    ]]
```

```
σ_u = 0.
u = np.random.normal(0, σ_u, size=T)
y = A_inv @ (b + u)
y_steady = A_inv @ (b_steady + u)
```

```
p = B @ y
```

```
plt.plot(np.arange(0, T)+1, y, label='y')
plt.plot(np.arange(0, T)+1, p, label='p')
plt.xlabel('t')
plt.ylabel('y/p')
plt.legend()

plt.show()
```
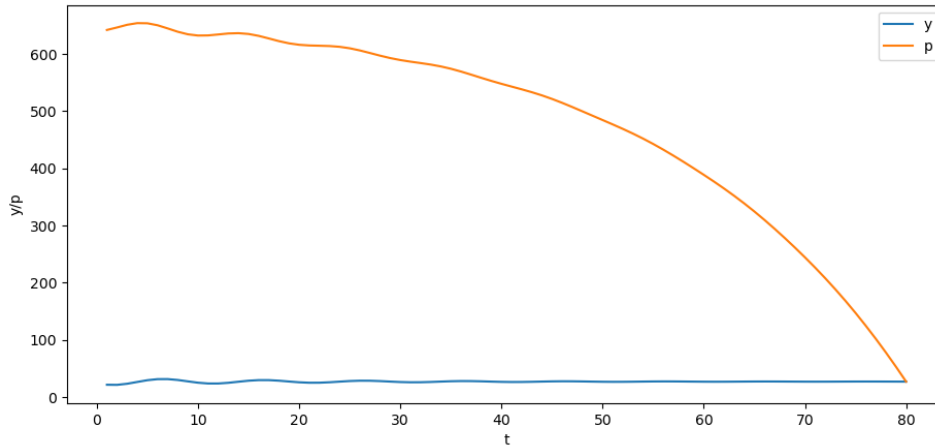
Can you explain why the trend of the price is downward over time?

Also consider the case when y_0 and y_{-1} are at the steady state.

```python
p_steady = B @ y_steady

plt.plot(np.arange(0, T)+1, y_steady, label='y')
plt.plot(np.arange(0, T)+1, p_steady, label='p')
plt.xlabel('t')
plt.ylabel('y/p')
plt.legend()

plt.show()
```
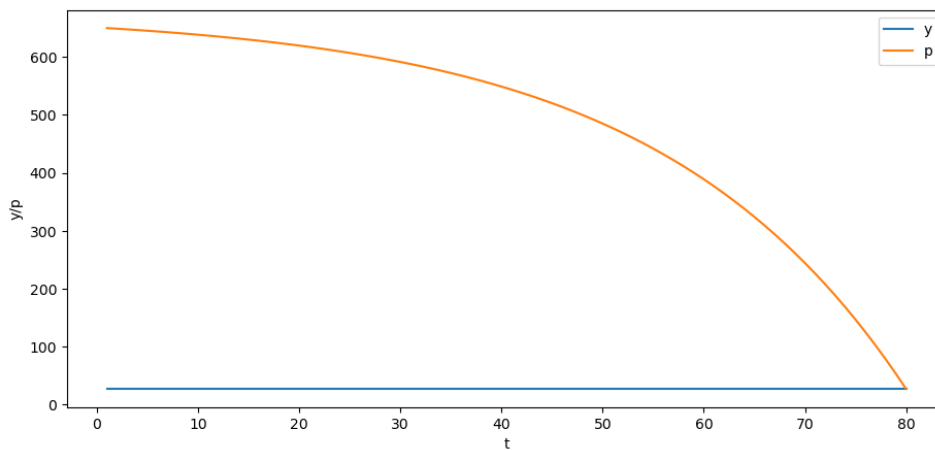
Part X

Optimization

LINEAR PROGRAMMING

In this lecture, we will need the following library. Install ortools using `pip`.

```
!pip install ortools
```

36.1 Overview

Linear programming problems either maximize or minimize a linear objective function subject to a set of linear equality and/or inequality constraints.

Linear programs come in pairs:

- an original **primal** problem, and

- an associated **dual** problem.

If a primal problem involves *maximization*, the dual problem involves *minimization*.

If a primal problem involves *minimization**, the dual problem involves *maximization.

We provide a standard form of a linear program and methods to transform other forms of linear programming problems into a standard form.

We tell how to solve a linear programming problem using SciPy and Google OR-Tools.

See also:

In another lecture, we will employ the linear programming method to solve the optimal transport problem.

Let's start with some standard imports.

```
import numpy as np
from ortools.linear_solver import pywraplp
from scipy.optimize import linprog
```

```python
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon
```

Let's start with some examples of linear programming problem.

36.2 Example 1: production problem

This example was created by [Bertsimas, 1997]

Suppose that a factory can produce two goods called Product 1 and Product 2.

To produce each product requires both material and labor.

Selling each product generates revenue.

Required per unit material and labor inputs and revenues are shown in table below:

	Product 1	Product 2
Material	2	5
Labor	4	2
Revenue	3	4

30 units of material and 20 units of labor available.

A firm's problem is to construct a production plan that uses its 30 units of materials and 20 units of labor to maximize its revenue.

Let x_i denote the quantity of Product i that the firm produces and z denote the total revenue.

This problem can be formulated as:

$$\max_{x_1, x_2} \; z = 3x_1 + 4x_2$$
$$\text{subject to } \; 2x_1 + 5x_2 \leq 30$$
$$4x_1 + 2x_2 \leq 20$$
$$x_1, x_2 \geq 0$$

The following graph illustrates the firm's constraints and iso-revenue lines.

Iso-revenue lines show all the combinations of materials and labor that produce the same revenue.
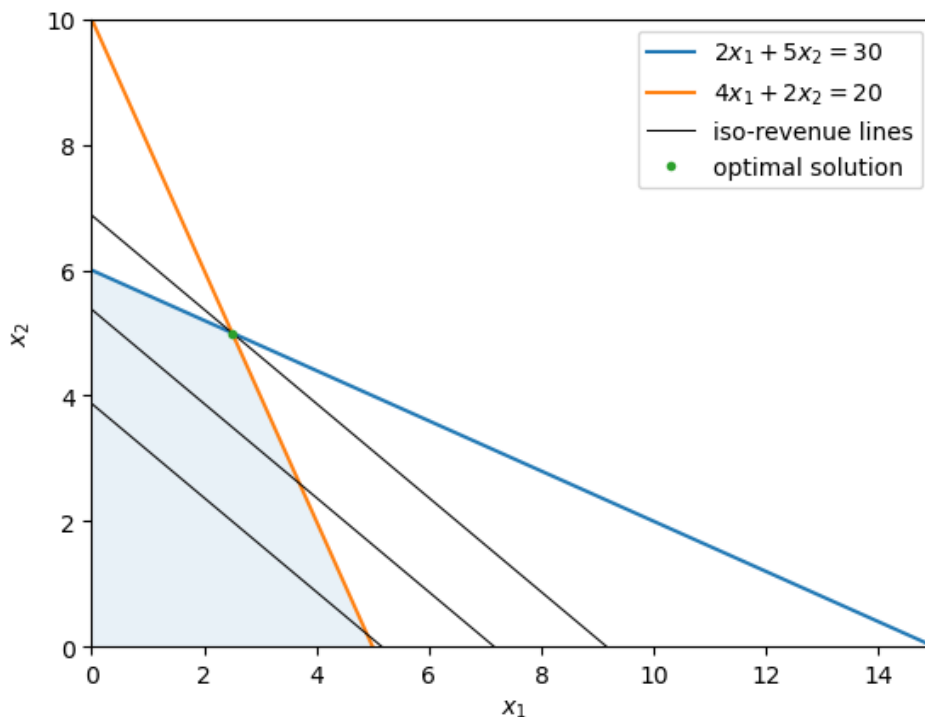
The blue region is the feasible set within which all constraints are satisfied.

Parallel black lines are iso-revenue lines.

The firm's objective is to find the parallel black lines to the upper boundary of the feasible set.

The intersection of the feasible set and the highest black line delineates the optimal set.

In this example, the optimal set is the point $(2.5, 5)$.

36.2.1 Computation: using OR-Tools

Let's try to solve the same problem using the package `ortools.linear_solver`.

The following cell instantiates a solver and creates two variables specifying the range of values that they can have.

```
# Instantiate a GLOP(Google Linear Optimization Package) solver
solver = pywraplp.Solver.CreateSolver('GLOP')
```

Let's create two variables x_1 and x_2 such that they can only have nonnegative values.

```
# Create the two variables and let them take on any non-negative value.
x1 = solver.NumVar(0, solver.infinity(), 'x1')
x2 = solver.NumVar(0, solver.infinity(), 'x2')
```

Add the constraints to the problem.

```
# Constraint 1: 2x_1 + 5x_2 <= 30.0
solver.Add(2 * x1 + 5 * x2 <= 30.0)


# Constraint 2: 4x_1 + 2x_2 <= 20.0
solver.Add(4 * x1 + 2 * x2 <= 20.0)
```

```
<ortools.linear_solver.pywraplp.Constraint; proxy of <Swig Object of type
↪'operations_research::MPConstraint *' at 0x120ea3e40> >
```

Let's specify the objective function. We use `solver.Maximize` method in the case when we want to maximize the objective function and in the case of minimization we can use `solver.Minimize`.

```
# Objective function: 3x_1 + 4x_2
solver.Maximize(3 * x1 + 4 * x2)
```

Once we solve the problem, we can check whether the solver was successful in solving the problem using its status. If it's successful, then the status will be equal to `pywraplp.Solver.OPTIMAL`.

```
# Solve the system.
status = solver.Solve()


if status == pywraplp.Solver.OPTIMAL:
    print('Objective value =', solver.Objective().Value())
    print(f'(x1, x2): ({x1.solution_value():.2}, {x2.solution_value():.2})')
else:
    print('The problem does not have an optimal solution.')
```

```
Objective value = 27.5
(x1, x2): (2.5, 5.0)
```

36.3 Example 2: investment problem

We now consider a problem posed and solved by [Hu, 2018].

A mutual fund has $100,000$ to be invested over a three-year horizon.

Three investment options are available:

1. Annuity: the fund can pay a same amount of new capital at the beginning of each of three years and receive a payoff of 130% of total capital invested at the end of the third year. Once the mutual fund decides to invest in this

annuity, it has to keep investing in all subsequent years in the three year horizon.

2. Bank account: the fund can deposit any amount into a bank at the beginning of each year and receive its capital plus 6% interest at the end of that year. In addition, the mutual fund is permitted to borrow no more than $20,000 at the beginning of each year and is asked to pay back the amount borrowed plus 6% interest at the end of the year. The mutual fund can choose whether to deposit or borrow at the beginning of each year.

3. Corporate bond: At the beginning of the second year, a corporate bond becomes available. The fund can buy an amount that is no more than $50,000 of this bond at the beginning of the second year and at the end of the third year receive a payout of 130% of the amount invested in the bond.

The mutual fund's objective is to maximize total payout that it owns at the end of the third year.

We can formulate this as a linear programming problem.

Let x_1 be the amount of put in the annuity, x_2, x_3, x_4 be bank deposit balances at the beginning of the three years, and x_5 be the amount invested in the corporate bond.

When x_2, x_3, x_4 are negative, it means that the mutual fund has borrowed from bank.

The table below shows the mutual fund's decision variables together with the timing protocol described above:

	Year 1	Year 2	Year 3
Annuity	x_1	x_1	x_1
Bank account	x_2	x_3	x_4
Corporate bond	0	x_5	0

The mutual fund's decision making proceeds according to the following timing protocol:

1. At the beginning of the first year, the mutual fund decides how much to invest in the annuity and how much to deposit in the bank. This decision is subject to the constraint:

$$x_1 + x_2 = 100,000$$

2. At the beginning of the second year, the mutual fund has a bank balance of $1.06x_2$. It must keep x_1 in the annuity. It can choose to put x_5 into the corporate bond, and put x_3 in the bank. These decisions are restricted by

$$x_1 + x_5 = 1.06x_2 - x_3$$

3. At the beginning of the third year, the mutual fund has a bank account balance equal to $1.06x_3$. It must again invest x_1 in the annuity, leaving it with a bank account balance equal to x_4. This situation is summarized by the restriction:

$$x_1 = 1.06x_3 - x_4$$

The mutual fund's objective function, i.e., its wealth at the end of the third year is:

$$1.30 \cdot 3x_1 + 1.06x_4 + 1.30x_5$$

Thus, the mutual fund confronts the linear program:

$$
\begin{aligned}
\max_x \ & 1.30 \cdot 3x_1 + 1.06x_4 + 1.30x_5 \\
\text{subject to } \ & x_1 + x_2 = 100,000 \\
& x_1 - 1.06x_2 + x_3 + x_5 = 0 \\
& x_1 - 1.06x_3 + x_4 = 0 \\
& x_2 \geq -20,000 \\
& x_3 \geq -20,000 \\
& x_4 \geq -20,000 \\
& x_5 \leq 50,000 \\
& x_j \geq 0, \quad j = 1, 5 \\
& x_j \text{ unrestricted}, \quad j = 2, 3, 4
\end{aligned}
$$

36.3.1 Computation: using OR-Tools

Let's try to solve the above problem using the package `ortools.linear_solver`.

The following cell instantiates a solver and creates two variables specifying the range of values that they can have.

```
# Instantiate a GLOP(Google Linear Optimization Package) solver
solver = pywraplp.Solver.CreateSolver('GLOP')
```

Let's create five variables x_1, x_2, x_3, x_4, and x_5 such that they can only have the values defined in the above constraints.

```
# Create the variables using the ranges available from constraints
x1 = solver.NumVar(0, solver.infinity(), 'x1')
x2 = solver.NumVar(-20_000, solver.infinity(), 'x2')
x3 = solver.NumVar(-20_000, solver.infinity(), 'x3')
x4 = solver.NumVar(-20_000, solver.infinity(), 'x4')
x5 = solver.NumVar(0, 50_000, 'x5')
```

Add the constraints to the problem.

```
# Constraint 1: x_1 + x_2 = 100,000
solver.Add(x1 + x2 == 100_000.0)

# Constraint 2: x_1 - 1.06 * x_2 + x_3 + x_5 = 0
```

(continues on next page)

```
solver.Add(x1 - 1.06 * x2 + x3 + x5 == 0.0)


# Constraint 3: x_1 - 1.06 * x_3 + x_4 = 0
solver.Add(x1 - 1.06 * x3 + x4 == 0.0)
```

```
<ortools.linear_solver.pywraplp.Constraint; proxy of <Swig Object of type
 ↪'operations_research::MPConstraint *' at 0x120ecad90> >
```

Let's specify the objective function.

```
# Objective function: 1.30 * 3 * x_1 + 1.06 * x_4 + 1.30 * x_5
solver.Maximize(1.30 * 3 * x1 + 1.06 * x4 + 1.30 * x5)
```

Let's solve the problem and check the status using `pywraplp.Solver.OPTIMAL`.

```
# Solve the system.
status = solver.Solve()


if status == pywraplp.Solver.OPTIMAL:
    print('Objective value =', solver.Objective().Value())
    x1_sol = round(x1.solution_value(), 3)
    x2_sol = round(x2.solution_value(), 3)
    x3_sol = round(x1.solution_value(), 3)
    x4_sol = round(x2.solution_value(), 3)
    x5_sol = round(x1.solution_value(), 3)
    print(f'(x1, x2, x3, x4, x5): ({x1_sol}, {x2_sol}, {x3_sol}, {x4_sol}, {x5_sol})')
else:
    print('The problem does not have an optimal solution.')
```

```
Objective value = 141018.24349792692
(x1, x2, x3, x4, x5): (24927.755, 75072.245, 24927.755, 75072.245, 24927.755)
```

OR-Tools tells us that the best investment strategy is:

1. At the beginning of the first year, the mutual fund should buy $24,927.755$ of the annuity. Its bank account balance should be $75,072.245$.

2. At the beginning of the second year, the mutual fund should buy $24,927.755$ of the corporate bond and keep invest in the annuity. Its bank balance should be $24,927.755$.

3. At the beginning of the third year, the bank balance should be $75,072.245$.

4. At the end of the third year, the mutual fund will get payouts from the annuity and corporate bond and repay its

---

loan from the bank. At the end it will own $\$141,018.24$, so that it's total net rate of return over the three periods is 41.02%.

36.4 Standard form

For purposes of

- unifying linear programs that are initially stated in superficially different forms, and

- having a form that is convenient to put into black-box software packages,

it is useful to devote some effort to describe a **standard form**.

Our standard form is:

$$\min_{x} c_1 x_1 + c_2 x_2 + \cdots + c_n x_n$$

$$\text{subject to } a_{11} x_1 + a_{12} x_2 + \cdots + a_{1n} x_n = b_1$$

$$a_{21} x_1 + a_{22} x_2 + \cdots + a_{2n} x_n = b_2$$

$$\vdots$$

$$a_{m1} x_1 + a_{m2} x_2 + \cdots + a_{mn} x_n = b_m$$

$$x_1, x_2, \ldots, x_n \geq 0$$

Let

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ & & \vdots & \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}, \quad c = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

The standard form linear programming problem can be expressed concisely as:

$$\min_{x} c'x$$

$$\text{subject to } Ax = b \tag{36.1}$$

$$x \geq 0$$

Here, $Ax = b$ means that the i-th entry of Ax equals the i-th entry of b for every i.

Similarly, $x \geq 0$ means that x_j is greater than equal to 0 for every j.

36.4.1 Useful transformations

It is useful to know how to transform a problem that initially is not stated in the standard form into one that is.

By deploying the following steps, any linear programming problem can be transformed into an equivalent standard form linear programming problem.

1. Objective function: If a problem is originally a constrained *maximization* problem, we can construct a new objective function that is the additive inverse of the original objective function. The transformed problem is then a *minimization* problem.

2. Decision variables: Given a variable x_j satisfying $x_j \leq 0$, we can introduce a new variable $x_j' = -x_j$ and substitute it into original problem. Given a free variable x_i with no restriction on its sign, we can introduce two new variables x_j^+ and x_j^- satisfying $x_j^+, x_j^- \geq 0$ and replace x_j by $x_j^+ - x_j^-$.

3. Inequality constraints: Given an inequality constraint $\sum_{j=1}^{n} a_{ij}x_j \leq 0$, we can introduce a new variable s_i, called a **slack variable** that satisfies $s_i \geq 0$ and replace the original constraint by $\sum_{j=1}^{n} a_{ij}x_j + s_i = 0$.

Let's apply the above steps to the two examples described above.

36.4.2 Example 1: production problem

The original problem is:

$$\max_{x_1, x_2} 3x_1 + 4x_2$$
$$\text{subject to } 2x_1 + 5x_2 \leq 30$$
$$4x_1 + 2x_2 \leq 20$$
$$x_1, x_2 \geq 0$$

This problem is equivalent to the following problem with a standard form:

$$\min_{x_1, x_2} -(3x_1 + 4x_2)$$
$$\text{subject to } 2x_1 + 5x_2 + s_1 = 30$$
$$4x_1 + 2x_2 + s_2 = 20$$
$$x_1, x_2, s_1, s_2 \geq 0$$

36.4.3 Computation: using SciPy

The package `scipy.optimize` provides a function `linprog` to solve linear programming problems with a form below:

$$\min_{x} c'x$$

$$\text{subject to} \ \ A_{ub}x \leq b_{ub}$$

$$A_{eq}x = b_{eq}$$

$$l \leq x \leq u$$

A_{eq}, b_{eq} denote the equality constraint matrix and vector, and A_{ub}, b_{ub} denote the inequality constraint matrix and vector.

Note: By default $l = 0$ and $u = $ None unless explicitly specified with the argument `bounds`.

Let's now try to solve the Problem 1 using SciPy.

```
# Construct parameters
c_ex1 = np.array([3, 4])

# Inequality constraints
A_ex1 = np.array([[2, 5],
                  [4, 2]])
b_ex1 = np.array([30,20])
```

Once we solve the problem, we can check whether the solver was successful in solving the problem using the boolean attribute `success`. If it's successful, then the `success` attribute is set to `True`.

```
# Solve the problem
# we put a negative sign on the objective as linprog does minimization
res_ex1 = linprog(-c_ex1, A_ub=A_ex1, b_ub=b_ex1)

if res_ex1.success:
    # We use negative sign to get the optimal value (maximized value)
    print('Optimal Value:', -res_ex1.fun)
    print(f'(x1, x2): {res_ex1.x[0], res_ex1.x[1]}')
else:
    print('The problem does not have an optimal solution.')
```

```
Optimal Value: 27.5
(x1, x2): (2.5, 5.0)
```

The optimal plan tells the factory to produce 2.5 units of Product 1 and 5 units of Product 2; that generates a maximizing value of revenue of 27.5.

We are using the `linprog` function as a *black box*.

Inside it, Python first transforms the problem into standard form.

To do that, for each inequality constraint it generates one slack variable.

Here the vector of slack variables is a two-dimensional NumPy array that equals $b_{ub} - A_{ub}x$.

See the official documentation for more details.

Note: This problem is to maximize the objective, so that we need to put a minus sign in front of parameter vector c.

36.4.4 Example 2: investment problem

The original problem is:

$$\max_{x} 1.30 \cdot 3x_1 + 1.06x_4 + 1.30x_5$$

$$\text{subject to } x_1 + x_2 = 100,000$$

$$x_1 - 1.06x_2 + x_3 + x_5 = 0$$

$$x_1 - 1.06x_3 + x_4 = 0$$

$$x_2 \geq -20,000$$

$$x_3 \geq -20,000$$

$$x_4 \geq -20,000$$

$$x_5 \leq 50,000$$

$$x_j \geq 0, \quad j = 1, 5$$

$$x_j \text{ unrestricted}, \quad j = 2, 3, 4$$

This problem is equivalent to the following problem with a standard form:

$$\min_{x} - (1.30 \cdot 3x_1 + 1.06x_4^+ - 1.06x_4^- + 1.30x_5)$$

$$\text{subject to } x_1 + x_2^+ - x_2^- = 100,000$$

$$x_1 - 1.06(x_2^+ - x_2^-) + x_3^+ - x_3^- + x_5 = 0$$

$$x_1 - 1.06(x_3^+ - x_3^-) + x_4^+ - x_4^- = 0$$

$$x_2^- - x_2^+ + s_1 = 20,000$$

$$x_3^- - x_3^+ + s_2 = 20,000$$

$$x_4^- - x_4^+ + s_3 = 20,000$$

$$x_5 + s_4 = 50,000$$

$$x_j \geq 0, \quad j = 1, 5$$

$$x_j^+, x_j^- \geq 0, \quad j = 2, 3, 4$$

$$s_j \geq 0, \quad j = 1, 2, 3, 4$$

```
# Construct parameters
rate = 1.06

# Objective function parameters
c_ex2 = np.array([1.30*3, 0, 0, 1.06, 1.30])

# Inequality constraints
A_ex2 = np.array([[1,  1,   0,   0,   0],
                  [1, -rate, 1, 0, 1],
                  [1, 0, -rate, 1, 0]])
b_ex2 = np.array([100_000, 0, 0])

# Bounds on decision variables
bounds_ex2 = [(  0,      None),
              (-20_000, None),
              (-20_000, None),
              (-20_000, None),
              (  0,    50_000)]
```

Let's solve the problem and check the status using `success` attribute.

```
# Solve the problem
res_ex2 = linprog(-c_ex2, A_eq=A_ex2, b_eq=b_ex2,
                  bounds=bounds_ex2)

if res_ex2.success:
    # We use negative sign to get the optimal value (maximized value)
    print('Optimal Value:', -res_ex2.fun)
    x1_sol = round(res_ex2.x[0], 3)
    x2_sol = round(res_ex2.x[1], 3)
    x3_sol = round(res_ex2.x[2], 3)
    x4_sol = round(res_ex2.x[3], 3)
    x5_sol = round(res_ex2.x[4], 3)
    print(f'(x1, x2, x3, x4, x5): {x1_sol, x2_sol, x3_sol, x4_sol, x5_sol}')
else:
    print('The problem does not have an optimal solution.')
```

```
Optimal Value: 141018.24349792697
(x1, x2, x3, x4, x5): (24927.755, 75072.245, 4648.825, -20000.0, 50000.0)
```

SciPy tells us that the best investment strategy is:

1. At the beginning of the first year, the mutual fund should buy $24,927.75$ of the annuity. Its bank account balance should be $75,072.25$.

---

2. At the beginning of the second year, the mutual fund should buy $50,000 of the corporate bond and keep invest in the annuity. Its bank account balance should be $4,648.83.

3. At the beginning of the third year, the mutual fund should borrow $20,000 from the bank and invest in the annuity.

4. At the end of the third year, the mutual fund will get payouts from the annuity and corporate bond and repay its loan from the bank. At the end it will own $141,018.24, so that it's total net rate of return over the three periods is 41.02%.

Note: You might notice the difference in the values of optimal solution using OR-Tools and SciPy but the optimal value is the same. It is because there can be many optimal solutions for the same problem.

36.5 Exercises

Exercise 36.5.1

Implement a new extended solution for the Problem 1 where in the factory owner decides that number of units of Product 1 should not be less than the number of units of Product 2.

Solution to Exercise 36.5.1

So we can reformulate the problem as:

$$\max_{x_1, x_2} z = 3x_1 + 4x_2$$

$$\text{subject to } 2x_1 + 5x_2 \leq 30$$

$$4x_1 + 2x_2 \leq 20$$

$$x_1 \geq x_2$$

$$x_1, x_2 \geq 0$$

```
# Instantiate a GLOP(Google Linear Optimization Package) solver
solver = pywraplp.Solver.CreateSolver('GLOP')

# Create the two variables and let them take on any non-negative value.
x1 = solver.NumVar(0, solver.infinity(), 'x1')
x2 = solver.NumVar(0, solver.infinity(), 'x2')
```

```
# Constraint 1: 2x_1 + 5x_2 <= 30.0
solver.Add(2 * x1 + 5 * x2 <= 30.0)
```

36.5. Exercises

```
# Constraint 2: 4x_1 + 2x_2 <= 20.0
solver.Add(4 * x1 + 2 * x2 <= 20.0)


# Constraint 3: x_1 >= x_2
solver.Add(x1 >= x2)
```

```
<ortools.linear_solver.pywraplp.Constraint; proxy of <Swig Object of type
 ↪'operations_research::MPConstraint *' at 0x120eca1f0> >
```

```
# Objective function: 3x_1 + 4x_2
solver.Maximize(3 * x1 + 4 * x2)
```

```
# Solve the system.
status = solver.Solve()

if status == pywraplp.Solver.OPTIMAL:
    print('Objective value =', solver.Objective().Value())
    x1_sol = round(x1.solution_value(), 2)
    x2_sol = round(x2.solution_value(), 2)
    print(f'(x1, x2): ({x1_sol}, {x2_sol})')
else:
    print('The problem does not have an optimal solution.')
```

```
Objective value = 23.333333333333336
(x1, x2): (3.33, 3.33)
```

Exercise 36.5.2

A carpenter manufactures 2 products - A and B.

Product A generates a profit of 23 and product B generates a profit of 10.

It takes 2 hours for the carpenter to produce A and 0.8 hours to produce B.

Moreover, he can't spend more than 25 hours per week and the total number of units of A and B should not be greater than 20.

Find the number of units of A and product B that he should manufacture in order to maximise his profit.

Solution to Exercise 36.5.2

Let us assume the carpenter produces x units of A and y units of B.

So we can formulate the problem as:

$$\max_{x,y} z = 23x + 10y$$

$$\text{subject to } x + y \leq 20$$

$$2x + 0.8y \leq 25$$

```
# Instantiate a GLOP(Google Linear Optimization Package) solver
solver = pywraplp.Solver.CreateSolver('GLOP')
```

Let's create two variables x_1 and x_2 such that they can only have nonnegative values.

```
# Create the two variables and let them take on any non-negative value.
x = solver.NumVar(0, solver.infinity(), 'x')
y = solver.NumVar(0, solver.infinity(), 'y')
```

```
# Constraint 1: x + y <= 20.0
solver.Add(x + y <= 20.0)
```

```
# Constraint 2: 2x + 0.8y <= 25.0
solver.Add(2 * x + 0.8 * y <= 25.0)
```

```
<ortools.linear_solver.pywraplp.Constraint; proxy of <Swig Object of type
 ↪'operations_research::MPConstraint *' at 0x120ecaf10> >
```

```
# Objective function: 23x + 10y
solver.Maximize(23 * x + 10 * y)
```

```
# Solve the system.
status = solver.Solve()

if status == pywraplp.Solver.OPTIMAL:
    print('Maximum Profit =', solver.Objective().Value())
    x_sol = round(x.solution_value(), 3)
    y_sol = round(y.solution_value(), 3)
    print(f'(x, y): ({x_sol}, {y_sol})')
else:
    print('The problem does not have an optimal solution.')
```

```
Maximum Profit = 297.5
(x, y): (7.5, 12.5)
```

SHORTEST PATHS

37.1 Overview

The shortest path problem is a classic problem in mathematics and computer science with applications in

- Economics (sequential decision making, analysis of social networks, etc.)

- Operations research and transportation

- Robotics and artificial intelligence

- Telecommunication network design and routing

- etc., etc.

Variations of the methods we discuss in this lecture are used millions of times every day, in applications such as

- Google Maps

- routing packets on the internet

For us, the shortest path problem also provides a nice introduction to the logic of **dynamic programming**.

Dynamic programming is an extremely powerful optimization technique that we apply in many lectures on this site.

The only scientific library we'll need in what follows is NumPy:

```
import numpy as np
```

37.2 Outline of the problem

The shortest path problem is one of finding how to traverse a graph from one specified node to another at minimum cost.

Consider the following graph
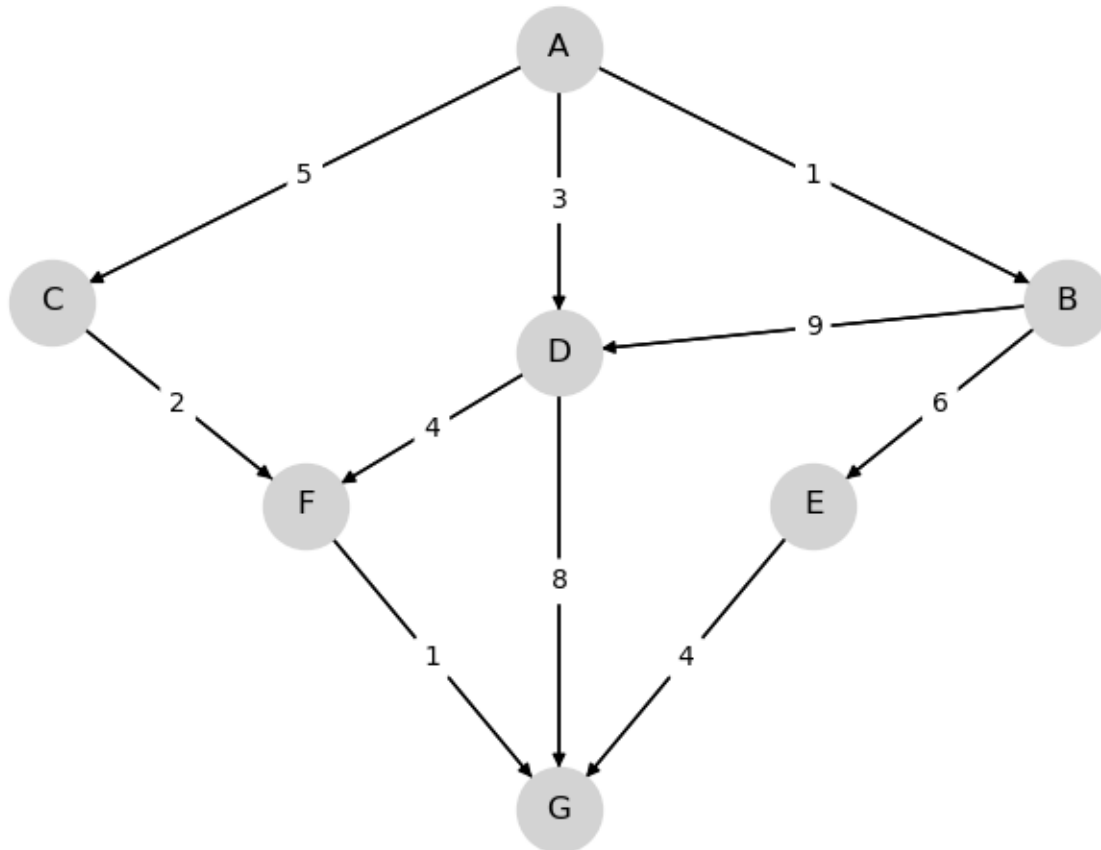
We wish to travel from node (vertex) A to node G at minimum cost

- Arrows (edges) indicate the movements we can take.

- Numbers on edges indicate the cost of traveling that edge.

(Graphs such as the one above are called weighted directed graphs.)

Possible interpretations of the graph include

- Minimum cost for supplier to reach a destination.

- Routing of packets on the internet (minimize time).

- etc., etc.

For this simple graph, a quick scan of the edges shows that the optimal paths are

- A, C, F, G at cost 8
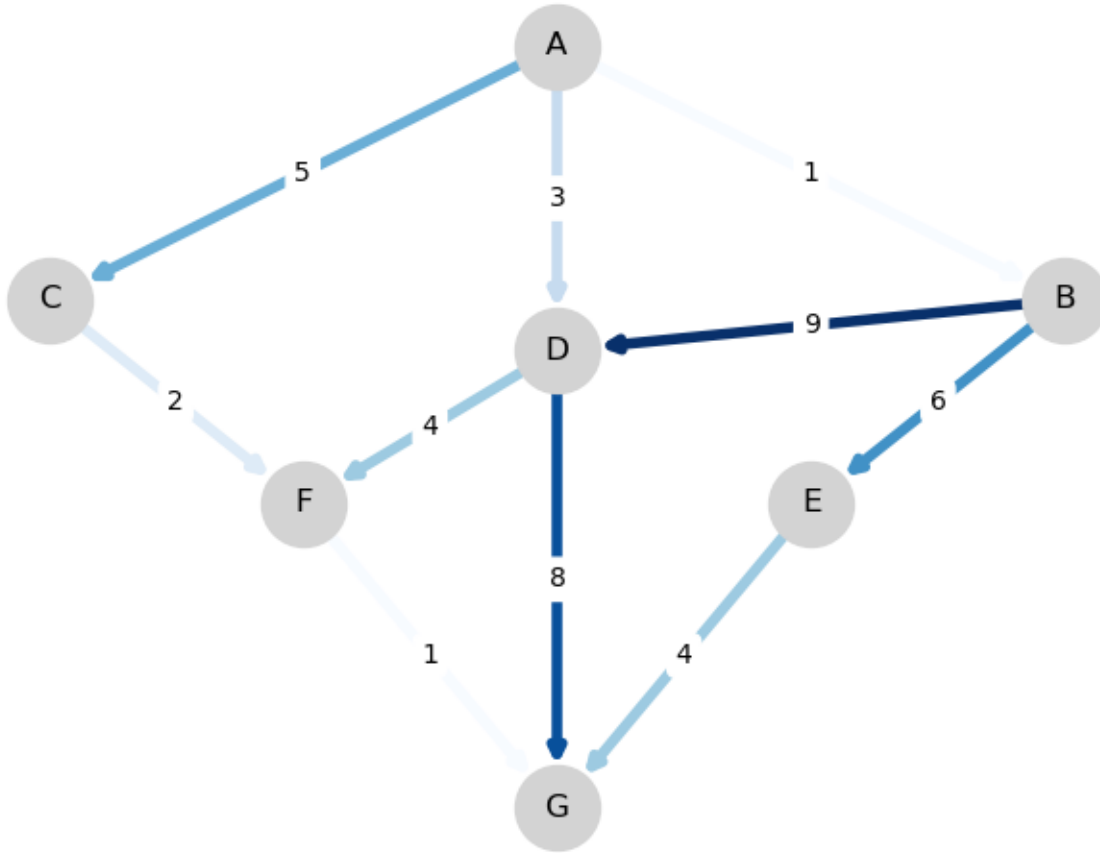
- A, D, F, G at cost 8

37.3 Finding least-cost paths

For large graphs, we need a systematic solution.

Let $J(v)$ denote the minimum cost-to-go from node v, understood as the total cost from v if we take the best route.

Suppose that we know $J(v)$ for each node v, as shown below for the graph from the preceding example.

Note that $J(G) = 0$.

The best path can now be found as follows

1. Start at node $v = A$

2. From current node v, move to any node that solves

$$\min_{w \in F_v} \{c(v, w) + J(w)\} \tag{37.1}$$
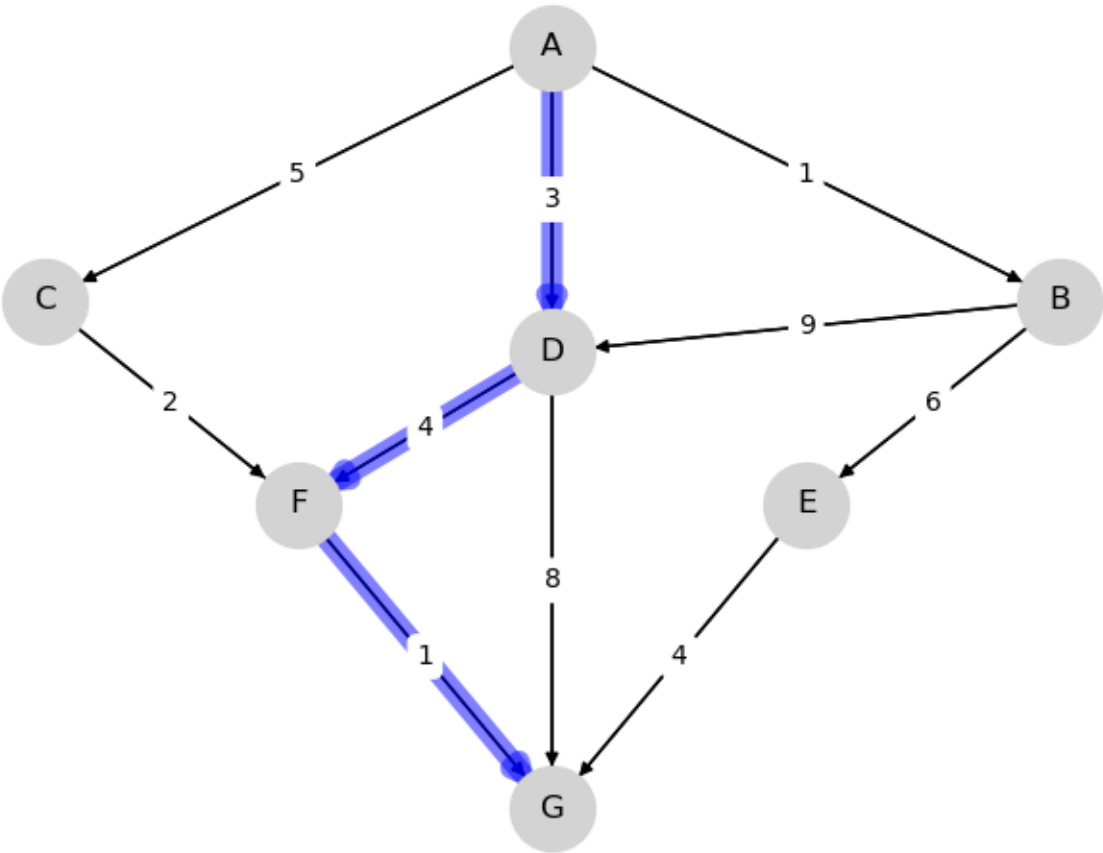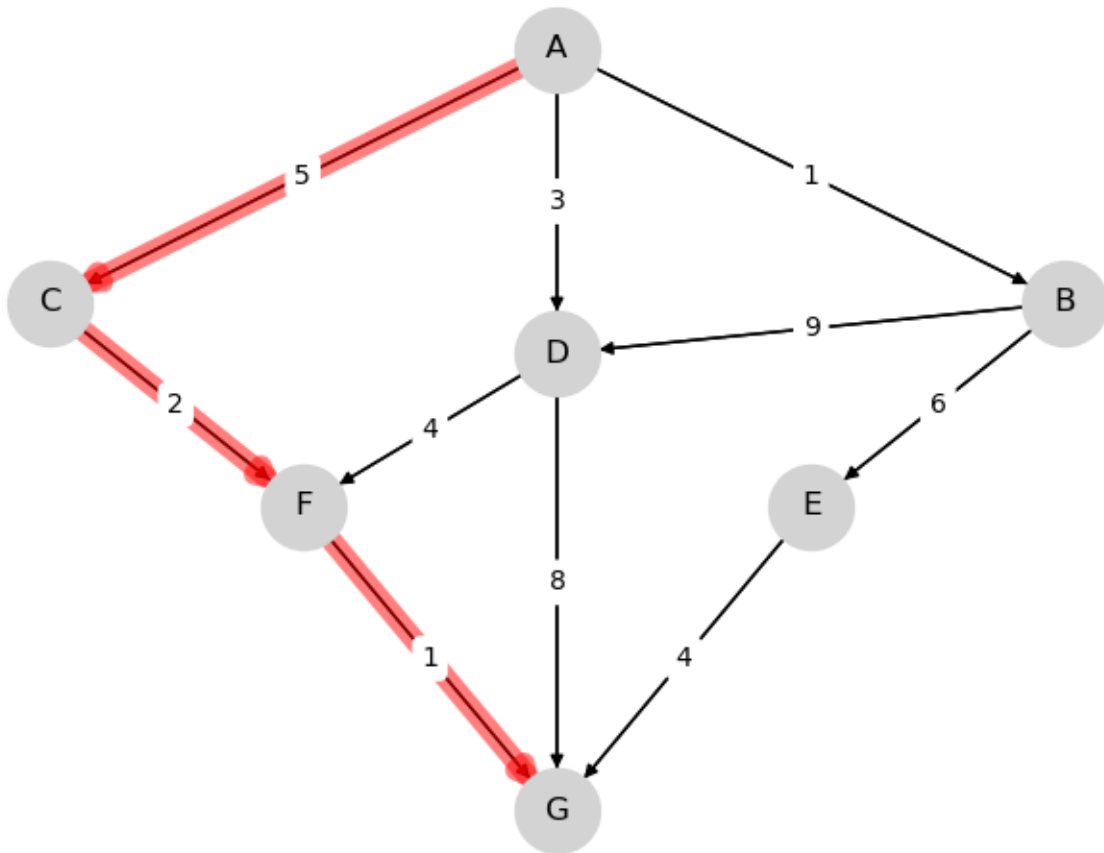
where

- F_v is the set of nodes that can be reached from v in one step.

- $c(v, w)$ is the cost of traveling from v to w.

Hence, if we know the function J, then finding the best path is almost trivial.

But how can we find the cost-to-go function J?

Some thought will convince you that, for every node v, the function J satisfies

$$J(v) = \min_{w \in F_v} \left\{ c(v, w) + J(w) \right\} \tag{37.2}$$

This is known as the **Bellman equation**, after the mathematician Richard Bellman.

The Bellman equation can be thought of as a restriction that J must satisfy.

What we want to do now is use this restriction to compute J.

37.4 Solving for minimum cost-to-go

Let's look at an algorithm for computing J and then think about how to implement it.

37.4.1 The algorithm

The standard algorithm for finding J is to start an initial guess and then iterate.

This is a standard approach to solving nonlinear equations, often called the method of **successive approximations**.

Our initial guess will be

$$J_0(v) = 0 \text{ for all } v \tag{37.3}$$

Now

1. Set $n = 0$

2. Set $J_{n+1}(v) = \min_{w \in F_v} \left\{ c(v, w) + J_n(w) \right\}$ for all v

3. If J_{n+1} and J_n are not equal then increment n, go to 2

This sequence converges to J.

Although we omit the proof, we'll prove similar claims in our other lectures on dynamic programming.

37.4.2 Implementation

Having an algorithm is a good start, but we also need to think about how to implement it on a computer.

First, for the cost function c, we'll implement it as a matrix Q, where a typical element is

$$Q(v, w) = \begin{cases} c(v, w) \text{ if } w \in F_v \\ +\infty \text{ otherwise} \end{cases}$$

In this context Q is usually called the **distance matrix**.

We're also numbering the nodes now, with $A = 0$, so, for example

$$Q(1, 2) = \text{ the cost of traveling from B to C}$$

For example, for the simple graph above, we set

```python
from numpy import inf

Q = np.array([[inf, 1,   5,   3,   inf, inf, inf],
              [inf, inf, inf, 9,   6,   inf, inf],
              [inf, inf, inf, inf, inf, 2,   inf],
              [inf, inf, inf, inf, inf, 4,   8],
              [inf, inf, inf, inf, inf, inf, 4],
              [inf, inf, inf, inf, inf, inf, 1],
              [inf, inf, inf, inf, inf, inf, 0]])
```

Notice that the cost of staying still (on the principle diagonal) is set to

- `np.inf` for non-destination nodes —moving on is required.

- 0 for the destination node —here is where we stop.

For the sequence of approximations $\{J_n\}$ of the cost-to-go functions, we can use NumPy arrays.

Let's try with this example and see how we go:

```python
nodes = range(7)                          # Nodes = 0, 1, ..., 6
J = np.zeros_like(nodes, dtype=int)        # Initial guess
next_J = np.empty_like(nodes, dtype=int)   # Stores updated guess

max_iter = 500
i = 0

while i < max_iter:
    for v in nodes:
        # Minimize Q[v, w] + J[w] over all choices of w
```

```
        next_J[v] = np.min(Q[v, :] + J)


    if np.array_equal(next_J, J):
        break


    J[:] = next_J                              # Copy contents of next_J to J
    i += 1


print("The cost-to-go function is", J)
```

```
The cost-to-go function is [ 8 10  3  5  4  1  0]
```

This matches with the numbers we obtained by inspection above.

But, importantly, we now have a methodology for tackling large graphs.

37.5 Exercises

Exercise 37.5.1

The text below describes a weighted directed graph.

The line `node0, node1 0.04, node8 11.11, node14 72.21` means that from node0 we can go to

- node1 at cost 0.04

- node8 at cost 11.11

- node14 at cost 72.21

No other nodes can be reached directly from node0.

Other lines have a similar interpretation.

Your task is to use the algorithm given above to find the optimal path and its cost.

Note: You will be dealing with floating point numbers now, rather than integers, so consider replacing `np.equal()` with `np.allclose()`.

```
%%file graph.txt
node0, node1 0.04, node8 11.11, node14 72.21
```

```
node1, node46 1247.25, node6 20.59, node13 64.94
node2, node66 54.18, node31 166.80, node45 1561.45
node3, node20 133.65, node6 2.06, node11 42.43
node4, node75 3706.67, node5 0.73, node7 1.02
node5, node45 1382.97, node7 3.33, node11 34.54
node6, node31 63.17, node9 0.72, node10 13.10
node7, node50 478.14, node9 3.15, node10 5.85
node8, node69 577.91, node11 7.45, node12 3.18
node9, node70 2454.28, node13 4.42, node20 16.53
node10, node89 5352.79, node12 1.87, node16 25.16
node11, node94 4961.32, node18 37.55, node20 65.08
node12, node84 3914.62, node24 34.32, node28 170.04
node13, node60 2135.95, node38 236.33, node40 475.33
node14, node67 1878.96, node16 2.70, node24 38.65
node15, node91 3597.11, node17 1.01, node18 2.57
node16, node36 392.92, node19 3.49, node38 278.71
node17, node76 783.29, node22 24.78, node23 26.45
node18, node91 3363.17, node23 16.23, node28 55.84
node19, node26 20.09, node20 0.24, node28 70.54
node20, node98 3523.33, node24 9.81, node33 145.80
node21, node56 626.04, node28 36.65, node31 27.06
node22, node72 1447.22, node39 136.32, node40 124.22
node23, node52 336.73, node26 2.66, node33 22.37
node24, node66 875.19, node26 1.80, node28 14.25
node25, node70 1343.63, node32 36.58, node35 45.55
node26, node47 135.78, node27 0.01, node42 122.00
node27, node65 480.55, node35 48.10, node43 246.24
node28, node82 2538.18, node34 21.79, node36 15.52
node29, node64 635.52, node32 4.22, node33 12.61
node30, node98 2616.03, node33 5.61, node35 13.95
node31, node98 3350.98, node36 20.44, node44 125.88
node32, node97 2613.92, node34 3.33, node35 1.46
node33, node81 1854.73, node41 3.23, node47 111.54
node34, node73 1075.38, node42 51.52, node48 129.45
node35, node52 17.57, node41 2.09, node50 78.81
node36, node71 1171.60, node54 101.08, node57 260.46
node37, node75 269.97, node38 0.36, node46 80.49
node38, node93 2767.85, node40 1.79, node42 8.78
node39, node50 39.88, node40 0.95, node41 1.34
node40, node75 548.68, node47 28.57, node54 53.46
node41, node53 18.23, node46 0.28, node54 162.24
node42, node59 141.86, node47 10.08, node72 437.49
node43, node98 2984.83, node54 95.06, node60 116.23
```

```
node44, node91 807.39, node46 1.56, node47 2.14
node45, node58 79.93, node47 3.68, node49 15.51
node46, node52 22.68, node57 27.50, node67 65.48
node47, node50 2.82, node56 49.31, node61 172.64
node48, node99 2564.12, node59 34.52, node60 66.44
node49, node78 53.79, node50 0.51, node56 10.89
node50, node85 251.76, node53 1.38, node55 20.10
node51, node98 2110.67, node59 23.67, node60 73.79
node52, node94 1471.80, node64 102.41, node66 123.03
node53, node72 22.85, node56 4.33, node67 88.35
node54, node88 967.59, node59 24.30, node73 238.61
node55, node84 86.09, node57 2.13, node64 60.80
node56, node76 197.03, node57 0.02, node61 11.06
node57, node86 701.09, node58 0.46, node60 7.01
node58, node83 556.70, node64 29.85, node65 34.32
node59, node90 820.66, node60 0.72, node71 0.67
node60, node76 48.03, node65 4.76, node67 1.63
node61, node98 1057.59, node63 0.95, node64 4.88
node62, node91 132.23, node64 2.94, node76 38.43
node63, node66 4.43, node72 70.08, node75 56.34
node64, node80 47.73, node65 0.30, node76 11.98
node65, node94 594.93, node66 0.64, node73 33.23
node66, node98 395.63, node68 2.66, node73 37.53
node67, node82 153.53, node68 0.09, node70 0.98
node68, node94 232.10, node70 3.35, node71 1.66
node69, node99 247.80, node70 0.06, node73 8.99
node70, node76 27.18, node72 1.50, node73 8.37
node71, node89 104.50, node74 8.86, node91 284.64
node72, node76 15.32, node84 102.77, node92 133.06
node73, node83 52.22, node76 1.40, node90 243.00
node74, node81 1.07, node76 0.52, node78 8.08
node75, node92 68.53, node76 0.81, node77 1.19
node76, node85 13.18, node77 0.45, node78 2.36
node77, node80 8.94, node78 0.98, node86 64.32
node78, node98 355.90, node81 2.59
node79, node81 0.09, node85 1.45, node91 22.35
node80, node92 121.87, node88 28.78, node98 264.34
node81, node94 99.78, node89 39.52, node92 99.89
node82, node91 47.44, node88 28.05, node93 11.99
node83, node94 114.95, node86 8.75, node88 5.78
node84, node89 19.14, node94 30.41, node98 121.05
node85, node97 94.51, node87 2.66, node89 4.90
node86, node97 85.09
```

```
node87, node88 0.21, node91 11.14, node92 21.23
node88, node93 1.31, node91 6.83, node98 6.12
node89, node97 36.97, node99 82.12
node90, node96 23.53, node94 10.47, node99 50.99
node91, node97 22.17
node92, node96 10.83, node97 11.24, node99 34.68
node93, node94 0.19, node97 6.71, node99 32.77
node94, node98 5.91, node96 2.03
node95, node98 6.17, node99 0.27
node96, node98 3.32, node97 0.43, node99 5.87
node97, node98 0.30
node98, node99 0.33
node99,
```

```
Writing graph.txt
```

Solution to Exercise 37.5.1

First let's write a function that reads in the graph data above and builds a distance matrix.

```python
num_nodes = 100
destination_node = 99


def map_graph_to_distance_matrix(in_file):

    # First let's set of the distance matrix Q with inf everywhere
    Q = np.full((num_nodes, num_nodes), np.inf)

    # Now we read in the data and modify Q
    with open(in_file) as infile:
        for line in infile:
            elements = line.split(',')
            node = elements.pop(0)
            node = int(node[4:])    # convert node description to integer
            if node != destination_node:
                for element in elements:
                    destination, cost = element.split()
                    destination = int(destination[4:])
                    Q[node, destination] = float(cost)
            Q[destination_node, destination_node] = 0
    return Q
```

In addition, let's write

1. a "Bellman operator" function that takes a distance matrix and current guess of J and returns an updated guess of J, and

2. a function that takes a distance matrix and returns a cost-to-go function.

We'll use the algorithm described above.

The minimization step is vectorized to make it faster.

```python
def bellman(J, Q):
    return np.min(Q + J, axis=1)


def compute_cost_to_go(Q):
    num_nodes = Q.shape[0]
    J = np.zeros(num_nodes)      # Initial guess
    max_iter = 500
    i = 0

    while i < max_iter:
        next_J = bellman(J, Q)
        if np.allclose(next_J, J):
            break
        else:
            J[:] = next_J    # Copy contents of next_J to J
            i += 1

    return(J)
```

We used np.allclose() rather than testing exact equality because we are dealing with floating point numbers now.

Finally, here's a function that uses the cost-to-go function to obtain the optimal path (and its cost).

```python
def print_best_path(J, Q):
    sum_costs = 0
    current_node = 0
    while current_node != destination_node:
        print(current_node)
        # Move to the next node and increment costs
        next_node = np.argmin(Q[current_node, :] + J)
        sum_costs += Q[current_node, next_node]
        current_node = next_node

    print(destination_node)
```

```
    print('Cost: ', sum_costs)
```

Okay, now we have the necessary functions, let's call them to do the job we were assigned.

```
Q = map_graph_to_distance_matrix('graph.txt')
J = compute_cost_to_go(Q)
print_best_path(J, Q)
```

```
0
8
11
18
23
33
41
53
56
57
60
67
70
73
76
85
87
88
93
94
96
97
98
99
Cost:  160.55000000000007
```

The total cost of the path should agree with $J[0]$ so let's check this.

```
J[0]
```

```
160.55
```

Part XI

Modeling in Higher Dimensions

THE PERRON-FROBENIUS THEOREM

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

```
!pip install quantecon
```

In this lecture we will begin with the foundational concepts in spectral theory.

Then we will explore the Perron-Frobenius theorem and connect it to applications in Markov chains and networks.

We will use the following imports:

```python
import numpy as np
from numpy.linalg import eig
import scipy as sp
import quantecon as qe
```

```
OMP: Info #276: omp_set_nested routine deprecated, please use omp_set_max_active_
↪levels instead.
```

38.1 Nonnegative matrices

Often, in economics, the matrix that we are dealing with is nonnegative.

Nonnegative matrices have several special and useful properties.

In this section we will discuss some of them —in particular, the connection between nonnegativity and eigenvalues.

An $n \times m$ matrix A is called **nonnegative** if every element of A is nonnegative, i.e., $a_{ij} \geq 0$ for every i, j.

We denote this as $A \geq 0$.

38.1.1 Irreducible matrices

We introduced irreducible matrices in the *Markov chain lecture*.

Here we generalize this concept:

Let a_{ij}^k be element (i, j) of A^k.

An $n \times n$ nonnegative matrix A is called irreducible if $A + A^2 + A^3 + \cdots \gg 0$, where $\gg 0$ indicates that every element in A is strictly positive.

In other words, for each i, j with $1 \le i, j \le n$, there exists a $k \ge 0$ such that $a_{ij}^k > 0$.

Example

Here are some examples to illustrate this further:

$$A = \begin{bmatrix} 0.5 & 0.1 \\ 0.2 & 0.2 \end{bmatrix}$$

A is irreducible since $a_{ij} > 0$ for all (i, j).

$$B = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad B^2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

B is irreducible since $B + B^2$ is a matrix of ones.

$$C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

C is not irreducible since $C^k = C$ for all $k \ge 0$ and thus $c_{12}^k, c_{21}^k = 0$ for all $k \ge 0$.

38.1.2 Left eigenvectors

Recall that we previously discussed eigenvectors in *Eigenvalues and Eigenvectors*.

In particular, λ is an eigenvalue of A and v is an eigenvector of A if v is nonzero and satisfy

$$Av = \lambda v.$$

In this section we introduce left eigenvectors.

To avoid confusion, what we previously referred to as "eigenvectors" will be called "right eigenvectors".

Left eigenvectors will play important roles in what follows, including that of stochastic steady states for dynamic models under a Markov assumption.

A vector w is called a left eigenvector of A if w is a right eigenvector of $A^\top$.

In other words, if w is a left eigenvector of matrix A, then $A^\top w = \lambda w$, where λ is the eigenvalue associated with the left eigenvector v.

This hints at how to compute left eigenvectors

```python
A = np.array([[3, 2],
              [1, 4]])

# Compute eigenvalues and right eigenvectors
λ, v = eig(A)

# Compute eigenvalues and left eigenvectors
λ, w = eig(A.T)

# Keep 5 decimals
np.set_printoptions(precision=5)

print(f"The eigenvalues of A are:\n {λ}\n")
print(f"The corresponding right eigenvectors are: \n {v[:,0]} and {-v[:,1]}\n")
print(f"The corresponding left eigenvectors are: \n {w[:,0]} and {-w[:,1]}\n")
```

```
The eigenvalues of A are:
 [2. 5.]

The corresponding right eigenvectors are:
 [-0.89443  0.44721] and [0.70711 0.70711]

The corresponding left eigenvectors are:
 [-0.70711  0.70711] and [0.44721 0.89443]
```

We can also use `scipy.linalg.eig` with argument `left=True` to find left eigenvectors directly

```python
eigenvals, ε, e = sp.linalg.eig(A, left=True)

print(f"The eigenvalues of A are:\n {eigenvals.real}\n")
print(f"The corresponding right eigenvectors are: \n {e[:,0]} and {-e[:,1]}\n")
print(f"The corresponding left eigenvectors are: \n {ε[:,0]} and {-ε[:,1]}\n")
```

```
The eigenvalues of A are:
 [2. 5.]

The corresponding right eigenvectors are:
 [-0.89443  0.44721] and [0.70711 0.70711]
```

```
The corresponding left eigenvectors are:
 [-0.70711  0.70711] and [0.44721 0.89443]
```

The eigenvalues are the same while the eigenvectors themselves are different.

(Also note that we are taking the nonnegative value of the eigenvector of *dominant eigenvalue*, this is because `eig` automatically normalizes the eigenvectors.)

We can then take transpose to obtain $A^\top w = \lambda w$ and obtain $w^\top A = \lambda w^\top$.

This is a more common expression and where the name left eigenvectors originates.

38.1.3 The Perron-Frobenius theorem

For a square nonnegative matrix A, the behavior of A^k as $k \to \infty$ is controlled by the eigenvalue with the largest absolute value, often called the **dominant eigenvalue**.

For any such matrix A, the Perron-Frobenius theorem characterizes certain properties of the dominant eigenvalue and its corresponding eigenvector.

Theorem (Perron-Frobenius Theorem)

If a matrix $A \geq 0$ then,

1. the dominant eigenvalue of A, $r(A)$, is real-valued and nonnegative.

2. for any other eigenvalue (possibly complex) λ of A, $|\lambda| \leq r(A)$.

3. we can find a nonnegative and nonzero eigenvector v such that $Av = r(A)v$.

Moreover if A is also irreducible then,

4. the eigenvector v associated with the eigenvalue $r(A)$ is strictly positive.

5. there exists no other positive eigenvector v (except scalar multiples of v) associated with $r(A)$.

(More of the Perron-Frobenius theorem about primitive matrices will be introduced *below*.)

(This is a relatively simple version of the theorem —for more details see here).

We will see applications of the theorem below.

Let's build our intuition for the theorem using a simple example we have seen *before*.

Now let's consider examples for each case.

Example: irreducible matrix

Consider the following irreducible matrix A:

```
A = np.array([[0, 1, 0],
              [.5, 0, .5],
              [0, 1, 0]])
```

We can compute the dominant eigenvalue and the corresponding eigenvector

```
eig(A)
```

```
EigResult(eigenvalues=array([-1.00000e+00, -3.30139e-18,  1.00000e+00]),↵
 ↪eigenvectors=array([[ 5.77350e-01,  7.07107e-01,  5.77350e-01],
        [-5.77350e-01,  2.95712e-18,  5.77350e-01],
        [ 5.77350e-01, -7.07107e-01,  5.77350e-01]]))
```

Now we can see the claims of the Perron-Frobenius theorem holds for the irreducible matrix A:

1. The dominant eigenvalue is real-valued and non-negative.

2. All other eigenvalues have absolute values less than or equal to the dominant eigenvalue.

3. A non-negative and nonzero eigenvector is associated with the dominant eigenvalue.

4. As the matrix is irreducible, the eigenvector associated with the dominant eigenvalue is strictly positive.

5. There exists no other positive eigenvector associated with the dominant eigenvalue.

38.1.4 Primitive matrices

We know that in real world situations it's hard for a matrix to be everywhere positive (although they have nice properties).

The primitive matrices, however, can still give us helpful properties with looser definitions.

Let A be a square nonnegative matrix and let A^k be the kth power of A.

A matrix is called **primitive** if there exists a $k \in \mathbb{N}$ such that A^k is everywhere positive.

Example

Recall the examples given in irreducible matrices:

$$A = \begin{bmatrix} 0.5 & 0.1 \\ 0.2 & 0.2 \end{bmatrix}$$

A here is also a primitive matrix since A^k is everywhere nonnegative for $k \in \mathbb{N}$.

$$B = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad B^2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

B is irreducible but not primitive since there are always zeros in either principal diagonal or secondary diagonal.

We can see that if a matrix is primitive, then it implies the matrix is irreducible but not vice versa.

Now let's step back to the primitive matrices part of the Perron-Frobenius theorem

---

Theorem (Continous of Perron-Frobenius Theorem)

If A is primitive then,

6. the inequality $|\lambda| \le r(A)$ is **strict** for all eigenvalues λ of A distinct from $r(A)$, and

7. with v and w normalized so that the inner product of w and $v = 1$, we have $r(A)^{-m}A^m$ converges to $vw^\top$ when $m \to \infty$. The matrix $vw^\top$ is called the **Perron projection** of A.

Example 1: primitive matrix

Consider the following primitive matrix B:

```
B = np.array([[0, 1, 1],
              [1, 0, 1],
              [1, 1, 0]])

np.linalg.matrix_power(B, 2)
```

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

We compute the dominant eigenvalue and the corresponding eigenvector

```
eig(B)
```

```
EigResult(eigenvalues=array([-1.,  2., -1.]), eigenvectors=array([[-0.8165 ,  0.
 ↪57735, -0.2788 ],
       [ 0.40825,  0.57735, -0.52521],
       [ 0.40825,  0.57735,  0.80401]]))
```

Now let's give some examples to see if the claims of the Perron-Frobenius theorem hold for the primitive matrix B:

1. The dominant eigenvalue is real-valued and non-negative.

2. All other eigenvalues have absolute values strictly less than the dominant eigenvalue.

3. A non-negative and nonzero eigenvector is associated with the dominant eigenvalue.

4. The eigenvector associated with the dominant eigenvalue is strictly positive.

5. There exists no other positive eigenvector associated with the dominant eigenvalue.

6. The inequality $|\lambda| < r(B)$ holds for all eigenvalues λ of B distinct from the dominant eigenvalue.

Furthermore, we can verify the convergence property (7) of the theorem on the following examples:

```python
def compute_perron_projection(M):

    eigval, v = eig(M)
    eigval, w = eig(M.T)


    r = np.max(eigval)


    # Find the index of the dominant (Perron) eigenvalue
    i = np.argmax(eigval)


    # Get the Perron eigenvectors
    v_P = v[:, i].reshape(-1, 1)
    w_P = w[:, i].reshape(-1, 1)


    # Normalize the left and right eigenvectors
    norm_factor = w_P.T @ v_P
    v_norm = v_P / norm_factor


    # Compute the Perron projection matrix
    P = v_norm @ w_P.T
    return P, r

def check_convergence(M):
    P, r = compute_perron_projection(M)
    print("Perron projection:")
    print(P)


    # Define a list of values for n
    n_list = [1, 10, 100, 1000, 10000]


    for n in n_list:
```

```python
        # Compute (A/r)^n
        M_n = np.linalg.matrix_power(M/r, n)

        # Compute the difference between A^n / r^n and the Perron projection
        diff = np.abs(M_n - P)

        # Calculate the norm of the difference matrix
        diff_norm = np.linalg.norm(diff, 'fro')
        print(f"n = {n}, error = {diff_norm:.10f}")


A1 = np.array([[1, 2],
               [1, 4]])

A2 = np.array([[0, 1, 1],
               [1, 0, 1],
               [1, 1, 0]])

A3 = np.array([[0.971, 0.029, 0.1, 1],
               [0.145, 0.778, 0.077, 0.59],
               [0.1, 0.508, 0.492, 1.12],
               [0.2, 0.8, 0.71, 0.95]])

for M in A1, A2, A3:
    print("Matrix:")
    print(M)
    check_convergence(M)
    print()
    print("-"*36)
    print()
```

```
Matrix:
[[1 2]
 [1 4]]
Perron projection:
[[0.1362  0.48507]
 [0.24254 0.8638 ]]
n = 1, error = 0.0989045731
n = 10, error = 0.0000000001
n = 100, error = 0.0000000000
n = 1000, error = 0.0000000000
n = 10000, error = 0.0000000000
```

```
------------------------------------

Matrix:
[[0 1 1]
 [1 0 1]
 [1 1 0]]
Perron projection:
[[0.33333 0.33333 0.33333]
 [0.33333 0.33333 0.33333]
 [0.33333 0.33333 0.33333]]
n = 1, error = 0.7071067812
n = 10, error = 0.0013810679
n = 100, error = 0.0000000000
n = 1000, error = 0.0000000000
n = 10000, error = 0.0000000000


------------------------------------

Matrix:
[[0.971 0.029 0.1   1.   ]
 [0.145 0.778 0.077 0.59 ]
 [0.1   0.508 0.492 1.12 ]
 [0.2   0.8   0.71  0.95 ]]
Perron projection:
[[0.12506 0.31949 0.20233 0.43341]
 [0.07714 0.19707 0.1248  0.26735]
 [0.12158 0.31058 0.19669 0.42133]
 [0.13885 0.3547  0.22463 0.48118]]
n = 1, error = 0.5361031549
n = 10, error = 0.0000434043
n = 100, error = 0.0000000000
n = 1000, error = 0.0000000000
n = 10000, error = 0.0000000000


------------------------------------
```

The convergence is not observed in cases of non-primitive matrices.

Let's go through an example

```
B = np.array([[0, 1, 1],
              [1, 0, 0],
```

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

# This shows that the matrix is not primitive
print("Matrix:")
print(B)
print("100th power of matrix B:")
print(np.linalg.matrix_power(B, 100))


check_convergence(B)
```

```
Matrix:
[[0 1 1]
 [1 0 0]
 [1 0 0]]
100th power of matrix B:
[[1125899906842624                0                0]
 [               0  562949953421312  562949953421312]
 [               0  562949953421312  562949953421312]]
Perron projection:
[[0.5     0.35355 0.35355]
 [0.35355 0.25    0.25    ]
 [0.35355 0.25    0.25    ]]
n = 1, error = 1.0000000000
n = 10, error = 1.0000000000
n = 100, error = 1.0000000000
n = 1000, error = 1.0000000000
n = 10000, error = 1.0000000000
```

The result shows that the matrix is not primitive as it is not everywhere positive.

These examples show how the Perron-Frobenius theorem relates to the eigenvalues and eigenvectors of positive matrices and the convergence of the power of matrices.

In fact we have already seen the theorem in action before in *the Markov chain lecture*.

Example 2: connection to Markov chains

We are now prepared to bridge the languages spoken in the two lectures.

A primitive matrix is both irreducible and aperiodic.

So Perron-Frobenius theorem explains why both *Imam and Temple matrix* and Hamilton matrix converge to a stationary distribution, which is the Perron projection of the two matrices

```
P = np.array([[0.68, 0.12, 0.20],
              [0.50, 0.24, 0.26],
              [0.36, 0.18, 0.46]])

print(compute_perron_projection(P)[0])
```

```
[[0.56146 0.15565 0.28289]
 [0.56146 0.15565 0.28289]
 [0.56146 0.15565 0.28289]]
```

```
mc = qe.MarkovChain(P)
ψ_star = mc.stationary_distributions[0]
ψ_star
```

```
array([0.56146, 0.15565, 0.28289])
```

```
P_hamilton = np.array([[0.971, 0.029, 0.000],
                       [0.145, 0.778, 0.077],
                       [0.000, 0.508, 0.492]])

print(compute_perron_projection(P_hamilton)[0])
```

```
[[0.8128  0.16256 0.02464]
 [0.8128  0.16256 0.02464]
 [0.8128  0.16256 0.02464]]
```

```
mc = qe.MarkovChain(P_hamilton)
ψ_star = mc.stationary_distributions[0]
ψ_star
```

```
array([0.8128 , 0.16256, 0.02464])
```

We can also verify other properties hinted by Perron-Frobenius in these stochastic matrices.

Another example is the relationship between convergence gap and convergence rate.

In the *exercise*, we stated that the convergence rate is determined by the spectral gap, the difference between the largest and the second largest eigenvalue.

This can be proven using what we have learned here.

Please note that we use $\mathbb{1}$ for a vector of ones in this lecture.

38.1. Nonnegative matrices

With Markov model M with state space S and transition matrix P, we can write P^t as

$$P^t = \sum_{i=1}^{n-1} \lambda_i^t v_i w_i^\top + \mathbb{1}\psi^*,$$

This is proven in [Sargent and Stachurski, 2023] and a nice discussion can be found here.

In this formula λ_i is an eigenvalue of P with corresponding right and left eigenvectors v_i and w_i .

Premultiplying P^t by arbitrary $\psi \in \mathcal{D}(S)$ and rearranging now gives

$$\psi P^t - \psi^* = \sum_{i=1}^{n-1} \lambda_i^t \psi v_i w_i^\top$$

Recall that eigenvalues are ordered from smallest to largest from $i = 1...n$.

As we have seen, the largest eigenvalue for a primitive stochastic matrix is one.

This can be proven using Gershgorin Circle Theorem, but it is out of the scope of this lecture.

So by the statement (6) of Perron-Frobenius theorem, $\lambda_i < 1$ for all $i < n$, and $\lambda_n = 1$ when P is primitive.

Hence, after taking the Euclidean norm deviation, we obtain

$$\|\psi P^t - \psi^*\| = O\left(\eta^t\right) \quad \text{where} \quad \eta := |\lambda_{n-1}| < 1$$

Thus, the rate of convergence is governed by the modulus of the second largest eigenvalue.

38.2 Exercises

Exercise 38.2.1 (Leontief's Input-Output Model)

Wassily Leontief developed a model of an economy with n sectors producing n different commodities representing the interdependencies of different sectors of an economy.

Under this model some of the output is consumed internally by the industries and the rest is consumed by external consumers.

We define a simple model with 3 sectors - agriculture, industry, and service.

The following table describes how output is distributed within the economy:

	Total output	Agriculture	Industry	Service	Consumer
Agriculture	x_1	$0.3x_1$	$0.2x_2$	$0.3x_3$	4
Industry	x_2	$0.2x_1$	$0.4x_2$	$0.3x_3$	5
Service	x_3	$0.2x_1$	$0.5x_2$	$0.1x_3$	12

The first row depicts how agriculture's total output x_1 is distributed

- $0.3x_1$ is used as inputs within agriculture itself,

- $0.2x_2$ is used as inputs by the industry sector to produce x_2 units,

- $0.3x_3$ is used as inputs by the service sector to produce x_3 units and

- 4 units is the external demand by consumers.

We can transform this into a system of linear equations for the 3 sectors as given below:

$$x_1 = 0.3x_1 + 0.2x_2 + 0.3x_3 + 4$$
$$x_2 = 0.2x_1 + 0.4x_2 + 0.3x_3 + 5$$
$$x_3 = 0.2x_1 + 0.5x_2 + 0.1x_3 + 12$$

This can be transformed into the matrix equation $x = Ax + d$ where

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \quad A = \begin{bmatrix} 0.3 & 0.2 & 0.3 \\ 0.2 & 0.4 & 0.3 \\ 0.2 & 0.5 & 0.1 \end{bmatrix} \text{ and } d = \begin{bmatrix} 4 \\ 5 \\ 12 \end{bmatrix}$$

The solution x^* is given by the equation $x^* = (I - A)^{-1}d$

1. Since A is a nonnegative irreducible matrix, find the Perron-Frobenius eigenvalue of A.

2. Use the *Neumann Series Lemma* to find the solution x^* if it exists.

Solution to Exercise 38.2.1 (Leontief's Input-Output Model)

```python
A = np.array([[0.3, 0.2, 0.3],
              [0.2, 0.4, 0.3],
              [0.2, 0.5, 0.1]])

evals, evecs = eig(A)

r = max(abs(λ) for λ in evals)   #dominant eigenvalue/spectral radius
print(r)
```

```
0.8444086477164554
```

Since we have $r(A) < 1$ we can thus find the solution using the Neumann Series Lemma.

```python
I = np.identity(3)
B = I - A
```

```
d = np.array([4, 5, 12])
d.shape = (3,1)

B_inv = np.linalg.inv(B)
x_star = B_inv @ d
print(x_star)
```

```
[[38.30189]
 [44.33962]
 [46.47799]]
```

INPUT-OUTPUT MODELS

39.1 Overview

This lecture requires the following imports and installs before we proceed.

```
!pip install quantecon_book_networks
!pip install quantecon
!pip install pandas-datareader
```

```python
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import quantecon_book_networks
import quantecon_book_networks.input_output as qbn_io
import quantecon_book_networks.plotting as qbn_plt
import quantecon_book_networks.data as qbn_data
import matplotlib as mpl
from matplotlib.patches import Polygon

quantecon_book_networks.config("matplotlib")
mpl.rcParams.update(mpl.rcParamsDefault)
```

The following figure illustrates a network of linkages among 15 sectors obtained from the US Bureau of Economic Analysis's 2021 Input-Output Accounts Data.
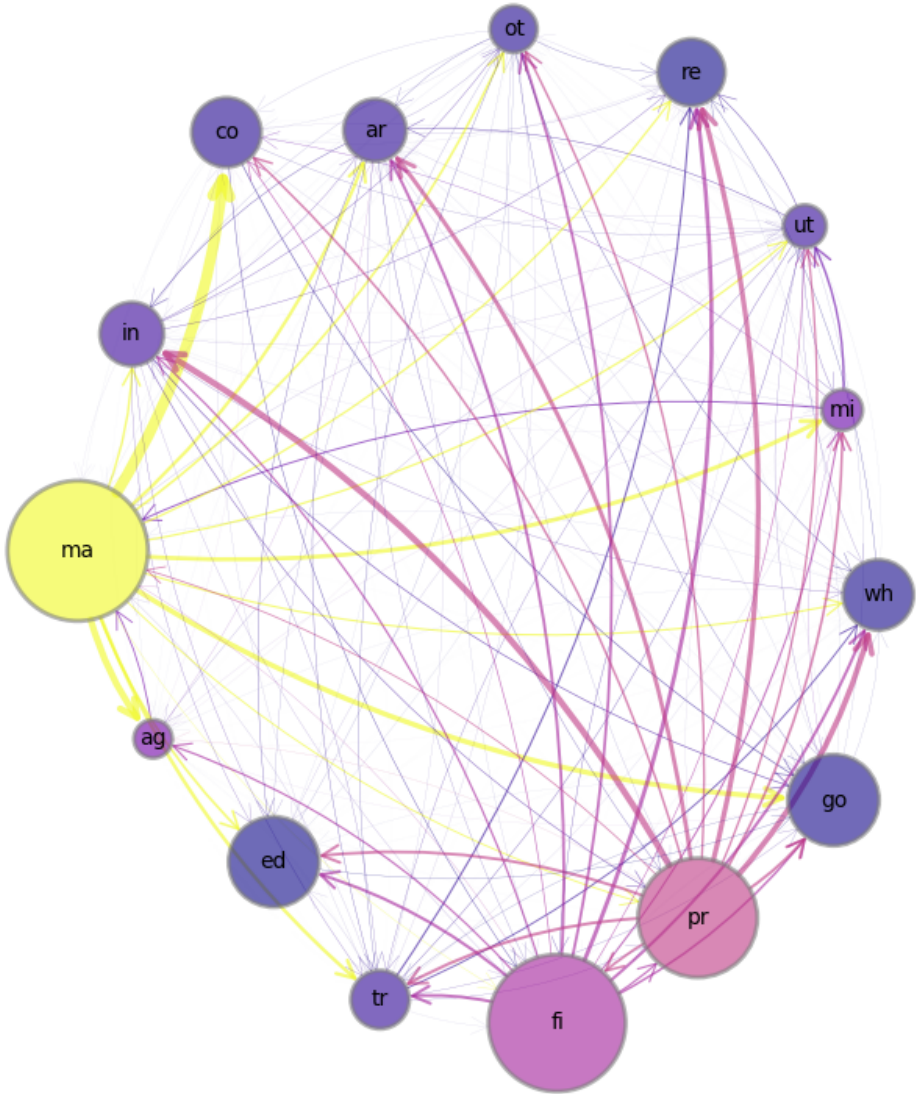
Fig. 39.1: US 15 sector production network

Label	Sector	Label	Sector	Label	Sector
ag	Agriculture	wh	Wholesale	pr	Professional Services
mi	Mining	re	Retail	ed	Education & Health
ut	Utilities	tr	Transportation	ar	Arts & Entertainment
co	Construction	in	Information	ot	Other Services (exc govt)
ma	Manufacturing	fi	Finance	go	Government

An arrow from i to j means that some of sector i's output serves as an input to production of sector j.

Economies are characterised by many such links.

A basic framework for their analysis is Leontief's input-output model.

After introducing the input-output model, we describe some of its connections to *linear programming lecture*.

39.2 Input-output analysis

Let

- x_0 be the amount of a single exogenous input to production, say labor

- $x_j, j = 1, \ldots n$ be the gross output of final good j

- $d_j, j = 1, \ldots n$ be the net output of final good j that is available for final consumption

- z_{ij} be the quantity of good i allocated to be an input to producing good j for $i = 1, \ldots n, j = 1, \ldots n$

- z_{0j} be the quantity of labor allocated to producing good j.

- a_{ij} be the number of units of good i required to produce one unit of good j, $i = 0, \ldots, n, j = 1, \ldots n$.

- $w > 0$ be an exogenous wage of labor, denominated in dollars per unit of labor

- p be an $n \times 1$ vector of prices of produced goods $i = 1, \ldots, n$.

The technology for producing good $j \in \{1, \ldots, n\}$ is described by the **Leontief** function

$$x_j = \min_{i \in \{0, \ldots, n\}} \left(\frac{z_{ij}}{a_{ij}} \right)$$

39.2.1 Two goods

To illustrate, we begin by setting $n = 2$ and formulating the following network.

```
/var/folders/zy/dpt7pvt11fvcm_yxjd7kmv880000gn/T/ipykernel_52706/1540025235.py:22:
↪DeprecationWarning: `alltrue` is deprecated as of NumPy 1.25.0, and will be
↪removed in NumPy 2.0. Please use `all` instead.
  nx.draw_networkx_edges(G, pos=pos, edgelist=edges2,
```
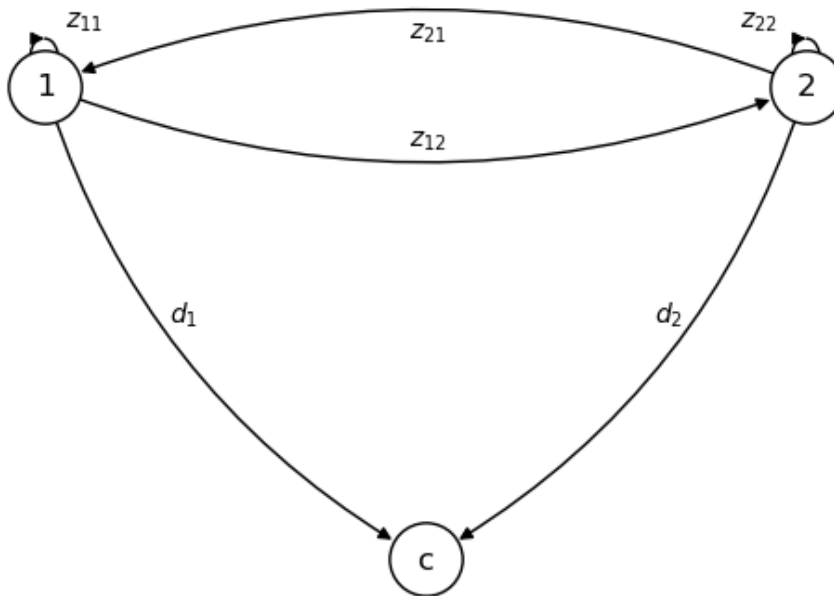

Feasible allocations must satisfy

$$(1 - a_{11})x_1 - a_{12}x_2 \geq d_1$$
$$-a_{21}x_1 + (1 - a_{22})x_2 \geq d_2$$
$$a_{01}x_1 + a_{02}x_2 \leq x_0$$

This can be graphically represented as follows.
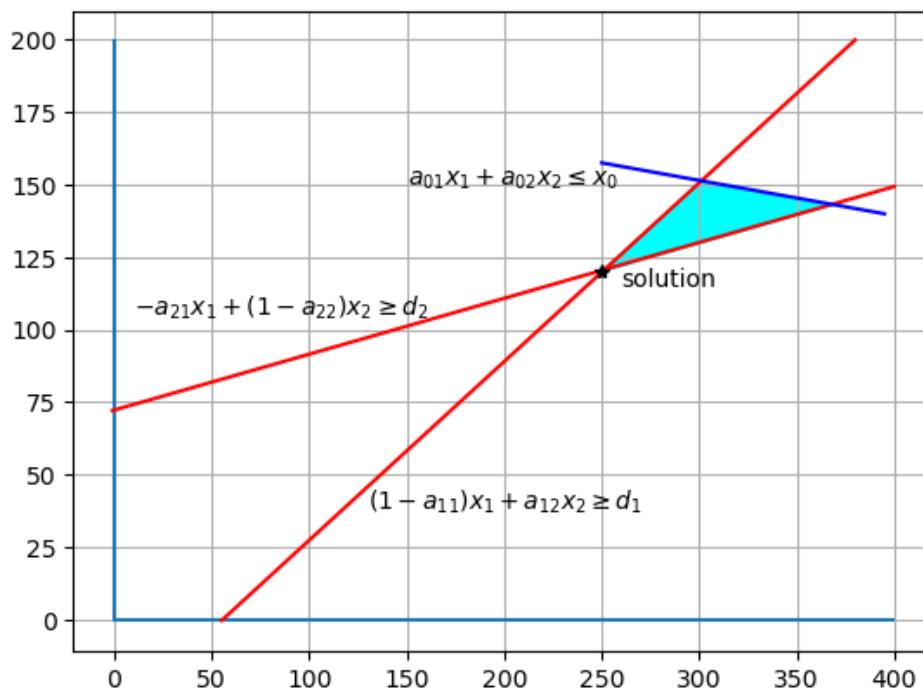
More generally, constraints on production are

$$(I - A)x \geq d$$
$$a_0^\top x \leq x_0$$

(39.1)

where A is the $n \times n$ matrix with typical element a_{ij} and $a_0^\top = \begin{bmatrix} a_{01} & \cdots & a_{0n} \end{bmatrix}$.

If we solve the first block of equations of (**??**) for gross output x we get

$$x = (I - A)^{-1}d \equiv Ld$$

(39.2)

where the matrix $L = (I - A)^{-1}$ is sometimes called a **Leontief Inverse**.

To assure that the solution X of (**??**) is a positive vector, the following **Hawkins-Simon conditions** suffice:

$$\det(I - A) > 0 \text{ and}$$
$$(I - A)_{ij} > 0 \text{ for all } i = j$$

Example 39.2.1

For example a two-good economy described by

$$A = \begin{bmatrix} 0.1 & 40 \\ 0.01 & 0 \end{bmatrix} \text{ and } d = \begin{bmatrix} 50 \\ 2 \end{bmatrix}$$

(39.3)

```
A = np.array([[0.1, 40],
              [0.01, 0]])
d = np.array([50, 2]).reshape((2, 1))
```

```
I = np.identity(2)
B = I - A
B
```

```
array([[ 9.e-01, -4.e+01],
       [-1.e-02,  1.e+00]])
```

Let's check the **Hawkins-Simon conditions**

```
np.linalg.det(B) > 0 # checking Hawkins-Simon conditions
```

```
True
```

Now, let's compute the **Leontief inverse** matrix

```
L = np.linalg.inv(B) # obtaining Leontief inverse matrix
L
```

```
array([[2.0e+00, 8.0e+01],
       [2.0e-02, 1.8e+00]])
```

```
x = L @ d   # solving for gross output
x
```

```
array([[260. ],
       [  4.6]])
```

39.3 Production possibility frontier

The second equation of (**??**) can be written

$$a_0^\top x = x_0$$

or

$$A_0^\top d = x_0 \tag{39.4}$$

where

$$A_0^\top = a_0^\top (I - A)^{-1}$$

For $i \in \{1, \ldots, n\}$, the ith component of A_0 is the amount of labor that is required to produce one unit of final output of good i.

Equation (**??**) sweeps out a **production possibility frontier** of final consumption bundles d that can be produced with exogenous labor input x_0.

Example 39.3.1

Consider the example in (**??**).

Suppose we are now given

$$a_0^\top = \begin{bmatrix} 4 & 100 \end{bmatrix}$$

Then we can find $A_0^\top$ by

```
a0 = np.array([4, 100])
A0 = a0 @ L
A0
```

```
array([ 10., 500.])
```

Thus, the production possibility frontier for this economy is

$$10d_1 + 500d_2 = x_0$$

39.4 Prices

[Dorfman *et al.*, 1958] argue that relative prices of the n produced goods must satisfy

$$p_1 = a_{11}p_1 + a_{21}p_2 + a_{01}w$$
$$p_2 = a_{12}p_1 + a_{22}p_2 + a_{02}w$$

More generally,

$$p = A^\top p + a_0 w$$

which states that the price of each final good equals the total cost of production, which consists of costs of intermediate inputs $A^\top p$ plus costs of labor $a_0 w$.

This equation can be written as

$$(I - A^\top)p = a_0 w \tag{39.5}$$

which implies

$$p = (I - A^\top)^{-1} a_0 w$$

Notice how (**??**) with (**??**) forms a **conjugate pair** through the appearance of operators that are transposes of one another.

This connection surfaces again in a classic linear program and its dual.

39.5 Linear programs

A **primal** problem is

$$\min_x w a_0^\top x$$

subject to

$$(I - A)x \geq d$$

The associated **dual** problem is

$$\max_p p^\top d$$

subject to

$$(I - A)^\top p \leq a_0 w$$

The primal problem chooses a feasible production plan to minimize costs for delivering a pre-assigned vector of final goods consumption d.

The dual problem chooses prices to maximize the value of a pre-assigned vector of final goods d subject to prices covering costs of production.

By the strong duality theorem, optimal value of the primal and dual problems coincide:

$$w a_0^\top x^* = p^* d$$

where $*$'s denote optimal choices for the primal and dual problems.

The dual problem can be graphically represented as follows.
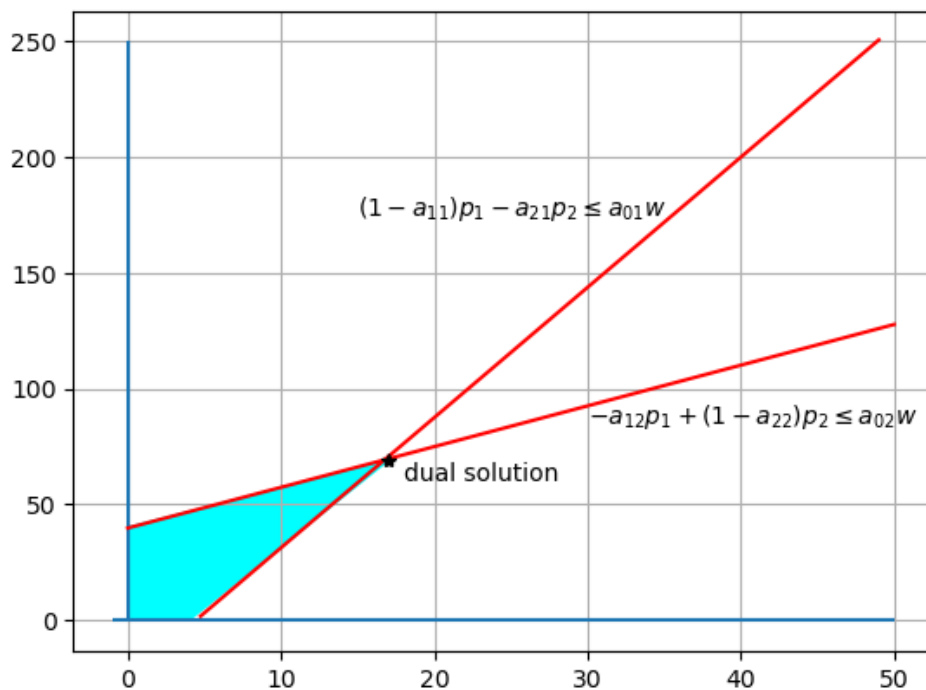
39.6 Leontief inverse

We have discussed that gross output x is given by (**??**), where L is called the Leontief Inverse.

Recall the *Neumann Series Lemma* which states that L exists if the spectral radius $r(A) < 1$.

In fact

$$L = \sum_{i=0}^{\infty} A^i$$

39.6.1 Demand shocks

Consider the impact of a demand shock Δd which shifts demand from d_0 to $d_1 = d_0 + \Delta d$.

Gross output shifts from $x_0 = Ld_0$ to $x_1 = Ld_1$.

If $r(A) < 1$ then a solution exists and

$$\Delta x = L\Delta d = \Delta d + A(\Delta d) + A^2(\Delta d) + \cdots$$

This illustrates that an element l_{ij} of L shows the total impact on sector i of a unit change in demand of good j.

39.7 Applications of graph theory

We can further study input-output networks through applications of *graph theory*.

An input-output network can be represented by a weighted directed graph induced by the adjacency matrix A.

The set of nodes $V = [n]$ is the list of sectors and the set of edges is given by

$$E = \{(i, j) \in V \times V : a_{ij} > 0\}$$

In Fig. **??** weights are indicated by the widths of the arrows, which are proportional to the corresponding input-output coefficients.

We can now use centrality measures to rank sectors and discuss their importance relative to the other sectors.

39.7.1 Eigenvector centrality

Eigenvector centrality of a node i is measured by

$$e_i = \frac{1}{r(A)} \sum_{1 \leq j \leq n} a_{ij} e_j$$

We plot a bar graph of hub-based eigenvector centrality for the sectors represented in Fig. **??**.
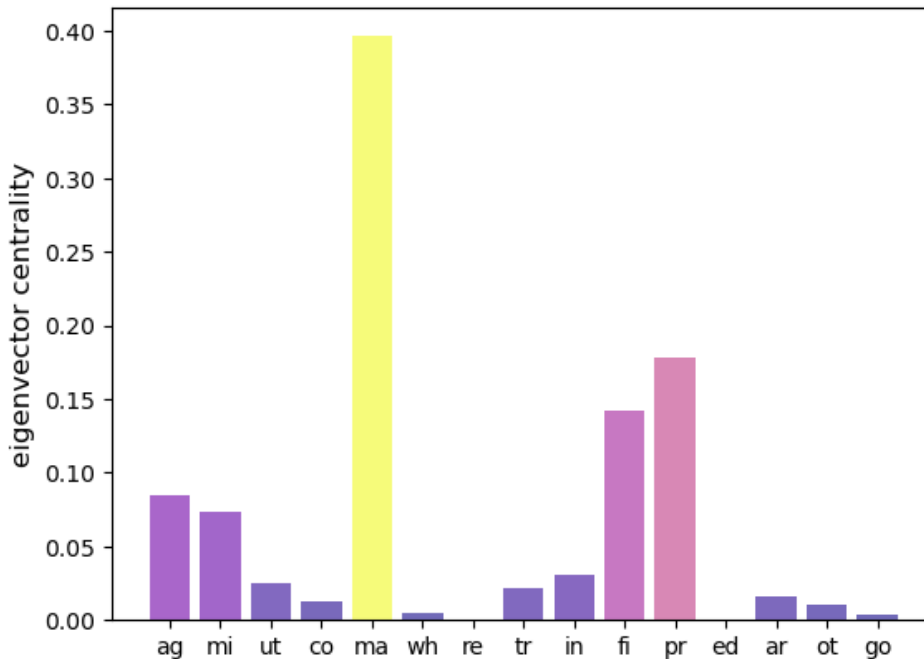
A higher measure indicates higher importance as a supplier.

As a result demand shocks in most sectors will significantly impact activity in sectors with high eigenvector centrality.

The above figure indicates that manufacturing is the most dominant sector in the US economy.

39.7.2 Output multipliers

Another way to rank sectors in input-output networks is via output multipliers.

The **output multiplier** of sector j denoted by μ_j is usually defined as the total sector-wide impact of a unit change of demand in sector j.

Earlier when disussing demand shocks we concluded that for $L = (l_{ij})$ the element l_{ij} represents the impact on sector i of a unit change in demand in sector j.

Thus,

$$\mu_j = \sum_{j=1}^{n} l_{ij}$$

This can be written as $\mu^\top = \mathbb{1}^\top L$ or

$$\mu^\top = \mathbb{1}^\top (I - A)^{-1}$$

Please note that here we use $\mathbb{1}$ to represent a vector of ones.

High ranking sectors within this measure are important buyers of intermediate goods.

A demand shock in such sectors will cause a large impact on the whole production network.

The following figure displays the output multipliers for the sectors represented in Fig. **??**.
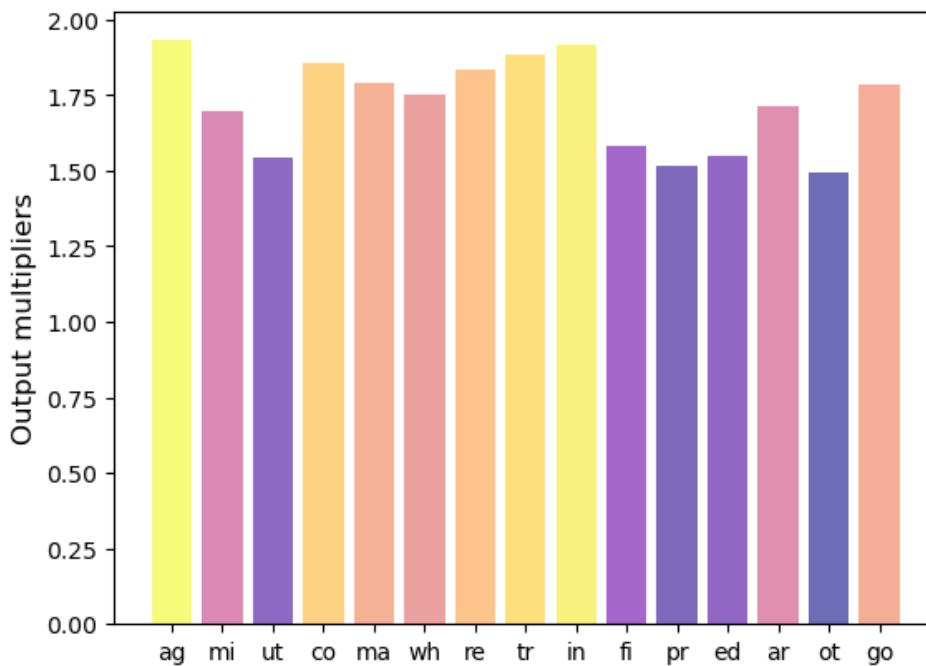
We observe that manufacturing and agriculture are highest ranking sectors.

39.8 Exercises

Exercise 39.8.1

[Dorfman *et al.*, 1958] Chapter 9 discusses an example with the following parameter settings:

$$A = \begin{bmatrix} 0.1 & 1.46 \\ 0.16 & 0.17 \end{bmatrix} \text{ and } a_0 = \begin{bmatrix} .04 & .33 \end{bmatrix}$$

$$x = \begin{bmatrix} 250 \\ 120 \end{bmatrix} \text{ and } x_0 = 50$$

$$d = \begin{bmatrix} 50 \\ 60 \end{bmatrix}$$

Describe how they infer the input-output coefficients in A and a_0 from the following hypothetical underlying "data" on agricultural and manufacturing industries:

$$z = \begin{bmatrix} 25 & 175 \\ 40 & 20 \end{bmatrix} \text{ and } z_0 = \begin{bmatrix} 10 & 40 \end{bmatrix}$$

where z_0 is a vector of labor services used in each industry.

Solution to Exercise 39.8.1

For each $i = 0, 1, 2$ and $j = 1, 2$

$$a_{ij} = \frac{z_{ij}}{x_j}$$

Exercise 39.8.2

Derive the production possibility frontier for the economy characterized in the previous exercise.

Solution to Exercise 39.8.2

```
A = np.array([[0.1, 1.46],
              [0.16, 0.17]])
a_0 = np.array([0.04, 0.33])
```

```
I = np.identity(2)
B = I - A
L = np.linalg.inv(B)
```

```
A_0 = a_0 @ L
A_0
```

```
array([0.16751071, 0.69224776])
```

Thus the production possibility frontier is given by

$$0.17d_1 + 0.69d_2 = 50$$

CHAPTER

FORTY

A LAKE MODEL OF EMPLOYMENT

40.1 Outline

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

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

40.2 The Lake model

This model is sometimes called the **lake model** because there are two pools of workers:

1. those who are currently employed.

2. those who are currently unemployed but are seeking employment.

The "flows" between the two lakes are as follows:

1. workers exit the labor market at rate d.

2. new workers enter the labor market at rate b.

3. employed workers separate from their jobs at rate α.

4. unemployed workers find jobs at rate λ.

The graph below illustrates the lake model.
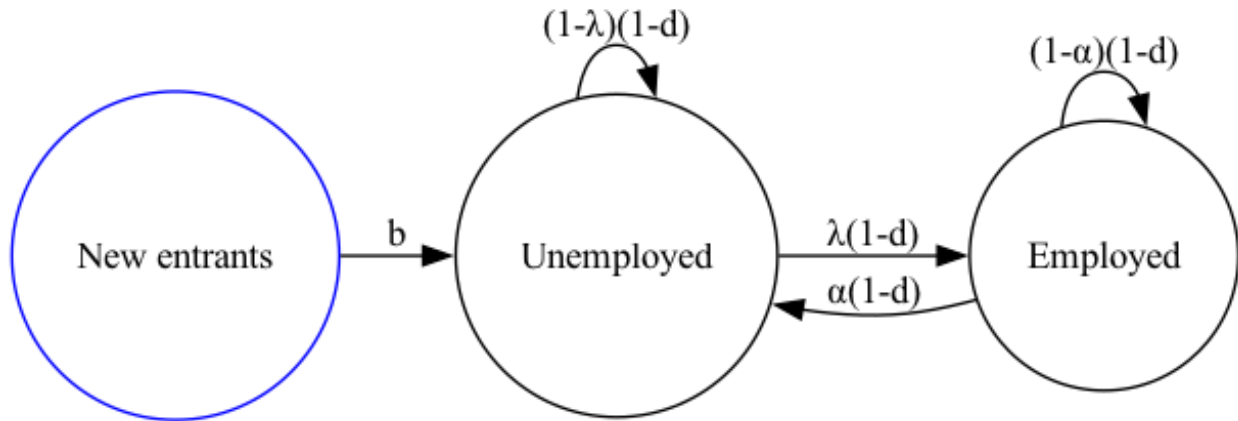
Fig. 40.1: An illustration of the lake model

40.3 Dynamics

Let e_t and u_t be the number of employed and unemployed workers at time t respectively.

The total population of workers is $n_t = e_t + u_t$.

The number of unemployed and employed workers thus evolves according to:

$$
\begin{aligned}
u_{t+1} &= (1-d)(1-\lambda)u_t + \alpha(1-d)e_t + bn_t \\
&= ((1-d)(1-\lambda) + b)u_t + (\alpha(1-d) + b)e_t \\
e_{t+1} &= (1-d)\lambda u_t + (1-\alpha)(1-d)e_t
\end{aligned}
\tag{40.1}
$$

We can arrange (**??**) as a linear system of equations in matrix form $x_{t+1} = Ax_t$ where

$$
x_{t+1} = \begin{bmatrix} u_{t+1} \\ e_{t+1} \end{bmatrix} \quad A = \begin{bmatrix} (1-d)(1-\lambda) + b & \alpha(1-d) + b \\ (1-d)\lambda & (1-\alpha)(1-d) \end{bmatrix} \quad \text{and} \quad x_t = \begin{bmatrix} u_t \\ e_t \end{bmatrix}.
$$

Suppose at $t = 0$ we have $x_0 = \begin{bmatrix} u_0 & e_0 \end{bmatrix}^\top$.

Then, $x_1 = Ax_0$, $x_2 = Ax_1 = A^2x_0$ and thus $x_t = A^tx_0$.

Thus the long-run outcomes of this system may depend on the initial condition x_0 and the matrix A.

We are interested in how u_t and e_t evolve over time.

What long-run unemployment rate and employment rate should we expect?

Do long-run outcomes depend on the initial values (u_0, e_o)?

40.3.1 Visualising the long-run outcomes

Let us first plot the time series of unemployment u_t, employment e_t, and labor force n_t.

```python
class LakeModel:
    """
    Solves the lake model and computes dynamics of the unemployment stocks and
    rates.

    Parameters:
    ------------
    λ : scalar
        The job finding rate for currently unemployed workers
    α : scalar
        The dismissal rate for currently employed workers
    b : scalar
        Entry rate into the labor force
    d : scalar
        Exit rate from the labor force

    """
    def __init__(self, λ=0.1, α=0.013, b=0.0124, d=0.00822):
        self.λ, self.α, self.b, self.d = λ, α, b, d

        λ, α, b, d = self.λ, self.α, self.b, self.d
        self.g = b - d
        g = self.g

        self.A = np.array([[(1-d)*(1-λ) + b,      α*(1-d) + b],
                           [        (1-d)*λ,      (1-α)*(1-d)]])


        self.ū = (1 + g - (1 - d) * (1 - α)) / (1 + g - (1 - d) * (1 - α) + (1 - d) *
    ↪λ)
        self.ē = 1 - self.ū


    def simulate_path(self, x0, T=1000):
        """
        Simulates the sequence of employment and unemployment

        Parameters
        ----------
        x0 : array
```

```
            Contains initial values (u0,e0)
        T : int
            Number of periods to simulate


        Returns
        ----------
        x : iterator
            Contains sequence of employment and unemployment rates


        """
        x0 = np.atleast_1d(x0)  # Recast as array just in case
        x_ts= np.zeros((2, T))
        x_ts[:, 0] = x0
        for t in range(1, T):
            x_ts[:, t] = self.A @ x_ts[:, t-1]
        return x_ts
```

```
lm = LakeModel()
e_0 = 0.92          # Initial employment
u_0 = 1 - e_0       # Initial unemployment, given initial n_0 = 1

lm = LakeModel()
T = 100           # Simulation length

x_0 = (u_0, e_0)
x_path = lm.simulate_path(x_0, T)

fig, axes = plt.subplots(3, 1, figsize=(10, 8))


axes[0].plot(x_path[0, :], lw=2)
axes[0].set_title('Unemployment')

axes[1].plot(x_path[1, :], lw=2)
axes[1].set_title('Employment')

axes[2].plot(x_path.sum(0), lw=2)
axes[2].set_title('Labor force')

for ax in axes:
    ax.grid()

plt.tight_layout()
```

```
plt.show()
```
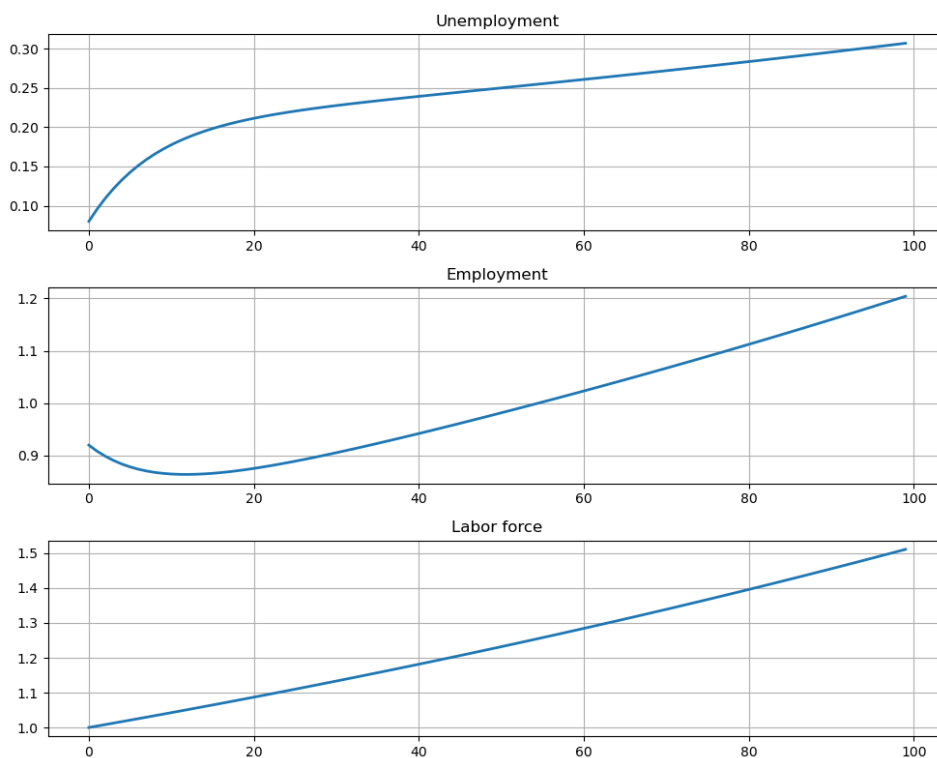

Not surprisingly, we observe that labor force n_t increases at a constant rate.

This coincides with the fact there is only one inflow source (new entrants pool) to unemployment and employment pools.

The inflow and outflow of labor market system is determined by constant exit rate and entry rate of labor market in the long run.

In detail, let $\mathbb{1} = [1, 1]^\top$ be a vector of ones.

Observe that

$$
\begin{aligned}
n_{t+1} &= u_{t+1} + e_{t+1} \\
&= \mathbb{1}^\top x_{t+1} \\
&= \mathbb{1}^\top A x_t \\
&= (1 + b - d)(u_t + e_t) \\
&= (1 + b - d)n_t.
\end{aligned}
$$

Hence, the growth rate of n_t is fixed at $1 + b - d$.

Moreover, the times series of unemployment and employment seems to grow at some stable rates in the long run.

40.3.2 The application of Perron-Frobenius theorem

Since by intuition if we consider unemployment pool and employment pool as a closed system, the growth should be similar to the labor force.

We next ask whether the long-run growth rates of e_t and u_t also dominated by $1 + b - d$ as labor force.

The answer will be clearer if we appeal to *Perron-Frobenius theorem*.

The importance of the Perron-Frobenius theorem stems from the fact that firstly in the real world most matrices we encounter are nonnegative matrices.

Secondly, many important models are simply linear iterative models that begin with an initial condition x_0 and then evolve recursively by the rule $x_{t+1} = Ax_t$ or in short $x_t = A^t x_0$.

This theorem helps characterise the dominant eigenvalue $r(A)$ which determines the behavior of this iterative process.

Dominant eigenvector

We now illustrate the power of the Perron-Frobenius theorem by showing how it helps us to analyze the lake model.

Since A is a nonnegative and irreducible matrix, the Perron-Frobenius theorem implies that:

- the spectral radius $r(A)$ is an eigenvalue of A, where

$$r(A) := \max\{|\lambda| : \lambda \text{ is an eigenvalue of } A\}$$

- any other eigenvalue λ in absolute value is strictly smaller than $r(A)$: $|\lambda| < r(A)$,

- there exist unique and everywhere positive right eigenvector ϕ (column vector) and left eigenvector ψ (row vector):

$$A\phi = r(A)\phi, \quad \psi A = r(A)\psi$$

- if further A is positive, then with $<\psi, \phi> = \psi\phi = 1$ we have

$$r(A)^{-t} A^t \to \phi\psi$$

The last statement implies that the magnitude of A^t is identical to the magnitude of $r(A)^t$ in the long run, where $r(A)$ can be considered as the dominant eigenvalue in this lecture.

Therefore, the magnitude $x_t = A^t x_0$ is also dominated by $r(A)^t$ in the long run.

Recall that the spectral radius is bounded by column sums: for $A \geq 0$, we have

$$\min_j \text{colsum}_j(A) \leq r(A) \leq \max_j \text{colsum}_j(A) \tag{40.2}$$

Note that $\text{colsum}_j(A) = 1 + b - d$ for $j = 1, 2$ and by (**??**) we can thus conclude that the dominant eigenvalue is $r(A) = 1 + b - d$.

Denote $g = b - d$ as the overall growth rate of the total labor force, so that $r(A) = 1 + g$.

The Perron-Frobenius implies that there is a unique positive eigenvector $\bar{x} = \begin{bmatrix} \bar{u} \\ \bar{e} \end{bmatrix}$ such that $A\bar{x} = r(A)\bar{x}$ and $\begin{bmatrix} 1 & 1 \end{bmatrix} \bar{x} = 1$:

$$
\begin{aligned}
\bar{u} &= \frac{b + \alpha(1-d)}{b + (\alpha + \lambda)(1-d)} \\
\bar{e} &= \frac{\lambda(1-d)}{b + (\alpha + \lambda)(1-d)}
\end{aligned}
\tag{40.3}
$$

Since $\bar{x}$ is the eigenvector corresponding to the dominant eigenvalue $r(A)$, we call $\bar{x}$ the dominant eigenvector.

This dominant eigenvector plays an important role in determining long-run outcomes as illustrated below.

```python
def plot_time_paths(lm, x0=None, T=1000, ax=None):
    """
    Plots the simulated time series.

    Parameters
    ----------
    lm : class
        Lake Model
    x0 : array
        Contains some different initial values.
    T : int
        Number of periods to simulate

    """

    if x0 is None:
        x0 = np.array([[5.0, 0.1]])

    ū, ē = lm.ū, lm.ē

    x0 = np.atleast_2d(x0)

    if ax is None:
        fig, ax = plt.subplots(figsize=(10, 8))
        # Plot line D
        s = 10
        ax.plot([0, s * ū], [0, s * ē], "k--", lw=1, label='set $D$')

        # Set the axes through the origin
        for spine in ["left", "bottom"]:
            ax.spines[spine].set_position("zero")
        for spine in ["right", "top"]:
            ax.spines[spine].set_color("none")
```

40.3. Dynamics

```python
        ax.set_xlim(-2, 6)
        ax.set_ylim(-2, 6)
        ax.set_xlabel("unemployed workforce")
        ax.set_ylabel("employed workforce")
        ax.set_xticks((0, 6))
        ax.set_yticks((0, 6))



        # Plot time series
        for x in x0:
            x_ts = lm.simulate_path(x0=x)

            ax.scatter(x_ts[0, :], x_ts[1, :], s=4,)

            u0, e0 = x
            ax.plot([u0], [e0], "ko", ms=2, alpha=0.6)
            ax.annotate(f'$x_0 = ({u0},{e0})$',
                        xy=(u0, e0),
                        xycoords="data",
                        xytext=(0, 20),
                        textcoords="offset points",
                        arrowprops=dict(arrowstyle = "->"))

        ax.plot([ū], [ē], "ko", ms=4, alpha=0.6)
        ax.annotate(r'$\bar{x}$',
                xy=(ū, ē),
                xycoords="data",
                xytext=(20, -20),
                textcoords="offset points",
                arrowprops=dict(arrowstyle = "->"))

        if ax is None:
            plt.show()
```

```python
lm = LakeModel(α=0.01, λ=0.1, d=0.02, b=0.025)
x0 = ((5.0, 0.1), (0.1, 4.0), (2.0, 1.0))
plot_time_paths(lm, x0=x0)
```
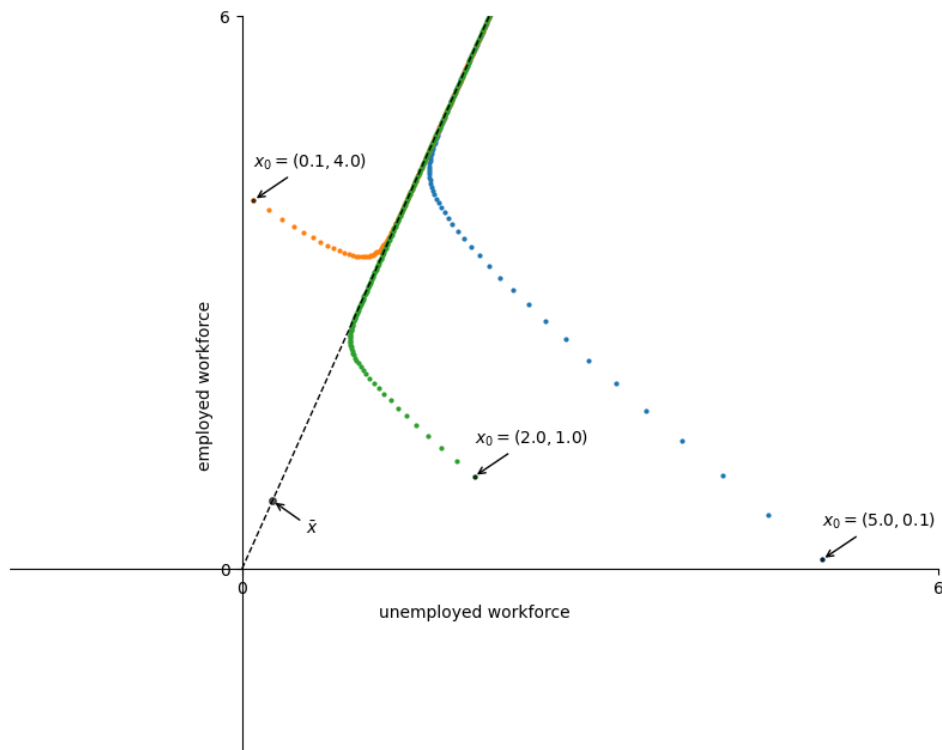
Since $\bar{x}$ is an eigenvector corresponding to the eigenvalue $r(A)$, all the vectors in the set $D := \{x \in \mathbb{R}^2 : x = \alpha\bar{x} \text{ for some } \alpha > 0\}$ are also eigenvectors corresponding to $r(A)$.

This set D is represented by a dashed line in the above figure.

The graph illustrates that for two distinct initial conditions x_0 the sequences of iterates $(A^t x_0)_{t \geq 0}$ move towards D over time.

This suggests that all such sequences share strong similarities in the long run, determined by the dominant eigenvector $\bar{x}$.

Negative growth rate

In the example illustrated above we considered parameters such that overall growth rate of the labor force $g > 0$.

Suppose now we are faced with a situation where the $g < 0$, i.e., negative growth in the labor force.

This means that $b - d < 0$, i.e., workers exit the market faster than they enter.

What would the behavior of the iterative sequence $x_{t+1} = Ax_t$ be now?

This is visualised below.

```
lm = LakeModel(α=0.01, λ=0.1, d=0.025, b=0.02)
plot_time_paths(lm, x0=x0)
```
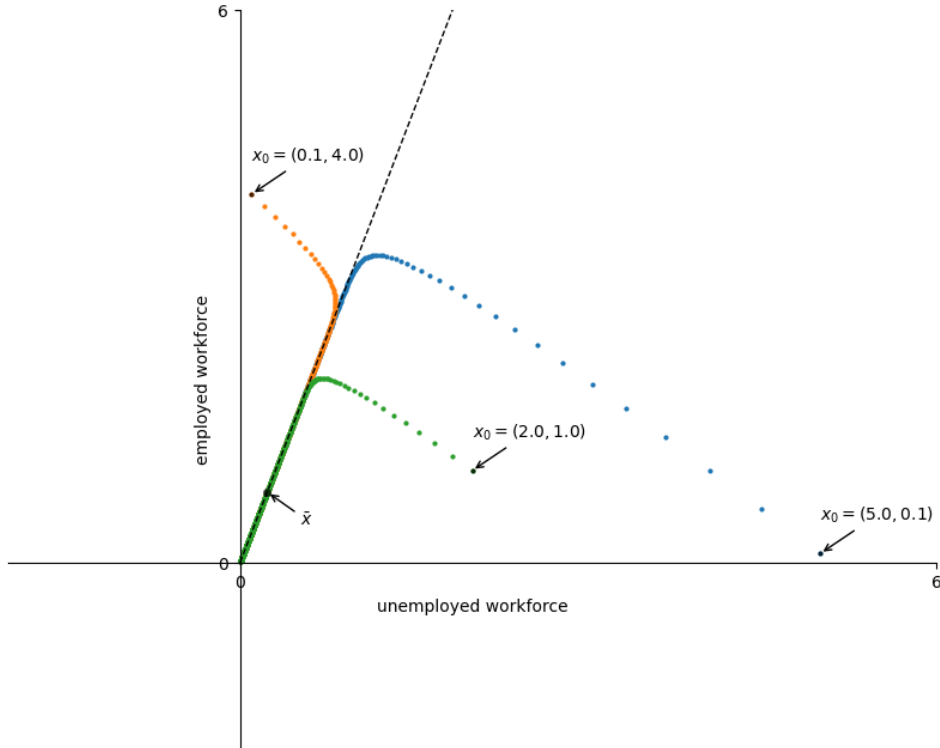
Thus, while the sequence of iterates still moves towards the dominant eigenvector $\bar{x}$, in this case they converge to the origin.

This is a result of the fact that $r(A) < 1$, which ensures that the iterative sequence $(A^t x_0)_{t \geq 0}$ will converge to some point, in this case to $(0, 0)$.

This leads us to the next result.

40.3.3 Properties

Since the column sums of A are $r(A) = 1$, the left eigenvector is $\mathbb{1}^\top = [1, 1]$.

Perron-Frobenius theory implies that

$$r(A)^{-t} A^t \approx \bar{x} \mathbb{1}^\top = \begin{bmatrix} \bar{u} & \bar{u} \\ \bar{e} & \bar{e} \end{bmatrix}.$$

As a result, for any $x_0 = (u_0, e_0)^\top$, we have

$$x_t = A^t x_0 \approx r(A)^t \begin{bmatrix} \bar{u} & \bar{u} \\ \bar{e} & \bar{e} \end{bmatrix} \begin{bmatrix} u_0 \\ e_0 \end{bmatrix}$$

$$= (1 + g)^t (u_0 + e_0) \begin{bmatrix} \bar{u} \\ \bar{e} \end{bmatrix}$$

$$= (1 + g)^t n_0 \bar{x}$$

$$= n_t \bar{x}.$$

as t is large enough.

We see that the growth of u_t and e_t also dominated by $r(A) = 1 + g$ in the long run: x_t grows along D as $r(A) > 1$ and converges to $(0, 0)$ as $r(A) < 1$.

Moreover, the long-run unemployment and employment are steady fractions of n_t.

The latter implies that $\bar{u}$ and $\bar{e}$ are long-run unemployment rate and employment rate, respectively.

In detail, we have the unemployment rates and employment rates: $x_t/n_t = A^t n_0/n_t \to \bar{x}$ as $t \to \infty$.

To illustrate the dynamics of the rates, let $\hat{A} := A/(1 + g)$ be the transition matrix of $r_t := x_t/n_t$.

The dynamics of the rates follow

$$r_{t+1} = \frac{x_{t+1}}{n_{t+1}} = \frac{x_{t+1}}{(1 + g)n_t} = \frac{Ax_t}{(1 + g)n_t} = \hat{A}\frac{x_t}{n_t} = \hat{A}r_t.$$

Observe that the column sums of $\hat{A}$ are all one so that $r(\hat{A}) = 1$.

One can check that $\bar{x}$ is also the right eigenvector of $\hat{A}$ corresponding to $r(\hat{A})$ that $\bar{x} = \hat{A}\bar{x}$.

Moreover, $\hat{A}^t r_0 \to \bar{x}$ as $t \to \infty$ for any $r_0 = x_0/n_0$, since the above discussion implies

$$r_t = \hat{A}^t r_0 = (1 + g)^{-t} A^t r_0 = r(A)^{-t} A^t r_0 \to \begin{bmatrix} \bar{u} & \bar{u} \\ \bar{e} & \bar{e} \end{bmatrix} r_0 = \begin{bmatrix} \bar{u} \\ \bar{e} \end{bmatrix}.$$

This is illustrated below.

```
lm = LakeModel()
e_0 = 0.92           # Initial employment
u_0 = 1 - e_0        # Initial unemployment, given initial n_0 = 1

lm = LakeModel()
T = 100              # Simulation length

x_0 = (u_0, e_0)

x_path = lm.simulate_path(x_0, T)

rate_path = x_path / x_path.sum(0)

fig, axes = plt.subplots(2, 1, figsize=(10, 8))

# Plot steady ū and ē
axes[0].hlines(lm.ū, 0, T, 'r', '--', lw=2, label='ū')
axes[1].hlines(lm.ē, 0, T, 'r', '--', lw=2, label='ē')

titles = ['Unemployment rate', 'Employment rate']
```

(continues on next page)

```
locations = ['lower right', 'upper right']

# Plot unemployment rate and employment rate
for i, ax in enumerate(axes):
    ax.plot(rate_path[i, :], lw=2, alpha=0.6)
    ax.set_title(titles[i])
    ax.grid()
    ax.legend(loc=locations[i])


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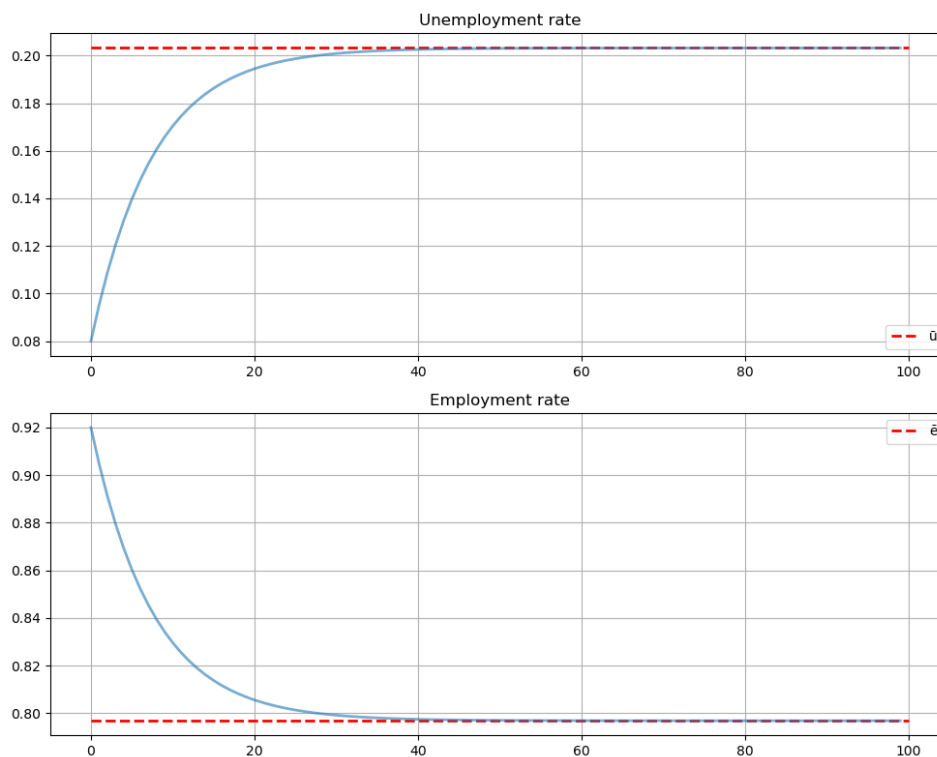

To provide more intuition for convergence, we further explain the convergence below without the Perron-Frobenius theorem.

Suppose that $\hat{A} = PDP^{-1}$ is diagonalizable, where $P = [v_1, v_2]$ consists of eigenvectors v_1 and v_2 of $\hat{A}$ corresponding to eigenvalues γ_1 and γ_2 respectively, and $D = \text{diag}(\gamma_1, \gamma_2)$.

Let $\gamma_1 = r(\hat{A}) = 1$ and $|\gamma_2| < \gamma_1$, so that the spectral radius is a dominant eigenvalue.

The dynamics of the rates follow $r_{t+1} = \hat{A} r_t$, where r_0 is a probability vector: $\sum_j r_{0,j} = 1$.

Consider $z_t = P^{-1} r_t$.

Then, we have $z_{t+1} = P^{-1}r_{t+1} = P^{-1}\hat{A}r_t = P^{-1}\hat{A}Pz_t = Dz_t$.

Hence, we obtain $z_t = D^t z_0$, and for some $z_0 = (c_1, c_2)^\top$ we have

$$r_t = Pz_t = \begin{bmatrix} v_1 & v_2 \end{bmatrix} \begin{bmatrix} \gamma_1^t & 0 \\ 0 & \gamma_2^t \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = c_1 \gamma_1^t v_1 + c_2 \gamma_2^t v_2.$$

Since $|\gamma_2| < |\gamma_1| = 1$, the second term in the right hand side converges to zero.

Therefore, the convergence follows $r_t \to c_1 v_1$.

Since the column sums of $\hat{A}$ are one and r_0 is a probability vector, r_t must be a probability vector.

In this case, $c_1 v_1$ must be a normalized eigenvector, so $c_1 v_1 = \bar{x}$ and then $r_t \to \bar{x}$.

40.4 Exercise

Exercise 40.4.1 (Evolution of unemployment and employment rate)

How do the long-run unemployment rate and employment rate evolve if there is an increase in the separation rate α or a decrease in job finding rate λ?

Is the result compatible with your intuition?

Plot the graph to illustrate how the line $D := \{x \in \mathbb{R}^2 : x = \alpha\bar{x} \text{ for some } \alpha > 0\}$ shifts in the unemployment-employment space.

Solution to Exercise 40.4.1 (Evolution of unemployment and employment rate)

Eq. (**??**) implies that the long-run unemployment rate will increase, and the employment rate will decrease if α increases or λ decreases.

Suppose first that $\alpha = 0.01, \lambda = 0.1, d = 0.02, b = 0.025$. Assume that α increases to 0.04.

The below graph illustrates that the line D shifts clockwise downward, which indicates that the fraction of unemployment rises as the separation rate increases.

```
fig, ax = plt.subplots(figsize=(10, 8))

lm = LakeModel(α=0.01, λ=0.1, d=0.02, b=0.025)
plot_time_paths(lm, ax=ax)
s=10
ax.plot([0, s * lm.ū], [0, s * lm.ē], "k--", lw=1, label='set $D$, α=0.01')
```

```
lm = LakeModel(α=0.04, λ=0.1, d=0.02, b=0.025)
plot_time_paths(lm, ax=ax)
ax.plot([0, s * lm.ū], [0, s * lm.ē], "r--", lw=1, label='set $D$, α=0.04')

ax.legend(loc='best')
plt.show()
```

CHAPTER

FORTYONE
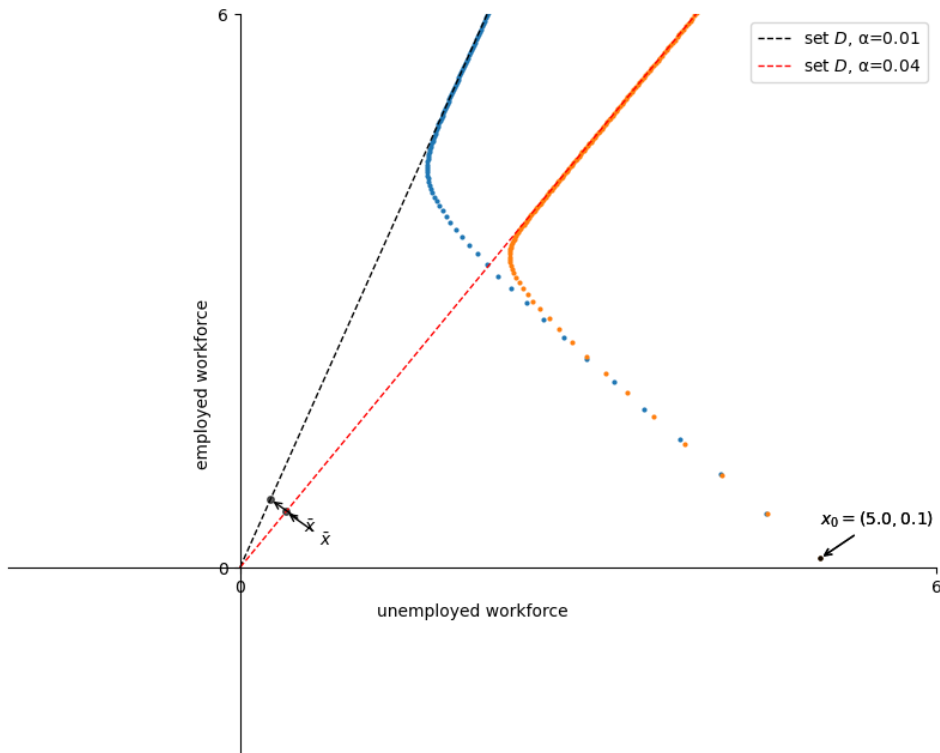
NETWORKS

```
!pip install quantecon-book-networks pandas-datareader
```

41.1 Outline

In recent years there has been rapid growth in a field called network science.

Network science studies relationships between groups of objects.

One important example is the world wide web , where web pages are connected by hyperlinks.

Another is the human brain: studies of brain function emphasize the network of connections between nerve cells (neurons).

Artificial neural networks are based on this idea, using data to build intricate connections between simple processing units.

Epidemiologists studying transmission of diseases like COVID-19 analyze interactions between groups of human hosts.

In operations research, network analysis is used to study fundamental problems as on minimum cost flow, the traveling salesman, shortest paths, and assignment.

This lecture gives an introduction to economic and financial networks.

Some parts of this lecture are drawn from the text https://networks.quantecon.org/ but the level of this lecture is more introductory.

We will need the following imports.

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import pandas as pd
import quantecon as qe
```

```python
import matplotlib.cm as cm
import quantecon_book_networks.input_output as qbn_io
import quantecon_book_networks.data as qbn_data


import matplotlib.patches as mpatches
```

```
OMP: Info #276: omp_set_nested routine deprecated, please use omp_set_max_active_
↪levels instead.
```

41.2 Economic and financial networks

Within economics, important examples of networks include

- financial networks

- production networks

- trade networks

- transport networks and

- social networks

Social networks affect trends in market sentiment and consumer decisions.

The structure of financial networks helps to determine relative fragility of the financial system.

The structure of production networks affects trade, innovation and the propagation of local shocks.

To better understand such networks, let's look at some examples in more depth.

41.2.1 Example: Aircraft Exports

The following figure shows international trade in large commercial aircraft in 2019 based on International Trade Data
SITC Revision 2.

```
/var/folders/zy/dpt7pvt11fvcm_yxjd7kmv880000gn/T/ipykernel_54746/2125706077.py:37:
↪DeprecationWarning: `alltrue` is deprecated as of NumPy 1.25.0, and will be
↪removed in NumPy 2.0. Please use `all` instead.
  nx.draw_networkx_edges(DG,
```

The circles in the figure are called **nodes** or **vertices** –in this case they represent countries.

The arrows in the figure are called **edges** or **links**.
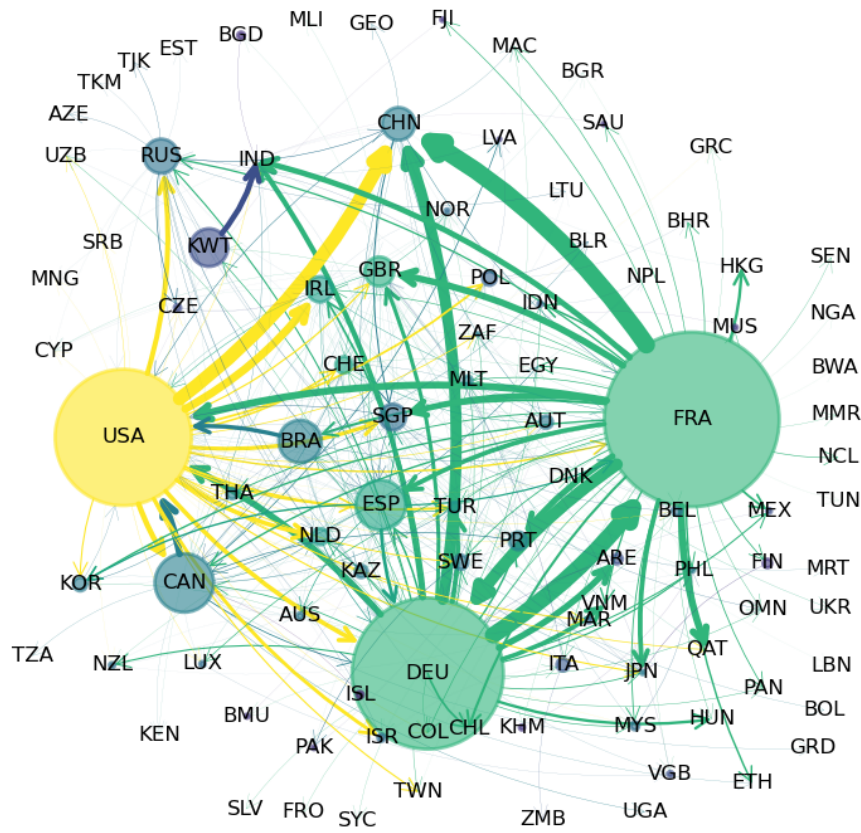
Fig. 41.1: Commercial Aircraft Network

Node size is proportional to total exports and edge width is proportional to exports to the target country.

(The data is for trade in commercial aircraft weighing at least 15,000kg and was sourced from CID Dataverse.)

The figure shows that the US, France and Germany are major export hubs.

In the discussion below, we learn to quantify such ideas.

41.2.2 Example: A Markov Chain

Recall that, in our lecture on *Markov chains* we studied a dynamic model of business cycles where the states are

- "ng"= "normal growth"

- "mr"= "mild recession"

- "sr"= "severe recession"

Let's examine the following figure
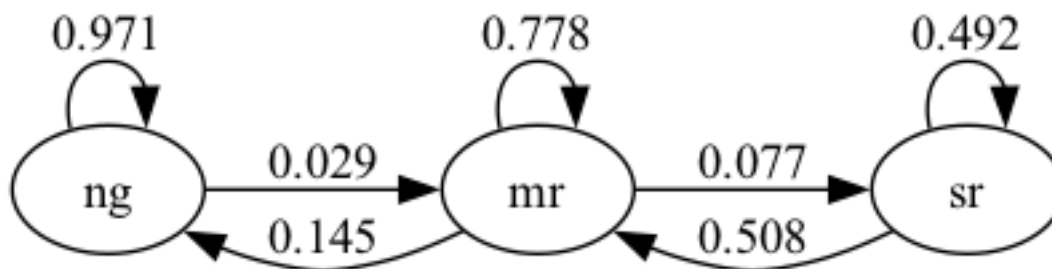
This is an example of a network, where the set of nodes V equals the states:

$$V = \{"ng", "mr", "sr"\}$$

The edges between the nodes show the one month transition probabilities.

41.3 An introduction to graph theory

Now we've looked at some examples, let's move on to theory.

This theory will allow us to better organize our thoughts.

The theoretical part of network science is constructed using a major branch of mathematics called graph theory.

Graph theory can be complicated and we will cover only the basics.

However, these concepts will already be enough for us to discuss interesting and important ideas on economic and financial networks.

We focus on "directed" graphs, where connections are, in general, asymmetric (arrows typically point one way, not both ways).

E.g.,

- bank A lends money to bank B

- firm A supplies goods to firm B

- individual A "follows" individual B on a given social network

("Undirected" graphs, where connections are symmetric, are a special case of directed graphs —we just need to insist that each arrow pointing from A to B is paired with another arrow pointing from B to A.)

41.3.1 Key definitions

A **directed graph** consists of two things:

1. a finite set V and

2. a collection of pairs (u, v) where u and v are elements of V.

The elements of V are called the **vertices** or **nodes** of the graph.

The pairs (u, v) are called the **edges** of the graph and the set of all edges will usually be denoted by E

Intuitively and visually, an edge (u, v) is understood as an arrow from node u to node v.

(A neat way to represent an arrow is to record the location of the tail and head of the arrow, and that's exactly what an edge does.)

In the aircraft export example shown in Fig. **??**

- V is all countries included in the data set.

- E is all the arrows in the figure, each indicating some positive amount of aircraft exports from one country to another.

Let's look at more examples.

Two graphs are shown below, each with three nodes.

We now construct a graph with the same nodes but different edges.

For these graphs, the arrows (edges) can be thought of as representing positive transition probabilities over a given unit of time.

In general, if an edge (u, v) exists, then the node u is called a **direct predecessor** of v and v is called a **direct successor** of u.

Also, for $v \in V$,
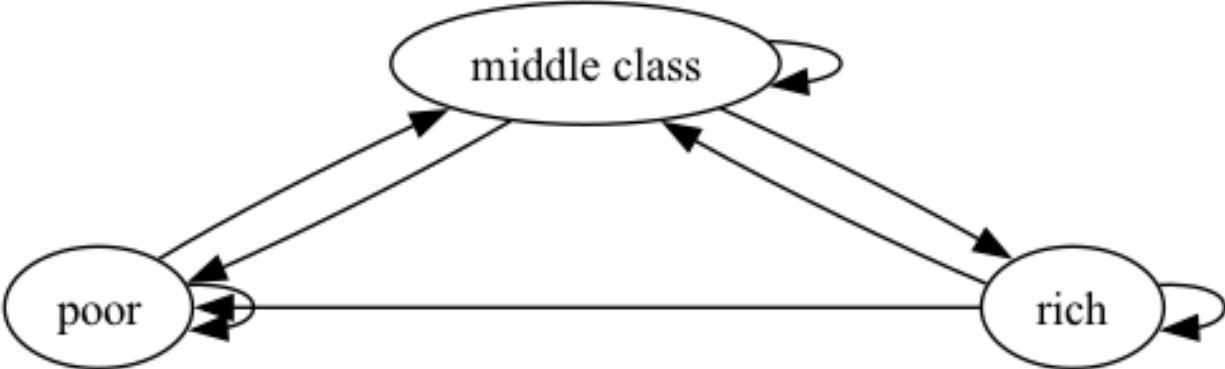
Fig. 41.2: Poverty Trap
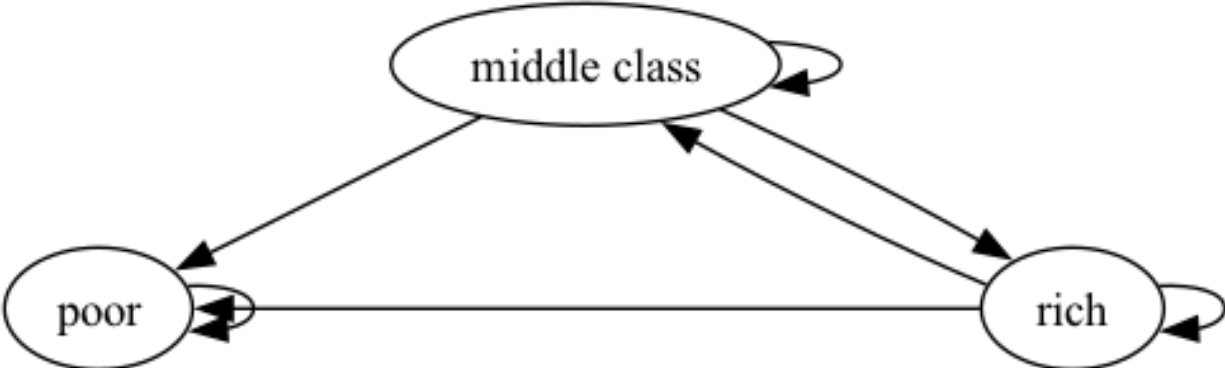
Fig. 41.3: Poverty Trap

- the **in-degree** is $i_d(v)$ = the number of direct predecessors of v and

- the **out-degree** is $o_d(v)$ = the number of direct successors of v.

41.3.2 Digraphs in Networkx

The Python package Networkx provides a convenient data structure for representing directed graphs and implements many common routines for analyzing them.

As an example, let us recreate Fig. **??** using Networkx.

To do so, we first create an empty `DiGraph` object:

```
G_p = nx.DiGraph()
```

Next we populate it with nodes and edges.

To do this we write down a list of all edges, with *poor* represented by *p* and so on:

```
edge_list = [('p', 'p'),
             ('m', 'p'), ('m', 'm'), ('m', 'r'),
             ('r', 'p'), ('r', 'm'), ('r', 'r')]
```

Finally, we add the edges to our `DiGraph` object:

```
for e in edge_list:
    u, v = e
    G_p.add_edge(u, v)
```

Alternatively, we can use the method `add_edges_from`.

```
G_p.add_edges_from(edge_list)
```

Adding the edges automatically adds the nodes, so `G_p` is now a correct representation of our graph.

We can verify this by plotting the graph via Networkx with the following code:

```
fig, ax = plt.subplots()
nx.draw_spring(G_p, ax=ax, node_size=500, with_labels=True,
               font_weight='bold', arrows=True, alpha=0.8,
               connectionstyle='arc3,rad=0.25', arrowsize=20)
plt.show()
```
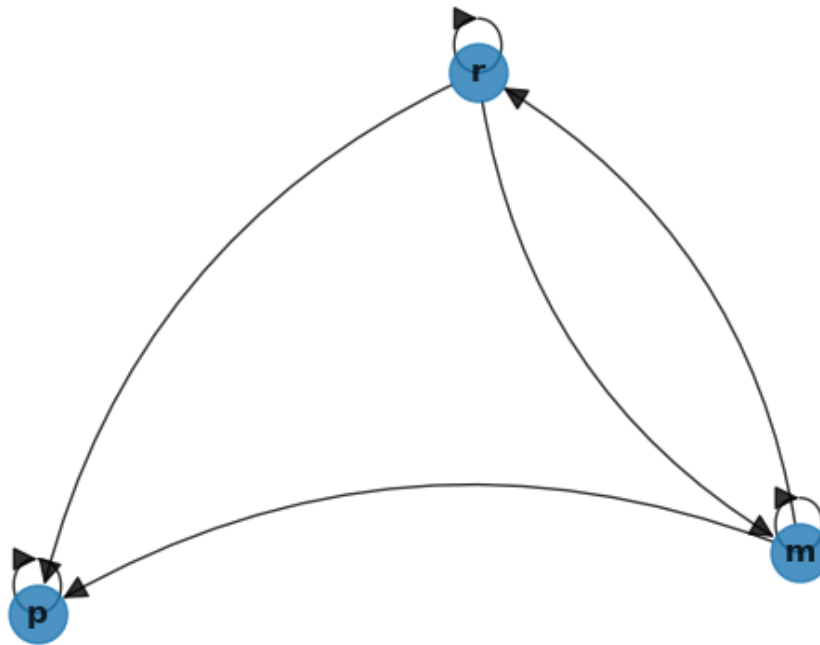
The figure obtained above matches the original directed graph in Fig. **??**.

`DiGraph` objects have methods that calculate in-degree and out-degree of nodes.

For example,

```
G_p.in_degree('p')
```

```
3
```

41.3.3 Communication

Next, we study communication and connectedness, which have important implications for economic networks.

Node v is called **accessible** from node u if either $u = v$ or there exists a sequence of edges that lead from u to v.

- in this case, we write $u \to v$

(Visually, there is a sequence of arrows leading from u to v.)

For example, suppose we have a directed graph representing a production network, where

- elements of V are industrial sectors and

- existence of an edge (i, j) means that i supplies products or services to j.

Then $m \to \ell$ means that sector m is an upstream supplier of sector ℓ.

Two nodes u and v are said to **communicate** if both $u \to v$ and $v \to u$.

A graph is called **strongly connected** if all nodes communicate.

For example, Fig. **??** is strongly connected however in Fig. **??** rich is not accessible from poor, thus it is not strongly connected.

We can verify this by first constructing the graphs using Networkx and then using `nx.is_strongly_connected`.

```
fig, ax = plt.subplots()
G1 = nx.DiGraph()

G1.add_edges_from([('p', 'p'),('p','m'),('p','r'),
            ('m', 'p'), ('m', 'm'), ('m', 'r'),
            ('r', 'p'), ('r', 'm'), ('r', 'r')])

nx.draw_networkx(G1, with_labels = True)
```
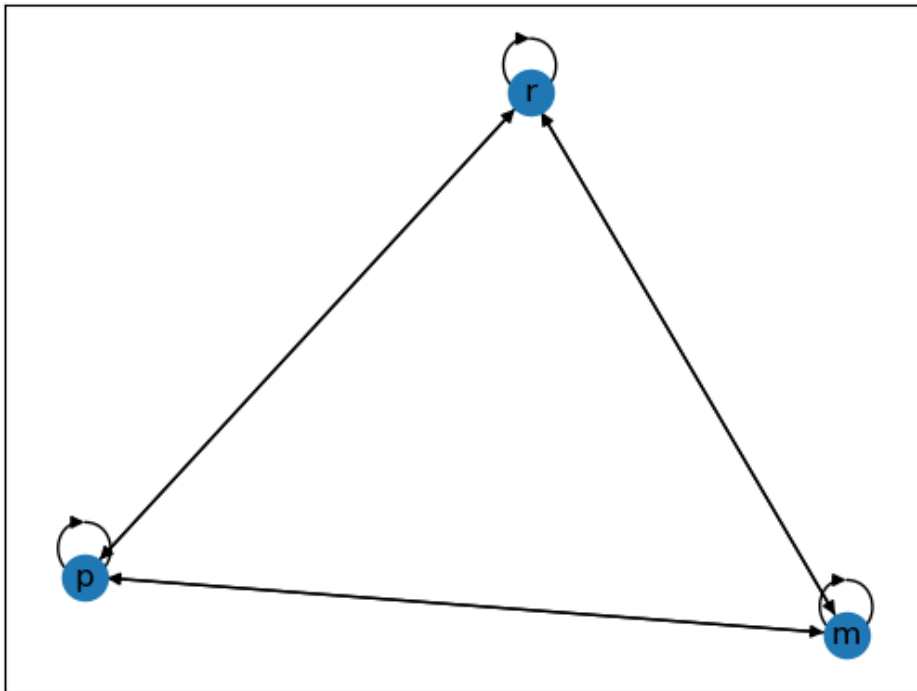


```
nx.is_strongly_connected(G1)    #checking if above graph is strongly connected
```

```
True
```

```
fig, ax = plt.subplots()
G2 = nx.DiGraph()

G2.add_edges_from([('p', 'p'),
            ('m', 'p'), ('m', 'm'), ('m', 'r'),
            ('r', 'p'), ('r', 'm'), ('r', 'r')])

nx.draw_networkx(G2, with_labels = True)
```
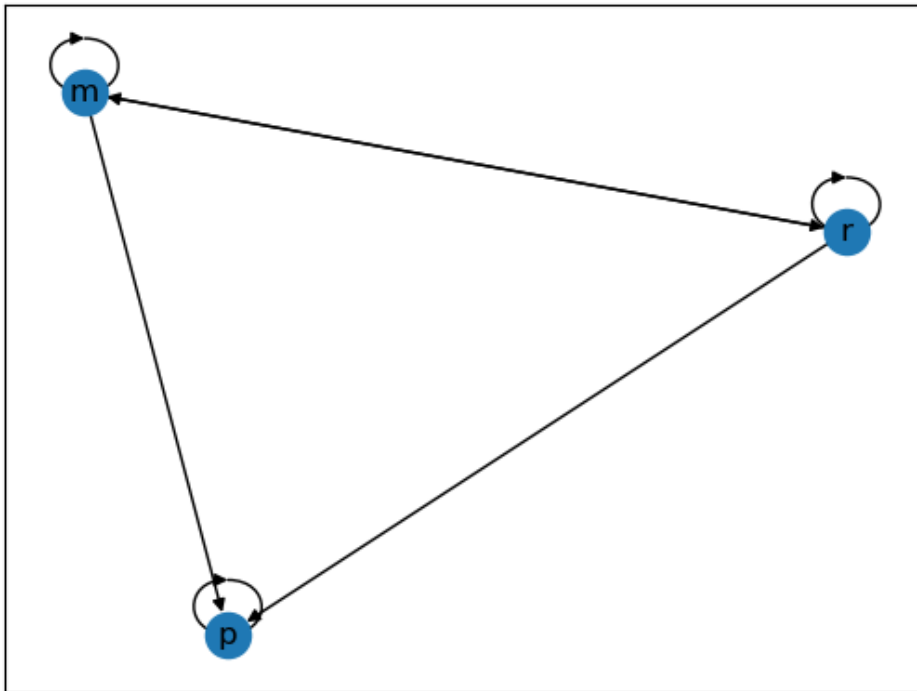


```
nx.is_strongly_connected(G2)      #checking if above graph is strongly connected
```

```
False
```

41.4 Weighted graphs

We now introduce weighted graphs, where weights (numbers) are attached to each edge.

41.4.1 International private credit flows by country

To motivate the idea, consider the following figure which shows flows of funds (i.e., loans) between private banks, grouped by country of origin.

```
/var/folders/zy/dpt7pvt11fvcm_yxjd7kmv880000gn/T/ipykernel_54746/2826816414.py:40:
↪DeprecationWarning: `alltrue` is deprecated as of NumPy 1.25.0, and will be
↪removed in NumPy 2.0. Please use `all` instead.
  nx.draw_networkx_edges(G,
```
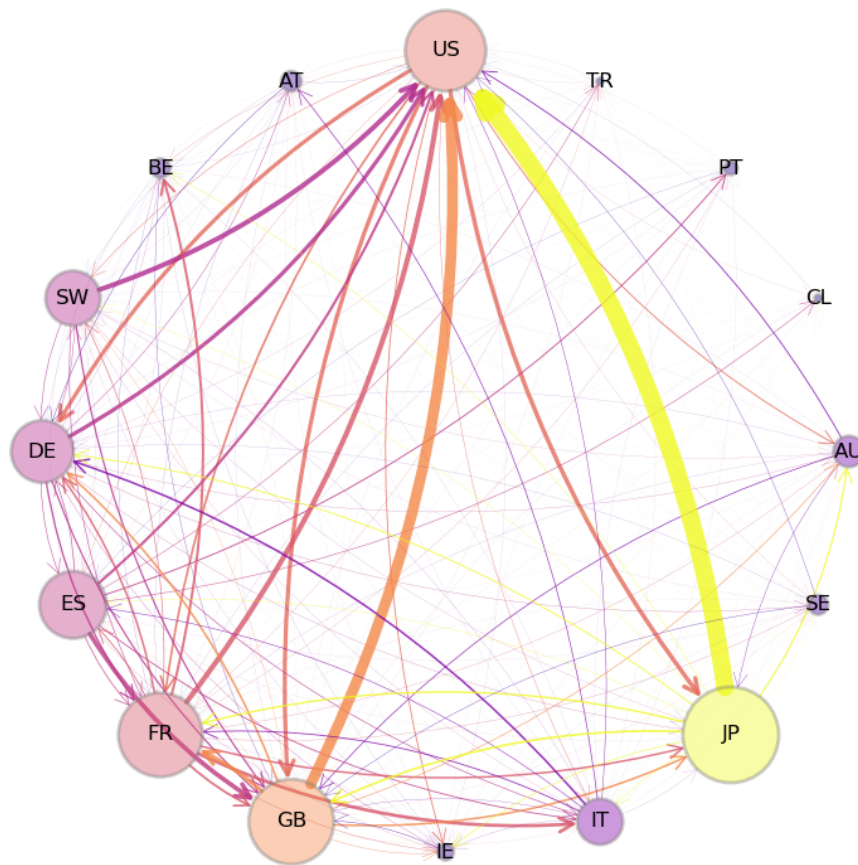

Fig. 41.4: International Credit Network

The country codes are given in the following table

Code	Country	Code	Country	Code	Country	Code	Country
AU	Australia	DE	Germany	CL	Chile	ES	Spain
PT	Portugal	FR	France	TR	Turkey	GB	United Kingdom
US	United States	IE	Ireland	AT	Austria	IT	Italy
BE	Belgium	JP	Japan	SW	Switzerland	SE	Sweden

An arrow from Japan to the US indicates aggregate claims held by Japanese banks on all US-registered banks, as collected by the Bank of International Settlements (BIS).

The size of each node in the figure is increasing in the total foreign claims of all other nodes on this node.

The widths of the arrows are proportional to the foreign claims they represent.

Notice that, in this network, an edge (u, v) exists for almost every choice of u and v (i.e., almost every country in the network).

(In fact, there are even more small arrows, which we have dropped for clarity.)

Hence the existence of an edge from one node to another is not particularly informative.

To understand the network, we need to record not just the existence or absence of a credit flow, but also the size of the flow.

The correct data structure for recording this information is a "weighted directed graph".

41.4.2 Definitions

A **weighted directed graph** is a directed graph to which we have added a **weight function** w that assigns a positive number to each edge.

The figure above shows one weighted directed graph, where the weights are the size of fund flows.

The following figure shows a weighted directed graph, with arrows representing edges of the induced directed graph.

The numbers next to the edges are the weights.

In this case, you can think of the numbers on the arrows as transition probabilities for a household over, say, one year.

We see that a rich household has a 10% chance of becoming poor in one year.
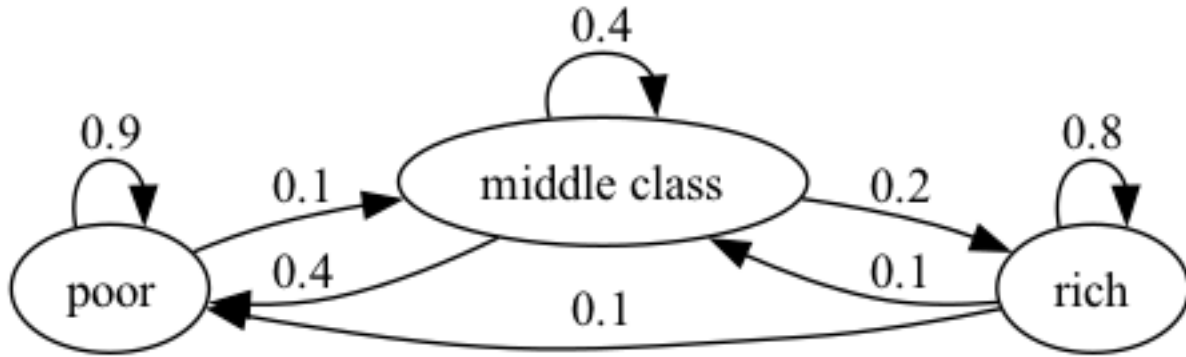
Fig. 41.5: Weighted Poverty Trap

41.5 Adjacency matrices

Another way that we can represent weights, which turns out to be very convenient for numerical work, is via a matrix.

The **adjacency matrix** of a weighted directed graph with nodes $\{v_1, \ldots, v_n\}$, edges E and weight function w is the matrix

$$A = (a_{ij})_{1 \leq i,j \leq n} \quad \text{with} \quad a_{ij} = \begin{cases} w(v_i, v_j) & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

Once the nodes in V are enumerated, the weight function and adjacency matrix provide essentially the same information.

For example, with $\{$poor, middle, rich$\}$ mapped to $\{1, 2, 3\}$ respectively, the adjacency matrix corresponding to the weighted directed graph in Fig. **??** is

$$\begin{pmatrix} 0.9 & 0.1 & 0 \\ 0.4 & 0.4 & 0.2 \\ 0.1 & 0.1 & 0.8 \end{pmatrix}.$$

In QuantEcon's `DiGraph` implementation, weights are recorded via the keyword `weighted`:

```
A = ((0.9, 0.1, 0.0),
     (0.4, 0.4, 0.2),
     (0.1, 0.1, 0.8))
A = np.array(A)
G = qe.DiGraph(A, weighted=True)    # store weights
```

One of the key points to remember about adjacency matrices is that taking the transpose *reverses all the arrows* in the associated directed graph.

For example, the following directed graph can be interpreted as a stylized version of a financial network, with nodes as banks and edges showing the flow of funds.

```
G4 = nx.DiGraph()

G4.add_edges_from([('1','2'),
                   ('2','1'),('2','3'),
                   ('3','4'),
                   ('4','2'),('4','5'),
                   ('5','1'),('5','3'),('5','4')])
pos = nx.circular_layout(G4)

edge_labels={('1','2'): '100',
             ('2','1'): '50', ('2','3'): '200',
             ('3','4'): '100',
             ('4','2'): '500', ('4','5'): '50',
             ('5','1'): '150',('5','3'): '250', ('5','4'): '300'}

nx.draw_networkx(G4, pos, node_color = 'none',node_size = 500)
nx.draw_networkx_edge_labels(G4, pos, edge_labels=edge_labels)
nx.draw_networkx_nodes(G4, pos, linewidths= 0.5, edgecolors = 'black',
                       node_color = 'none',node_size = 500)

plt.show()
```
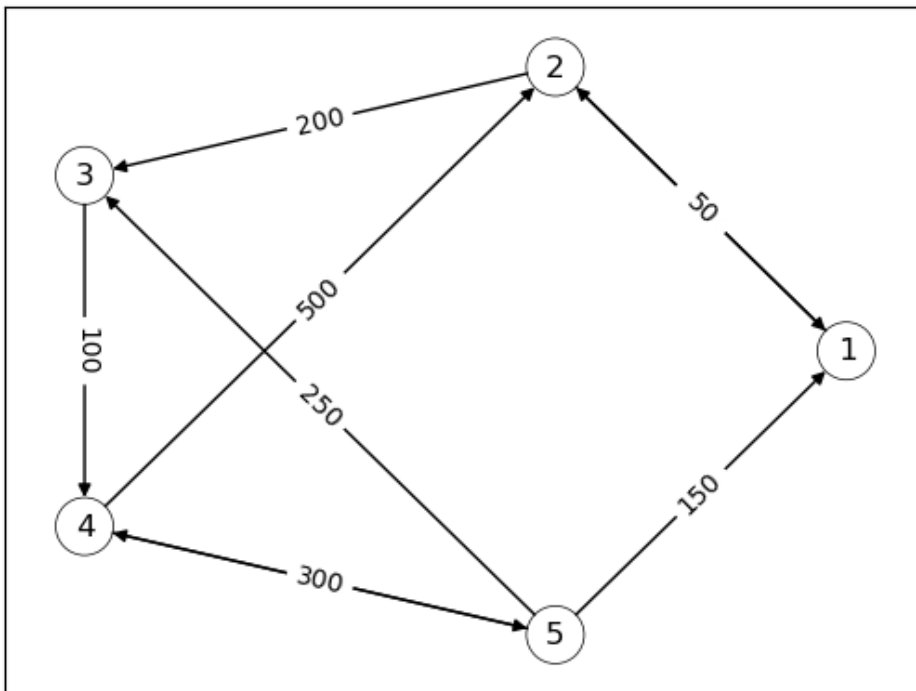

We see that bank 2 extends a loan of size 200 to bank 3.

The corresponding adjacency matrix is

$$
A = \begin{pmatrix}
0 & 100 & 0 & 0 & 0 \\
50 & 0 & 200 & 0 & 0 \\
0 & 0 & 0 & 100 & 0 \\
0 & 500 & 0 & 0 & 50 \\
150 & 0 & 250 & 300 & 0
\end{pmatrix}.
$$

The transpose is

$$
A^\top = \begin{pmatrix}
0 & 50 & 0 & 0 & 150 \\
100 & 0 & 0 & 500 & 0 \\
0 & 200 & 0 & 0 & 250 \\
0 & 0 & 100 & 0 & 300 \\
0 & 0 & 0 & 50 & 0
\end{pmatrix}.
$$

The corresponding network is visualized in the following figure which shows the network of liabilities after the loans have been granted.

Both of these networks (original and transpose) are useful for analyzing financial markets.

```
G5 = nx.DiGraph()

G5.add_edges_from([('1','2'),('1','5'),
                   ('2','1'),('2','4'),
                   ('3','2'),('3','5'),
                   ('4','3'),('4','5'),
                   ('5','4')])

edge_labels={('1','2'): '50', ('1','5'): '150',
             ('2','1'): '100', ('2','4'): '500',
             ('3','2'): '200', ('3','5'): '250',
             ('4','3'): '100', ('4','5'): '300',
             ('5','4'): '50'}

nx.draw_networkx(G5, pos, node_color = 'none',node_size = 500)
nx.draw_networkx_edge_labels(G5, pos, edge_labels=edge_labels)
nx.draw_networkx_nodes(G5, pos, linewidths= 0.5, edgecolors = 'black',
                       node_color = 'none',node_size = 500)

plt.show()
```
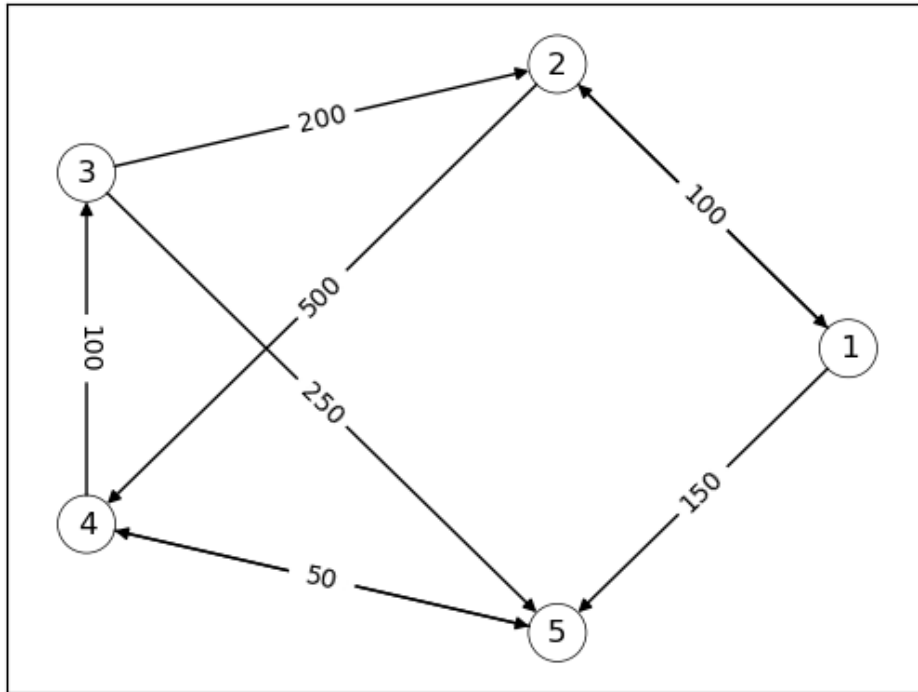
In general, every nonnegative $n \times n$ matrix $A = (a_{ij})$ can be viewed as the adjacency matrix of a weighted directed graph.

To build the graph we set $V = 1, \dots, n$ and take the edge set E to be all (i, j) such that $a_{ij} > 0$.

For the weight function we set $w(i, j) = a_{ij}$ for all edges (i, j).

We call this graph the weighted directed graph induced by A.

41.6 Properties

Consider a weighted directed graph with adjacency matrix A.

Let a_{ij}^k be element i, j of A^k, the k-th power of A.

The following result is useful in many applications:

Theorem 41.6.1

For distinct nodes i, j in V and any integer k, we have

$$a_{ij}^k > 0 \quad \text{if and only if} \quad j \text{ is accessible from } i.$$

The above result is obvious when $k = 1$ and a proof of the general case can be found in [Sargent and Stachurski, 2022].

Now recall from the eigenvalues lecture that a nonnegative matrix A is called *irreducible* if for each (i, j) there is an integer $k \geq 0$ such that $a_{ij}^k > 0$.

From the preceding theorem, it is not too difficult (see [Sargent and Stachurski, 2022] for details) to get the next result.

Theorem 41.6.2

For a weighted directed graph the following statements are equivalent:

1. The directed graph is strongly connected.

2. The adjacency matrix of the graph is irreducible.

We illustrate the above theorem with a simple example.

Consider the following weighted directed graph.
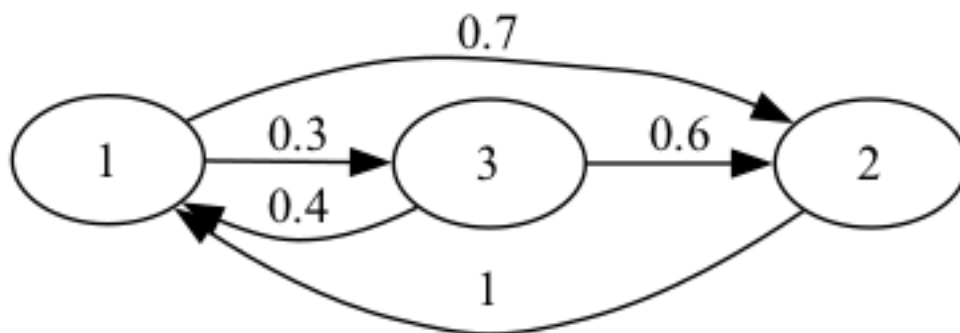
We first create the above network as a Networkx `DiGraph` object.

```
G6 = nx.DiGraph()

G6.add_edges_from([('1','2'),('1','3'),
                   ('2','1'),
                   ('3','1'),('3','2')])
```

Then we construct the associated adjacency matrix A.

```
A = np.array([[0,0.7,0.3],    # adjacency matrix A
              [1,0,0],
              [0.4,0.6,0]])
```

```
is_irreducible(A)     # check irreducibility of A
```

```
True
```

```
nx.is_strongly_connected(G6)        # check connectedness of graph
```

```
True
```

41.7 Network centrality

When studying networks of all varieties, a recurring topic is the relative "centrality" or "importance" of different nodes.

Examples include

- ranking of web pages by search engines
- determining the most important bank in a financial network (which one a central bank should rescue if there is a financial crisis)
- determining the most important industrial sector in an economy.

In what follows, a **centrality measure** associates to each weighted directed graph a vector m where the m_i is interpreted as the centrality (or rank) of node v_i.

41.7.1 Degree centrality

Two elementary measures of "importance" of a node in a given directed graph are its in-degree and out-degree.

Both of these provide a centrality measure.

In-degree centrality is a vector containing the in-degree of each node in the graph.

Consider the following simple example.

```
G7 = nx.DiGraph()

G7.add_nodes_from(['1','2','3','4','5','6','7'])

G7.add_edges_from([('1','2'),('1','6'),
                   ('2','1'),('2','4'),
                   ('3','2'),
                   ('4','2'),
                   ('5','3'),('5','4'),
                   ('6','1'),
```

```
                      ('7','4'),('7','6')])
pos = nx.planar_layout(G7)


nx.draw_networkx(G7, pos, node_color='none', node_size=500)
nx.draw_networkx_nodes(G7, pos, linewidths=0.5, edgecolors='black',
                       node_color='none',node_size=500)


plt.show()
```
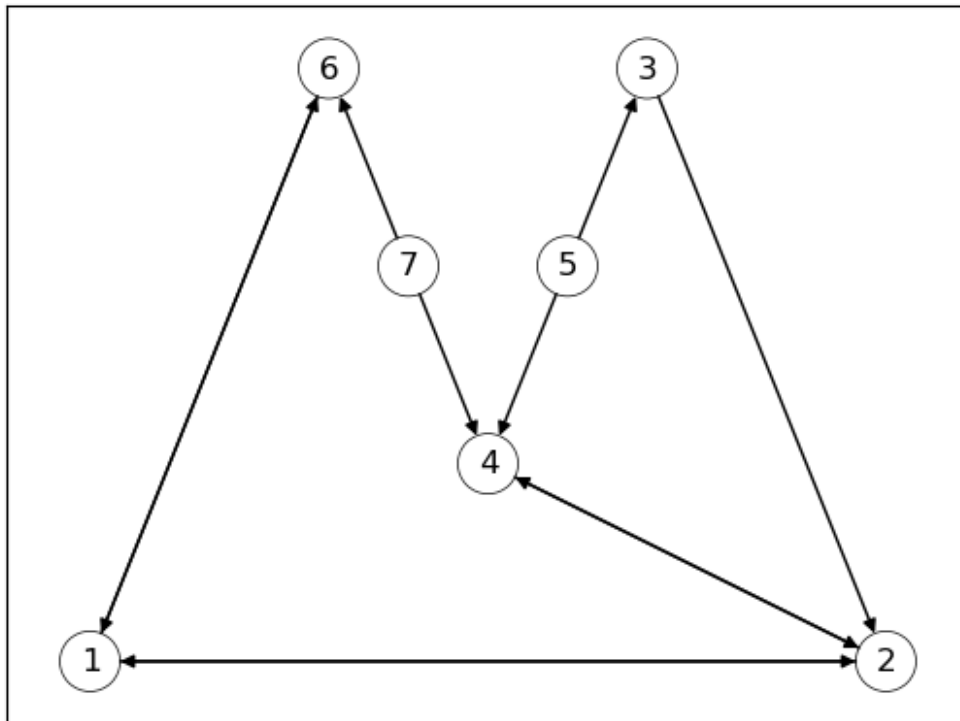

Fig. 41.6: Sample Graph

The following code displays the in-degree centrality of all nodes.

```
iG7 = [G7.in_degree(v) for v in G7.nodes()]    # computing in-degree centrality


for i, d in enumerate(iG7):
    print(i+1, d)
```

```
1 2
2 3
```

41.7. Network centrality

```
3 1
4 3
5 0
6 2
7 0
```

Consider the international credit network displayed in Fig. **??**.

The following plot displays the in-degree centrality of each country.

```
D = qbn_io.build_unweighted_matrix(Z)
indegree = D.sum(axis=0)
```

```python
def centrality_plot_data(countries, centrality_measures):
    df = pd.DataFrame({'code': countries,
                        'centrality':centrality_measures,
                        'color': qbn_io.colorise_weights(centrality_measures).tolist()
                        })
    return df.sort_values('centrality')
```

```python
fig, ax = plt.subplots()

df = centrality_plot_data(countries, indegree)

ax.bar('code', 'centrality', data=df, color=df["color"], alpha=0.6)

patch = mpatches.Patch(color=None, label='in degree', visible=False)
ax.legend(handles=[patch], fontsize=12, loc="upper left", handlelength=0,␣
 ↪frameon=False)

ax.set_ylim((0,20))

plt.show()
```
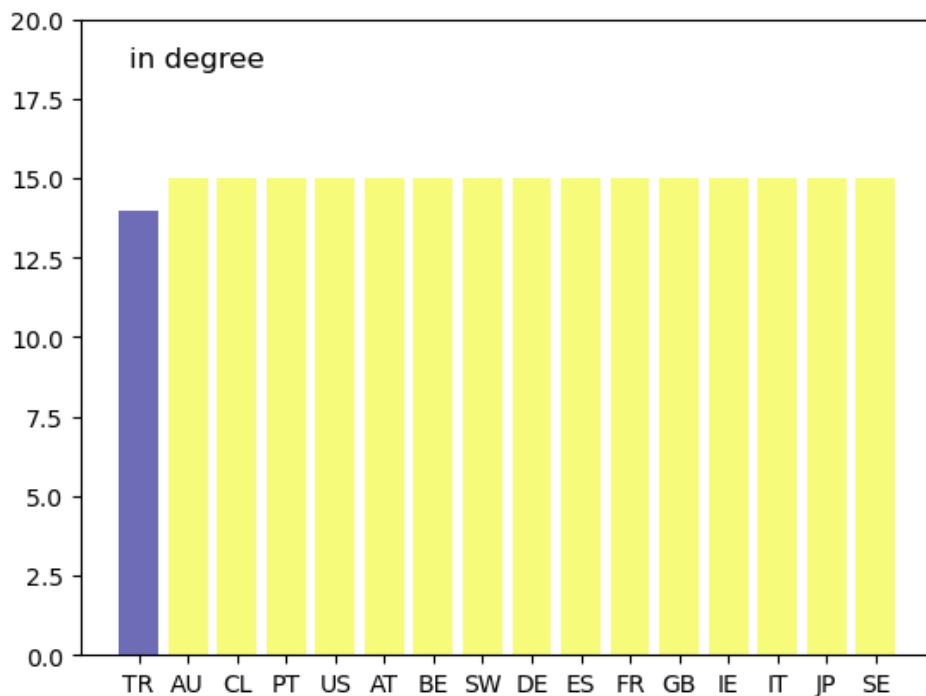
Unfortunately, while in-degree and out-degree centrality are simple to calculate, they are not always informative.

In Fig. **??**, an edge exists between almost every node, so the in- or out-degree based centrality ranking fails to effectively separate the countries.

This can be seen in the above graph as well.

Another example is the task of a web search engine, which ranks pages by relevance whenever a user enters a search.

Suppose web page A has twice as many inbound links as page B.

In-degree centrality tells us that page A deserves a higher rank.

But in fact, page A might be less important than page B.

To see why, suppose that the links to A are from pages that receive almost no traffic, while the links to B are from pages that receive very heavy traffic.

In this case, page B probably receives more visitors, which in turn suggests that page B contains more valuable (or entertaining) content.

Thinking about this point suggests that importance might be *recursive*.

This means that the importance of a given node depends on the importance of other nodes that link to it.

As another example, we can imagine a production network where the importance of a given sector depends on the importance of the sectors that it supplies.

This reverses the order of the previous example: now the importance of a given node depends on the importance of other

nodes that *it links to*.

The next centrality measures will have these recursive features.

41.7.2 Eigenvector centrality

Suppose we have a weighted directed graph with adjacency matrix A.

For simplicity, we will suppose that the nodes V of the graph are just the integers $1, \dots, n$.

Let $r(A)$ denote the *spectral radius* of A.

The **eigenvector centrality** of the graph is defined as the n-vector e that solves

$$e = \frac{1}{r(A)} Ae. \tag{41.1}$$

In other words, e is the dominant eigenvector of A (the eigenvector of the largest eigenvalue —see the discussion of the *Perron-Frobenius theorem* in the eigenvalue lecture.

To better understand (**??**), we write out the full expression for some element e_i

$$e_i = \frac{1}{r(A)} \sum_{1 \le j \le n} a_{ij} e_j \tag{41.2}$$

Note the recursive nature of the definition: the centrality obtained by node i is proportional to a sum of the centrality of all nodes, weighted by the *rates of flow* from i into these nodes.

A node i is highly ranked if

1. there are many edges leaving i,

2. these edges have large weights, and

3. the edges point to other highly ranked nodes.

Later, when we study demand shocks in production networks, there will be a more concrete interpretation of eigenvector centrality.

We will see that, in production networks, sectors with high eigenvector centrality are important *suppliers*.

In particular, they are activated by a wide array of demand shocks once orders flow backwards through the network.

To compute eigenvector centrality we can use the following function.

```
def eigenvector_centrality(A, k=40, authority=False):
    """
    Computes the dominant eigenvector of A. Assumes A is
    primitive and uses the power method.
```

```
    """
    A_temp = A.T if authority else A
    n = len(A_temp)
    r = np.max(np.abs(np.linalg.eigvals(A_temp)))
    e = r**(-k) * (np.linalg.matrix_power(A_temp, k) @ np.ones(n))
    return e / np.sum(e)
```

Let's compute eigenvector centrality for the graph generated in Fig. **??**.

```
A = nx.to_numpy_array(G7)          # compute adjacency matrix of graph
```

```
e = eigenvector_centrality(A)
n = len(e)

for i in range(n):
    print(i+1,e[i])
```

```
1 0.18580570704268035
2 0.18580570704268035
3 0.11483424225608216
4 0.11483424225608216
5 0.14194292957319637
6 0.11483424225608216
7 0.14194292957319637
```

While nodes 2 and 4 had the highest in-degree centrality, we can see that nodes 1 and 2 have the highest eigenvector centrality.

Let's revisit the international credit network in Fig. **??**.

```
eig_central = eigenvector_centrality(Z)
```

```
fig, ax = plt.subplots()

df = centrality_plot_data(countries, eig_central)

ax.bar('code', 'centrality', data=df, color=df["color"], alpha=0.6)

patch = mpatches.Patch(color=None, visible=False)
ax.legend(handles=[patch], fontsize=12, loc="upper left", handlelength=0,␣
 ↪frameon=False)
```

```
plt.show()
```
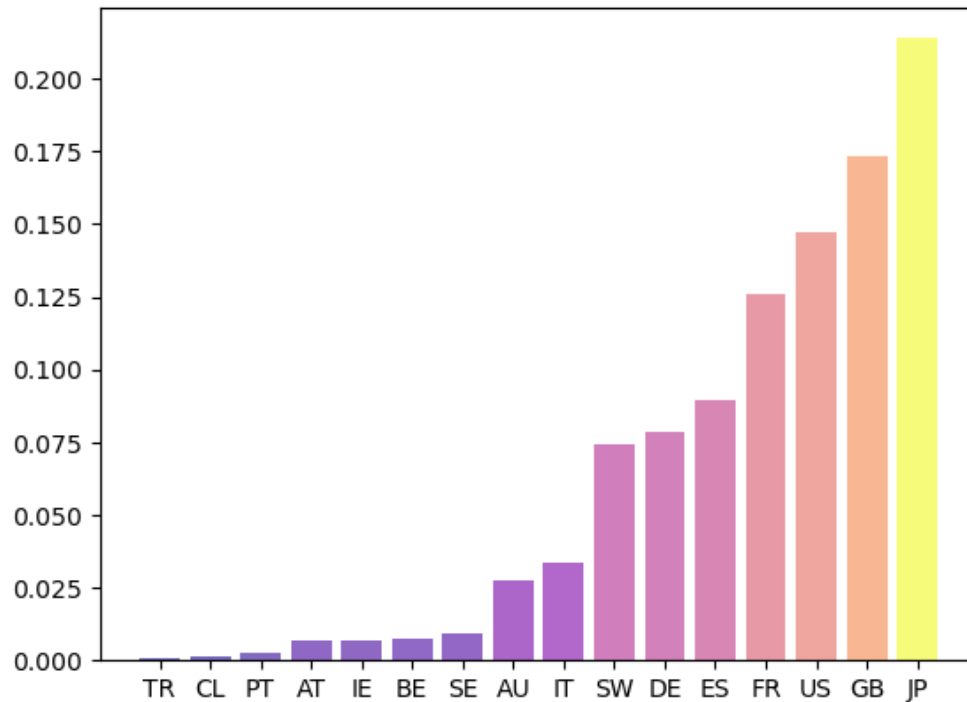

Fig. 41.7: Eigenvector centrality

Countries that are rated highly according to this rank tend to be important players in terms of supply of credit.

Japan takes the highest rank according to this measure, although countries with large financial sectors such as Great Britain and France are not far behind.

The advantage of eigenvector centrality is that it measures a node's importance while considering the importance of its neighbours.

A variant of eigenvector centrality is at the core of Google's PageRank algorithm, which is used to rank web pages.

The main principle is that links from important nodes (as measured by degree centrality) are worth more than links from unimportant nodes.

41.7.3 Katz centrality

One problem with eigenvector centrality is that $r(A)$ might be zero, in which case $1/r(A)$ is not defined.

For this and other reasons, some researchers prefer another measure of centrality for networks called Katz centrality.

Fixing β in $(0, 1/r(A))$, the **Katz centrality** of a weighted directed graph with adjacency matrix A is defined as the vector κ that solves

$$\kappa_i = \beta \sum_{1 \leq j1} a_{ij} \kappa_j + 1 \qquad \text{for all } i \in \{0, \dots, n-1\}. \tag{41.3}$$

Here β is a parameter that we can choose.

In vector form we can write

$$\kappa = \mathbf{1} + \beta A \kappa \tag{41.4}$$

where $\mathbf{1}$ is a column vector of ones.

The intuition behind this centrality measure is similar to that provided for eigenvector centrality: high centrality is conferred on i when it is linked to by nodes that themselves have high centrality.

Provided that $0 < \beta < 1/r(A)$, Katz centrality is always finite and well-defined because then $r(\beta A) < 1$.

This means that (**??**) has the unique solution

$$\kappa = (I - \beta A)^{-1} \mathbf{1}$$

This follows from the *Neumann series theorem*.

The parameter β is used to ensure that κ is finite

When $r(A) < 1$, we use $\beta = 1$ as the default for Katz centrality computations.

41.7.4 Authorities vs hubs

Search engine designers recognize that web pages can be important in two different ways.

Some pages have high **hub centrality**, meaning that they link to valuable sources of information (e.g., news aggregation sites).

Other pages have high **authority centrality**, meaning that they contain valuable information, as indicated by the number and significance of incoming links (e.g., websites of respected news organizations).

Similar ideas can and have been applied to economic networks (often using different terminology).

The eigenvector centrality and Katz centrality measures we discussed above measure hub centrality.

(Nodes have high centrality if they point to other nodes with high centrality.)

If we care more about authority centrality, we can use the same definitions except that we take the transpose of the adjacency matrix.

This works because taking the transpose reverses the direction of the arrows.

(Now nodes will have high centrality if they receive links from other nodes with high centrality.)

For example, the **authority-based eigenvector centrality** of a weighted directed graph with adjacency matrix A is the vector e solving

$$e = \frac{1}{r(A)} A^\top e. \tag{41.5}$$

The only difference from the original definition is that A is replaced by its transpose.

(Transposes do not affect the spectral radius of a matrix so we wrote $r(A)$ instead of $r(A^\top)$.)

Element-by-element, this is given by

$$e_j = \frac{1}{r(A)} \sum_{1 \leq i \leq n} a_{ij} e_i \tag{41.6}$$

We see e_j will be high if many nodes with high authority rankings link to j.

The following figurenshows the authority-based eigenvector centrality ranking for the international credit network shown in Fig. **??**.

```
ecentral_authority = eigenvector_centrality(Z, authority=True)
```

```
fig, ax = plt.subplots()

df = centrality_plot_data(countries, ecentral_authority)

ax.bar('code', 'centrality', data=df, color=df["color"], alpha=0.6)

patch = mpatches.Patch(color=None, visible=False)
ax.legend(handles=[patch], fontsize=12, loc="upper left", handlelength=0,␣
 ↪frameon=False)

plt.show()
```

Highly ranked countries are those that attract large inflows of credit, or credit inflows from other major players.

In this case the US clearly dominates the rankings as a target of interbank credit.
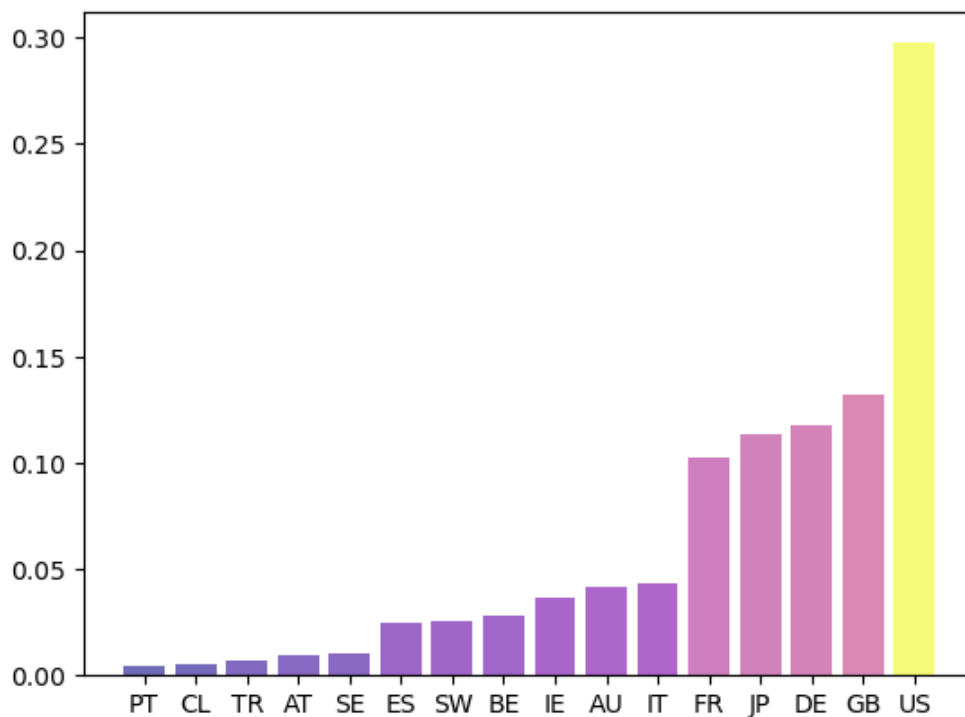
Fig. 41.8: Eigenvector authority

41.8 Further reading

We apply the ideas discussed in this lecture to:

Textbooks on economic and social networks include [Jackson, 2010], [Easley *et al.*, 2010], [Borgatti *et al.*, 2018], [Sargent and Stachurski, 2022] and [Goyal, 2023].

Within the realm of network science, the texts by [Newman, 2018], [Menczer *et al.*, 2020] and [Coscia, 2021] are excellent.

41.9 Exercises

Exercise 41.9.1

Here is a mathematical exercise for those who like proofs.

Let (V, E) be a directed graph and write $u \sim v$ if u and v communicate.

Show that $\sim$ is an equivalence relation on V.

Solution to Exercise 41.9.1

Reflexivity:

Trivially, $u = v \Rightarrow u \to v$.

Thus, $u \sim u$.

Symmetry: Suppose, $u \sim v$

$\Rightarrow u \to v$ and $v \to u$.

By definition, this implies $v \sim u$.

Transitivity:

Suppose, $u \sim v$ and $v \sim w$

This implies, $u \to v$ and $v \to u$ and also $v \to w$ and $w \to v$.

Thus, we can conclude $u \to v \to w$ and $w \to v \to u$.

Which means $u \sim w$.

Exercise 41.9.2

Consider a directed graph G with the set of nodes

$$V = \{0, 1, 2, 3, 4, 5, 6, 7\}$$

and the set of edges

$$E = \{(0, 1), (0, 3), (1, 0), (2, 4), (3, 2), (3, 4), (3, 7), (4, 3), (5, 4), (5, 6), (6, 3), (6, 5), (7, 0)\}$$

1. Use `Networkx` to draw graph G.

2. Find the associated adjacency matrix A for G.

3. Use the functions defined above to compute in-degree centrality, out-degree centrality and eigenvector centrality of G.

Solution to Exercise 41.9.2

```
# First, let's plot the given graph
```

<div align="right">(continues on next page)</div>

```
G = nx.DiGraph()

G.add_nodes_from(np.arange(8))   # adding nodes

G.add_edges_from([(0,1),(0,3),        # adding edges
                  (1,0),
                  (2,4),
                  (3,2),(3,4),(3,7),
                  (4,3),
                  (5,4),(5,6),
                  (6,3),(6,5),
                  (7,0)])

nx.draw_networkx(G, pos=nx.circular_layout(G), node_color='gray', node_size=500, with_
 ↪labels=True)

plt.show()
```
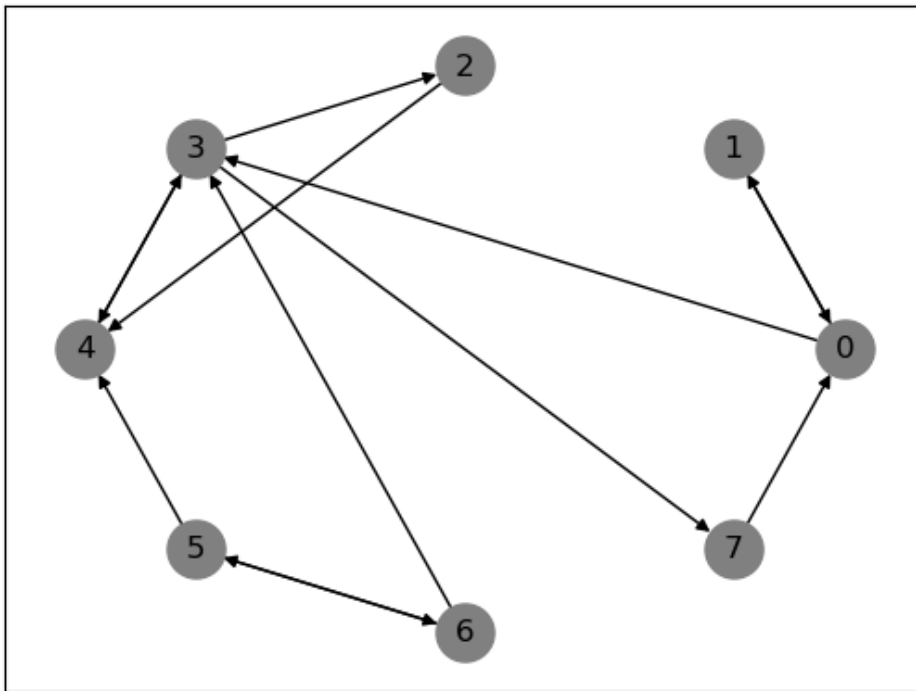


```
A = nx.to_numpy_array(G)      #find adjacency matrix associated with G

A
```

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

```python
oG = [G.out_degree(v) for v in G.nodes()]   # computing in-degree centrality

for i, d in enumerate(oG):
    print(i, d)
```

```
0 2
1 1
2 1
3 3
4 1
5 2
6 2
7 1
```

```python
e = eigenvector_centrality(A)   # computing eigenvector centrality
n = len(e)

for i in range(n):
    print(i+1, e[i])
```

```
1 0.1458980838002507
2 0.09016989800748738
3 0.055728056024793506
4 0.14589810100962303
5 0.09016994824024989
6 0.1803397955498566
7 0.20162621936025152
8 0.09016989800748738
```

Exercise 41.9.3

Consider a graph G with n nodes and $n \times n$ adjacency matrix A.

Let $S = \sum_{k=0}^{n-1} A^k$

We can say for any two nodes i and j, j is accessible from i if and only if $S_{ij} > 0$.

Devise a function `is_accessible` that checks if any two nodes of a given graph are accessible.

Consider the graph in Exercise 41.9.2 and use this function to check if

1. 1 is accessible from 2

2. 6 is accessible from 3

Solution to Exercise 41.9.3

```python
def is_accessible(G,i,j):
    A = nx.to_numpy_array(G)
    n = len(A)
    result = np.zeros((n, n))
    for i in range(n):
        result += np.linalg.matrix_power(A, i)
    if result[i,j]>0:
        return True
    else:
        return False
```

```python
G = nx.DiGraph()

G.add_nodes_from(np.arange(8))    # adding nodes

G.add_edges_from([(0,1),(0,3),        # adding edges
                 (1,0),
                 (2,4),
                 (3,2),(3,4),(3,7),
                 (4,3),
                 (5,4),(5,6),
                 (6,3),(6,5),
                 (7,0)])
```

```python
is_accessible(G, 2, 1)
```

```
True
```

```
is_accessible(G, 3, 6)
```

```
False
```

Part XII

Markets and Competitive Equilibrium

SUPPLY AND DEMAND WITH MANY GOODS

42.1 Overview

In a *previous lecture* we studied supply, demand and welfare in a market with a single consumption good.

In this lecture, we study a setting with n goods and n corresponding prices.

Key infrastructure concepts that we'll encounter in this lecture are

- inverse demand curves

- marginal utilities of wealth

- inverse supply curves

- consumer surplus

- producer surplus

- social welfare as a sum of consumer and producer surpluses

- competitive equilibrium

We will provide a version of the first fundamental welfare theorem, which was formulated by

- Leon Walras

- Francis Ysidro Edgeworth

- Vilfredo Pareto

Important extensions to the key ideas were obtained by

- Abba Lerner

- Harold Hotelling

- Paul Samuelson

  • Kenneth Arrow

  • Gerard Debreu

We shall describe two classic welfare theorems:

  • **first welfare theorem:** for a given distribution of wealth among consumers, a competitive equilibrium allocation of goods solves a social planning problem.

  • **second welfare theorem:** An allocation of goods to consumers that solves a social planning problem can be supported by a competitive equilibrium with an appropriate initial distribution of wealth.

As usual, we start by importing some Python modules.

```python
# import some packages
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import inv
```

42.2 Formulas from linear algebra

We shall apply formulas from linear algebra that

  • differentiate an inner product with respect to each vector

  • differentiate a product of a matrix and a vector with respect to the vector

  • differentiate a quadratic form in a vector with respect to the vector

Where a is an $n \times 1$ vector, A is an $n \times n$ matrix, and x is an $n \times 1$ vector:

$$\frac{\partial a^\top x}{\partial x} = \frac{\partial x^\top a}{\partial x} = a$$

$$\frac{\partial Ax}{\partial x} = A$$

$$\frac{\partial x^\top Ax}{\partial x} = (A + A^\top)x$$

42.3 From utility function to demand curve

Our study of consumers will use the following primitives

  • Π be an $m \times n$ matrix,

  • b be an $m \times 1$ vector of bliss points,

  • e be an $n \times 1$ vector of endowments, and

We will analyze endogenous objects c and p, where

- c is an $n \times 1$ vector of consumptions of various goods,

- p is an $n \times 1$ vector of prices

The matrix Π describes a consumer's willingness to substitute one good for every other good.

We assume that Π has linearly independent columns, which implies that $\Pi^\top \Pi$ is a positive definite matrix.

- it follows that $\Pi^\top \Pi$ has an inverse.

We shall see below that $(\Pi^\top \Pi)^{-1}$ is a matrix of slopes of (compensated) demand curves for c with respect to a vector of prices:

$$\frac{\partial c}{\partial p} = (\Pi^\top \Pi)^{-1}$$

A consumer faces p as a price taker and chooses c to maximize the utility function

$$-\frac{1}{2}(\Pi c - b)^\top (\Pi c - b) \tag{42.1}$$

subject to the budget constraint

$$p^\top (c - e) = 0 \tag{42.2}$$

We shall specify examples in which Π and b are such that it typically happens that

$$\Pi c \ll b \tag{42.3}$$

This means that the consumer has much less of each good than he wants.

The deviation in (**??**) will ultimately assure us that competitive equilibrium prices are positive.

42.3.1 Demand curve implied by constrained utility maximization

For now, we assume that the budget constraint is (**??**).

So we'll be deriving what is known as a **Marshallian** demand curve.

Our aim is to maximize *(42.3.1)* subject to *(42.3.2)*.

Form a Lagrangian

$$L = -\frac{1}{2}(\Pi c - b)^\top (\Pi c - b) + \mu[p^\top (e - c)]$$

where μ is a Lagrange multiplier that is often called a **marginal utility of wealth**.

The consumer chooses c to maximize L and μ to minimize it.

First-order conditions for c are

$$\frac{\partial L}{\partial c} = -\Pi^\top \Pi c + \Pi^\top b - \mu p = 0$$

so that, given μ, the consumer chooses

$$c = (\Pi^\top \Pi)^{-1}(\Pi^\top b - \mu p) \tag{42.4}$$

Substituting (**??**) into budget constraint (**??**) and solving for μ gives

$$\mu(p, e) = \frac{p^\top (\Pi^\top \Pi)^{-1} \Pi^\top b - p^\top e}{p^\top (\Pi^\top \Pi)^{-1} p}. \tag{42.5}$$

Equation (**??**) tells how marginal utility of wealth depends on the endowment vector e and the price vector p.

Note: Equation (**??**) is a consequence of imposing that $p^\top(c - e) = 0$.

We could instead take μ as a parameter and use (**??**) and the budget constraint (**??**) to solve for wealth.

Which way we proceed determines whether we are constructing a **Marshallian** or **Hicksian** demand curve.

42.4 Endowment economy

We now study a pure-exchange economy, or what is sometimes called an endowment economy.

Consider a single-consumer, multiple-goods economy without production.

The only source of goods is the single consumer's endowment vector e.

A competitive equilibrium price vector induces the consumer to choose $c = e$.

This implies that the equilibrium price vector satisfies

$$p = \mu^{-1}(\Pi^\top b - \Pi^\top \Pi e)$$

In the present case where we have imposed budget constraint in the form (**??**), we are free to normalize the price vector by setting the marginal utility of wealth $\mu = 1$ (or any other value for that matter).

This amounts to choosing a common unit (or numeraire) in which prices of all goods are expressed.

(Doubling all prices will affect neither quantities nor relative prices.)

We'll set $\mu = 1$.

Exercise 42.4.1

Verify that setting $\mu = 1$ in (**??**) implies that formula (**??**) is satisfied.

Exercise 42.4.2

Verify that setting $\mu = 2$ in (**??**) also implies that formula (**??**) is satisfied.

Here is a class that computes competitive equilibria for our economy.

```python
class ExchangeEconomy:

    def __init__(self,
                 Π,
                 b,
                 e,
                 thres=1.5):
        """
        Set up the environment for an exchange economy

        Args:
            Π (np.array): shared matrix of substitution
            b (list):  the consumer's bliss point
            e (list):  the consumer's endowment
            thres (float): a threshold to check p >> Π e condition
        """

        # check non-satiation
        if np.min(b / np.max(Π @ e)) <= thres:
            raise Exception('set bliss points further away')


        self.Π, self.b, self.e = Π, b, e


    def competitive_equilibrium(self):
        """
        Compute the competitive equilibrium prices and allocation
        """
        Π, b, e = self.Π, self.b, self.e

        # compute price vector with μ=1
        p = Π.T @ b - Π.T @ Π @ e

        # compute consumption vector
        slope_dc = inv(Π.T @ Π)
        Π_inv = inv(Π)
        c = Π_inv @ b - slope_dc @ p

        if any(c < 0):
```

```
            print('allocation: ', c)
            raise Exception('negative allocation: equilibrium does not exist')


    return p, c
```

42.5 Digression: Marshallian and Hicksian demand curves

Sometimes we'll use budget constraint (**??**) in situations in which a consumer's endowment vector e is his **only** source of income.

Other times we'll instead assume that the consumer has another source of income (positive or negative) and write his budget constraint as

$$p^\top(c - e) = w \tag{42.6}$$

where w is measured in "dollars"(or some other **numeraire**) and component p_i of the price vector is measured in dollars per unit of good i.

Whether the consumer's budget constraint is (**??**) or (**??**) and whether we take w as a free parameter or instead as an endogenous variable will affect the consumer's marginal utility of wealth.

Consequently, how we set μ determines whether we are constructing

- a **Marshallian** demand curve, as when we use (**??**) and solve for μ using equation (**??**) above, or

- a **Hicksian** demand curve, as when we treat μ as a fixed parameter and solve for w from (**??**).

Marshallian and Hicksian demand curves contemplate different mental experiments:

For a Marshallian demand curve, hypothetical changes in a price vector have both **substitution** and **income** effects

- income effects are consequences of changes in $p^\top e$ associated with the change in the price vector

For a Hicksian demand curve, hypothetical price vector changes have only **substitution** effects

- changes in the price vector leave the $p^\top e + w$ unaltered because we freeze μ and solve for w

Sometimes a Hicksian demand curve is called a **compensated** demand curve in order to emphasize that, to disarm the income (or wealth) effect associated with a price change, the consumer's wealth w is adjusted.

We'll discuss these distinct demand curves more below.

42.6 Dynamics and risk as special cases

Special cases of our n-good pure exchange model can be created to represent

- **dynamics** —by putting different dates on different commodities

- **risk** —by interpreting delivery of goods as being contingent on states of the world whose realizations are described by a *known probability distribution*

Let's illustrate how.

42.6.1 Dynamics

Suppose that we want to represent a utility function

$$-\frac{1}{2}[(c_1 - b_1)^2 + \beta(c_2 - b_2)^2]$$

where $\beta \in (0, 1)$ is a discount factor, c_1 is consumption at time 1 and c_2 is consumption at time 2.

To capture this with our quadratic utility function (**??**), set

$$\Pi = \begin{bmatrix} 1 & 0 \\ 0 & \sqrt{\beta} \end{bmatrix}$$

$$e = \begin{bmatrix} e_1 \\ e_2 \end{bmatrix}$$

and

$$b = \begin{bmatrix} b_1 \\ \sqrt{\beta}b_2 \end{bmatrix}$$

The budget constraint (**??**) becomes

$$p_1 c_1 + p_2 c_2 = p_1 e_1 + p_2 e_2$$

The left side is the **discounted present value** of consumption.

The right side is the **discounted present value** of the consumer's endowment.

The relative price $\frac{p_1}{p_2}$ has units of time 2 goods per unit of time 1 goods.

Consequently,

$$(1 + r) := R := \frac{p_1}{p_2}$$

is the **gross interest rate** and r is the **net interest rate**.

Here is an example.

```
beta = 0.95

Π = np.array([[1, 0],
              [0, np.sqrt(beta)]])

b = np.array([5, np.sqrt(beta) * 5])

e = np.array([1, 1])

dynamics = ExchangeEconomy(Π, b, e)
p, c = dynamics.competitive_equilibrium()

print('Competitive equilibrium price vector:', p)
print('Competitive equilibrium allocation:', c)
```

```
Competitive equilibrium price vector: [4.  3.8]
Competitive equilibrium allocation: [1. 1.]
```

42.6.2 Risk and state-contingent claims

We study risk in the context of a **static** environment, meaning that there is only one period.

By **risk** we mean that an outcome is not known in advance, but that it is governed by a known probability distribution.

As an example, our consumer confronts **risk** means in particular that

- there are two states of nature, 1 and 2.

- the consumer knows that the probability that state 1 occurs is λ.

- the consumer knows that the probability that state 2 occurs is $(1 - \lambda)$.

Before the outcome is realized, the consumer's **expected utility** is

$$-\frac{1}{2}[\lambda(c_1 - b_1)^2 + (1 - \lambda)(c_2 - b_2)^2]$$

where

- c_1 is consumption in state 1

- c_2 is consumption in state 2

To capture these preferences we set

$$\Pi = \begin{bmatrix} \sqrt{\lambda} & 0 \\ 0 & \sqrt{1-\lambda} \end{bmatrix}$$

$$e = \begin{bmatrix} e_1 \\ e_2 \end{bmatrix}$$

$$b = \begin{bmatrix} \sqrt{\lambda}b_1 \\ \sqrt{1-\lambda}b_2 \end{bmatrix}$$

A consumer's endowment vector is

$$c = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$$

A price vector is

$$p = \begin{bmatrix} p_1 \\ p_2 \end{bmatrix}$$

where p_i is the price of one unit of consumption in state $i \in \{1, 2\}$.

The state-contingent goods being traded are often called **Arrow securities**.

Before the random state of the world i is realized, the consumer sells his/her state-contingent endowment bundle and purchases a state-contingent consumption bundle.

Trading such state-contingent goods is one way economists often model **insurance**.

We use the tricks described above to interpret c_1, c_2 as "Arrow securities" that are state-contingent claims to consumption goods.

Here is an instance of the risk economy:

```
prob = 0.2

Π = np.array([[np.sqrt(prob), 0],
              [0, np.sqrt(1 - prob)]])

b = np.array([np.sqrt(prob) * 5, np.sqrt(1 - prob) * 5])

e = np.array([1, 1])

risk = ExchangeEconomy(Π, b, e)
p, c = risk.competitive_equilibrium()

print('Competitive equilibrium price vector:', p)
print('Competitive equilibrium allocation:', c)
```

```
Competitive equilibrium price vector: [0.8 3.2]
Competitive equilibrium allocation: [1. 1.]
```

Exercise 42.6.1

Consider the instance above.

Please numerically study how each of the following cases affects the equilibrium prices and allocations:

- the consumer gets poorer,

- they like the first good more, or

- the probability that state 1 occurs is higher.

Hints. For each case choose some parameter $e, b,$ or λ different from the instance.

Solution to Exercise 42.6.1

First consider when the consumer is poorer.

Here we just decrease the endowment.

```python
risk.e = np.array([0.5, 0.5])

p, c = risk.competitive_equilibrium()

print('Competitive equilibrium price vector:', p)
print('Competitive equilibrium allocation:', c)
```

```
Competitive equilibrium price vector: [0.9 3.6]
Competitive equilibrium allocation: [0.5 0.5]
```

If the consumer likes the first (or second) good more, then we can set a larger bliss value for good 1.

```python
risk.b = np.array([np.sqrt(prob) * 6, np.sqrt(1 - prob) * 5])
p, c = risk.competitive_equilibrium()

print('Competitive equilibrium price vector:', p)
print('Competitive equilibrium allocation:', c)
```

```
Competitive equilibrium price vector: [1.1 3.6]
Competitive equilibrium allocation: [0.5 0.5]
```

Increase the probability that state 1 occurs.

```
prob = 0.8

Π = np.array([[np.sqrt(prob), 0],
              [0, np.sqrt(1 - prob)]])

b = np.array([np.sqrt(prob) * 5, np.sqrt(1 - prob) * 5])

e = np.array([1, 1])

risk = ExchangeEconomy(Π, b, e)
p, c = risk.competitive_equilibrium()

print('Competitive equilibrium price vector:', p)
print('Competitive equilibrium allocation:', c)
```

```
Competitive equilibrium price vector: [3.2 0.8]
Competitive equilibrium allocation: [1. 1.]
```

42.7 Economies with endogenous supplies of goods

Up to now we have described a pure exchange economy in which endowments of goods are exogenous, meaning that they are taken as given from outside the model.

42.7.1 Supply curve of a competitive firm

A competitive firm that can produce goods takes a price vector p as given and chooses a quantity q to maximize total revenue minus total costs.

The firm's total revenue equals $p^\top q$ and its total cost equals $C(q)$ where $C(q)$ is a total cost function

$$C(q) = h^\top q + \frac{1}{2}q^\top Jq$$

and J is a positive definite matrix.

So the firm's profits are

$$p^\top q - C(q) \tag{42.7}$$

An $n \times 1$ vector of **marginal costs** is

$$\frac{\partial C(q)}{\partial q} = h + Hq$$

where

$$H = \frac{1}{2}(J + J^\top)$$

The firm maximizes total profits by setting **marginal revenue to marginal costs**.

An $n \times 1$ vector of marginal revenues for the price-taking firm is $\frac{\partial p^\top q}{\partial q} = p$.

So **price equals marginal revenue** for our price-taking competitive firm.

This leads to the following **inverse supply curve** for the competitive firm:

$$p = h + Hq$$

42.7.2 Competitive equilibrium

To compute a competitive equilibrium for a production economy where demand curve is pinned down by the marginal utility of wealth μ, we first compute an allocation by solving a planning problem.

Then we compute the equilibrium price vector using the inverse demand or supply curve.

$\mu = 1$ **warmup**

As a special case, let's pin down a demand curve by setting the marginal utility of wealth $\mu = 1$.

Equating supply price to demand price and letting $q = c$ we get

$$p = h + Hc = \Pi^\top b - \Pi^\top \Pi c,$$

which implies the equilibrium quantity vector

$$c = (\Pi^\top \Pi + H)^{-1}(\Pi^\top b - h) \tag{42.8}$$

This equation is the counterpart of equilibrium quantity (7.3) for the scalar $n = 1$ model with which we began.

General $\mu \neq 1$ **case**

Now let's extend the preceding analysis to a more general case by allowing $\mu \neq 1$.

Then the inverse demand curve is

$$p = \mu^{-1}[\Pi^\top b - \Pi^\top \Pi c] \tag{42.9}$$

Equating this to the inverse supply curve, letting $q = c$ and solving for c gives

$$c = [\Pi^\top \Pi + \mu H]^{-1}[\Pi^\top b - \mu h] \tag{42.10}$$

42.7.3 Implementation

A Production Economy will consist of

- a single **person** that we'll interpret as a representative consumer

- a single set of **production costs**

- a multiplier μ that weights "consumers" versus "producers" in a planner's welfare function, as described above in the main text

- an $n \times 1$ vector p of competitive equilibrium prices

- an $n \times 1$ vector c of competitive equilibrium quantities

- **consumer surplus**

- **producer surplus**

Here we define a class `ProductionEconomy`.

```
class ProductionEconomy:

    def __init__(self,
                 Π,
                 b,
                 h,
                 J,
                 μ):
        """
        Set up the environment for a production economy

        Args:
            Π (np.ndarray): matrix of substitution
            b (np.array): bliss points
            h (np.array): h in cost func
            J (np.ndarray): J in cost func
            μ (float): welfare weight of the corresponding planning problem
        """
        self.n = len(b)
        self.Π, self.b, self.h, self.J, self.μ = Π, b, h, J, μ

    def competitive_equilibrium(self):
        """
        Compute a competitive equilibrium of the production economy
        """
        Π, b, h, μ, J = self.Π, self.b, self.h, self.μ, self.J
```

(continues on next page)

```
        H = .5 * (J + J.T)


        # allocation
        c = inv(Π.T @ Π + μ * H) @ (Π.T @ b - μ * h)


        # price
        p = 1 / μ * (Π.T @ b - Π.T @ Π @ c)


        # check non-satiation
        if any(Π @ c - b >= 0):
            raise Exception('invalid result: set bliss points further away')


        return c, p


    def compute_surplus(self):
        """
        Compute consumer and producer surplus for single good case
        """
        if self.n != 1:
            raise Exception('not single good')
        h, J, Π, b, μ = self.h.item(), self.J.item(), self.Π.item(), self.b.item(),␣
→self.μ
        H = J


        # supply/demand curve coefficients
        s0, s1 = h, H
        d0, d1 = 1 / μ * Π * b, 1 / μ * Π**2


        # competitive equilibrium
        c, p = self.competitive_equilibrium()


        # calculate surplus
        c_surplus = d0 * c - .5 * d1 * c**2 - p * c
        p_surplus = p * c - s0 * c - .5 * s1 * c**2


        return c_surplus, p_surplus
```

Then define a function that plots demand and supply curves and labels surpluses and equilibrium.

Example: single agent with one good and production

Now let's construct an example of a production economy with one good.

To do this we

- specify a single **person** and a **cost curve** in a way that let's us replicate the simple single-good supply demand example with which we started

- compute equilibrium p and c and consumer and producer surpluses

- draw graphs of both surpluses

- do experiments in which we shift b and watch what happens to p, c.

```python
Π = np.array([[1]])   # the matrix now is a singleton
b = np.array([10])
h = np.array([0.5])
J = np.array([[1]])
μ = 1


PE = ProductionEconomy(Π, b, h, J, μ)
c, p = PE.competitive_equilibrium()


print('Competitive equilibrium price:', p.item())
print('Competitive equilibrium allocation:', c.item())


# plot
plot_competitive_equilibrium(PE)
```

```
Competitive equilibrium price: 5.25
Competitive equilibrium allocation: 4.75
```

```
c_surplus, p_surplus = PE.compute_surplus()

print('Consumer surplus:', c_surplus.item())
print('Producer surplus:', p_surplus.item())
```

```
Consumer surplus: 11.28125
Producer surplus: 11.28125
```

Let's give the consumer a lower welfare weight by raising μ.

```
PE.μ = 2
c, p = PE.competitive_equilibrium()

print('Competitive equilibrium price:', p.item())
print('Competitive equilibrium allocation:', c.item())

# plot
plot_competitive_equilibrium(PE)
```

```
Competitive equilibrium price: 3.5
Competitive equilibrium allocation: 3.0
```
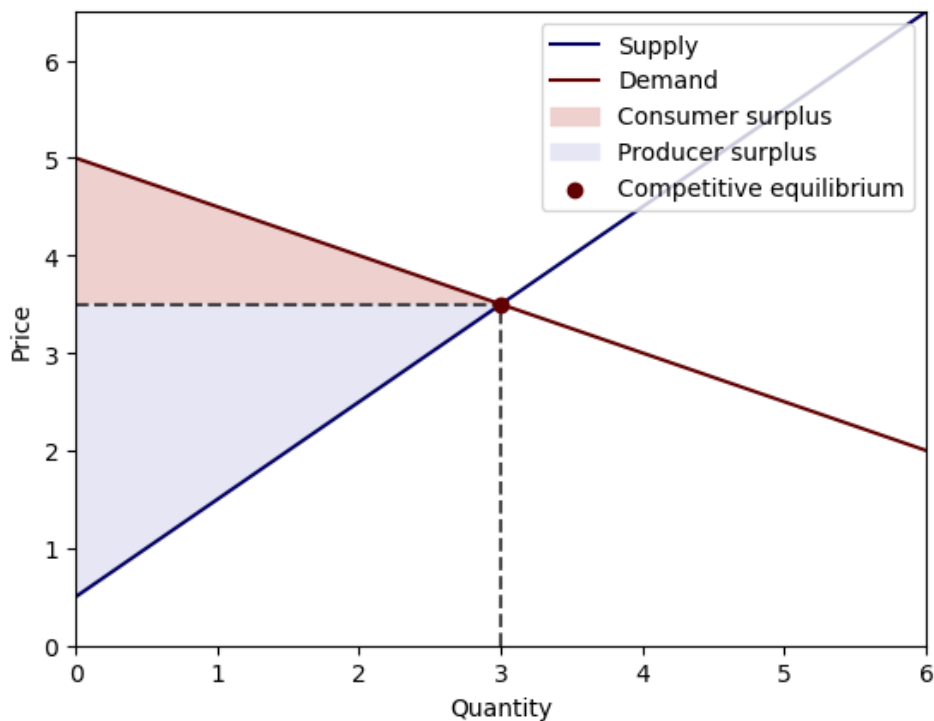
```
c_surplus, p_surplus = PE.compute_surplus()

print('Consumer surplus:', c_surplus.item())
print('Producer surplus:', p_surplus.item())
```

```
Consumer surplus: 2.25
Producer surplus: 4.5
```

Now we change the bliss point so that the consumer derives more utility from consumption.

```
PE.μ = 1
PE.b = PE.b * 1.5
c, p = PE.competitive_equilibrium()

print('Competitive equilibrium price:', p.item())
print('Competitive equilibrium allocation:', c.item())

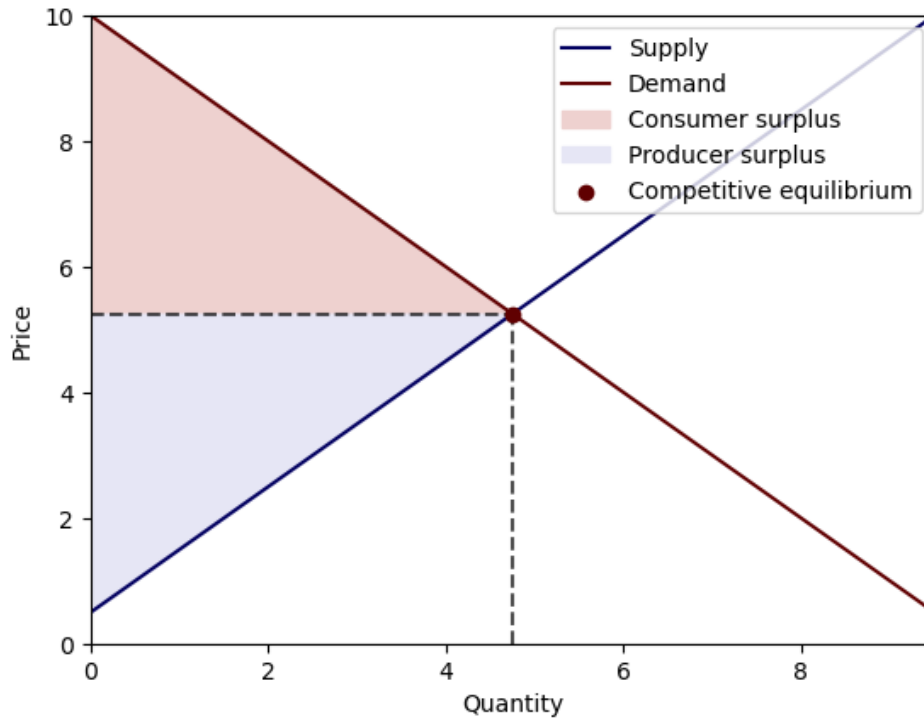# plot
plot_competitive_equilibrium(PE)
```

```
Competitive equilibrium price: 7.75
Competitive equilibrium allocation: 7.25
```
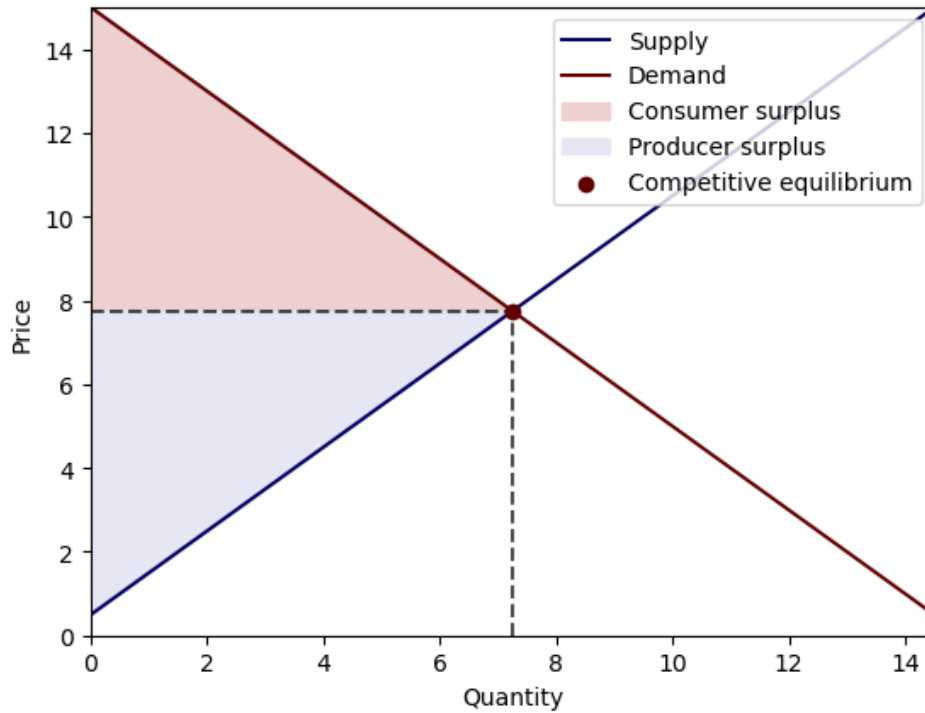
This raises both the equilibrium price and quantity.

Example: single agent two-good economy with production

- we'll do some experiments like those above

- we can do experiments with a **diagonal** Π and also with a **non-diagonal** Π matrices to study how cross-slopes affect responses of p and c to various shifts in b (TODO)

```
Π = np.array([[1, 0],
              [0, 1]])

b = np.array([10, 10])

h = np.array([0.5, 0.5])

J = np.array([[1, 0.5],
              [0.5, 1]])
μ = 1

PE = ProductionEconomy(Π, b, h, J, μ)
c, p = PE.competitive_equilibrium()

print('Competitive equilibrium price:', p)
```

```
print('Competitive equilibrium allocation:', c)
```

```
Competitive equilibrium price: [6.2 6.2]
Competitive equilibrium allocation: [3.8 3.8]
```

```
PE.b = np.array([12, 10])

c, p = PE.competitive_equilibrium()

print('Competitive equilibrium price:', p)
print('Competitive equilibrium allocation:', c)
```

```
Competitive equilibrium price: [7.13333333 6.46666667]
Competitive equilibrium allocation: [4.86666667 3.53333333]
```

```
PE.Π = np.array([[1, 0.5],
                 [0.5, 1]])

PE.b = np.array([10, 10])

c, p = PE.competitive_equilibrium()

print('Competitive equilibrium price:', p)
print('Competitive equilibrium allocation:', c)
```

```
Competitive equilibrium price: [6.3 6.3]
Competitive equilibrium allocation: [3.86666667 3.86666667]
```

```
PE.b = np.array([12, 10])
c, p = PE.competitive_equilibrium()

print('Competitive equilibrium price:', p)
print('Competitive equilibrium allocation:', c)
```

```
Competitive equilibrium price: [7.23333333 6.56666667]
Competitive equilibrium allocation: [4.93333333 3.6       ]
```

42.7.4 Digression: a supplier who is a monopolist

A competitive firm is a **price-taker** who regards the price and therefore its marginal revenue as being beyond its control.

A monopolist knows that it has no competition and can influence the price and its marginal revenue by setting quantity.

A monopolist takes a **demand curve** and not the **price** as beyond its control.

Thus, instead of being a price-taker, a monopolist sets prices to maximize profits subject to the inverse demand curve (**??**).

So the monopolist's total profits as a function of its output q is

$$[\mu^{-1}\Pi^\top(b - \Pi q)]^\top q - h^\top q - \frac{1}{2}q^\top Jq \tag{42.11}$$

After finding first-order necessary conditions for maximizing monopoly profits with respect to q and solving them for q, we find that the monopolist sets

$$q = (H + 2\mu^{-1}\Pi^\top\Pi)^{-1}(\mu^{-1}\Pi^\top b - h) \tag{42.12}$$

We'll soon see that a monopolist sets a **lower output** q than does either a

- planner who chooses q to maximize social welfare

- a competitive equilibrium

Exercise 42.7.1

Please verify the monopolist's supply curve (**??**).

42.7.5 A monopolist

Let's consider a monopolist supplier.

We have included a method in our `ProductionEconomy` class to compute an equilibrium price and allocation when the supplier is a monopolist.

Since the supplier now has the price-setting power

- we first compute the optimal quantity that solves the monopolist's profit maximization problem.

- Then we back out an equilibrium price from the consumer's inverse demand curve.

Next, we use a graph for the single good case to illustrate the difference between a competitive equilibrium and an equilibrium with a monopolist supplier.

Recall that in a competitive equilibrium, a price-taking supplier equates marginal revenue p to marginal cost $h + Hq$.

This yields a competitive producer's inverse supply curve.

A monopolist's marginal revenue is not constant but instead is a non-trivial function of the quantity it sets.

The monopolist's marginal revenue is

$$MR(q) = -2\mu^{-1}\Pi^{\top}\Pi q + \mu^{-1}\Pi^{\top}b,$$

which the monopolist equates to its marginal cost.

The plot indicates that the monopolist's sets output lower than either the competitive equilibrium quantity.

In a single good case, this equilibrium is associated with a higher price of the good.

```python
class Monopoly(ProductionEconomy):

    def __init__(self,
                 Π,
                 b,
                 h,
                 J,
                 μ):
        """
        Inherit all properties and methods from class ProductionEconomy
        """
        super().__init__(Π, b, h, J, μ)


    def equilibrium_with_monopoly(self):
        """
        Compute the equilibrium price and allocation when there is a monopolist␣
 ↪supplier
        """
        Π, b, h, μ, J = self.Π, self.b, self.h, self.μ, self.J
        H = .5 * (J + J.T)

        # allocation
        q = inv(μ * H + 2 * Π.T @ Π) @ (Π.T @ b - μ * h)

        # price
        p = 1 / μ * (Π.T @ b - Π.T @ Π @ q)

        if any(Π @ q - b >= 0):
            raise Exception('invalid result: set bliss points further away')

        return q, p
```

Define a function that plots the demand, marginal cost and marginal revenue curves with surpluses and equilibrium labelled.

A multiple good example

Let's compare competitive equilibrium and monopoly outcomes in a multiple goods economy.

```python
Π = np.array([[1, 0],
              [0, 1.2]])

b = np.array([10, 10])

h = np.array([0.5, 0.5])

J = np.array([[1, 0.5],
              [0.5, 1]])
μ = 1

M = Monopoly(Π, b, h, J, μ)
c, p = M.competitive_equilibrium()
q, pm = M.equilibrium_with_monopoly()

print('Competitive equilibrium price:', p)
print('Competitive equilibrium allocation:', c)

print('Equilibrium with monopolist supplier price:', pm)
print('Equilibrium with monopolist supplier allocation:', q)
```

```
Competitive equilibrium price: [6.23542117 6.32397408]
Competitive equilibrium allocation: [3.76457883 3.94168467]
Equilibrium with monopolist supplier price: [7.26865672 8.23880597]
Equilibrium with monopolist supplier allocation: [2.73134328 2.6119403 ]
```

A single-good example

```python
Π = np.array([[1]])   # the matrix now is a singleton
b = np.array([10])
h = np.array([0.5])
J = np.array([[1]])
μ = 1
```

```
M = Monopoly(Π, b, h, J, μ)
c, p = M.competitive_equilibrium()
q, pm = M.equilibrium_with_monopoly()

print('Competitive equilibrium price:', p.item())
print('Competitive equilibrium allocation:', c.item())

print('Equilibrium with monopolist supplier price:', pm.item())
print('Equilibrium with monopolist supplier allocation:', q.item())

# plot
plot_monopoly(M)
```

```
Competitive equilibrium price: 5.25
Competitive equilibrium allocation: 4.75
Equilibrium with monopolist supplier price: 6.833333333333334
Equilibrium with monopolist supplier allocation: 3.1666666666666665
```
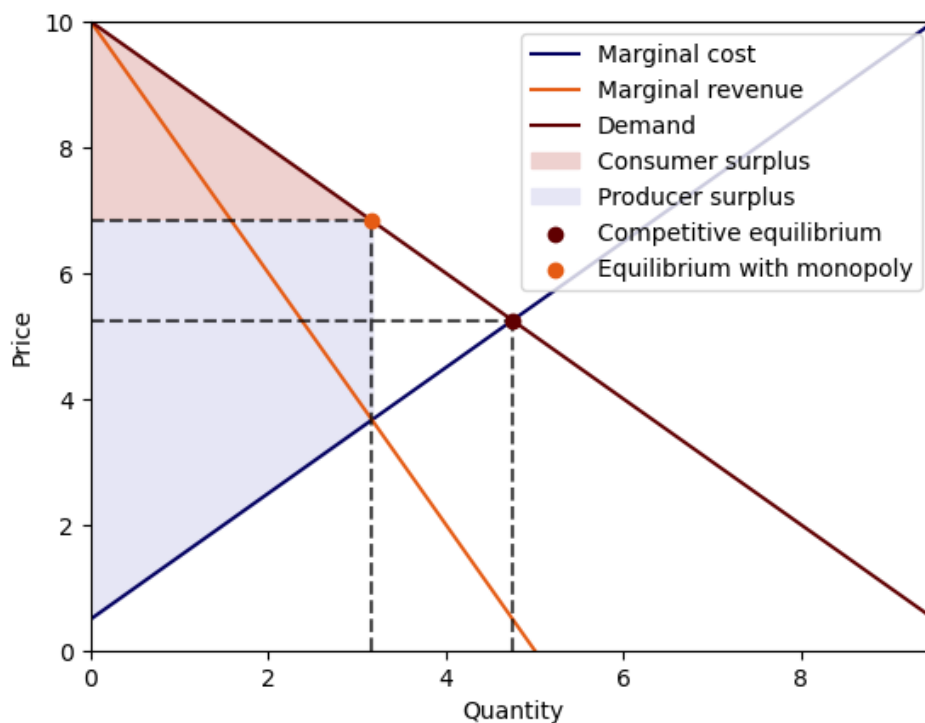
42.8 Multi-good welfare maximization problem

Our welfare maximization problem –also sometimes called a social planning problem –is to choose c to maximize

$$-\frac{1}{2}\mu^{-1}(\Pi c - b)^{\top}(\Pi c - b)$$

minus the area under the inverse supply curve, namely,

$$hc + \frac{1}{2}c^{\top}Jc$$

So the welfare criterion is

$$-\frac{1}{2}\mu^{-1}(\Pi c - b)^{\top}(\Pi c - b) - hc - \frac{1}{2}c^{\top}Jc$$

In this formulation, μ is a parameter that describes how the planner weighs interests of outside suppliers and our representative consumer.

The first-order condition with respect to c is

$$-\mu^{-1}\Pi^{\top}\Pi c + \mu^{-1}\Pi^{\top}b - h - Hc = 0$$

which implies (**??**).

Thus, as for the single-good case, with multiple goods a competitive equilibrium quantity vector solves a planning problem.

(This is another version of the first welfare theorem.)

We can deduce a competitive equilibrium price vector from either

- the inverse demand curve, or

- the inverse supply curve

MARKET EQUILIBRIUM WITH HETEROGENEITY

43.1 Overview

In the *previous lecture*, we studied competitive equilibria in an economy with many goods.

While the results of the study were informative, we used a strong simplifying assumption: all of the agents in the economy are identical.

In the real world, households, firms and other economic agents differ from one another along many dimensions.

In this lecture, we introduce heterogeneity across consumers by allowing their preferences and endowments to differ.

We will examine competitive equilibrium in this setting.

We will also show how a "representative consumer" can be constructed.

Here are some imports:

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

43.2 An simple example

Let's study a simple example of **pure exchange** economy without production.

There are two consumers who differ in their endowment vectors e_i and their bliss-point vectors b_i for $i = 1, 2$.

The total endowment is $e_1 + e_2$.

A competitive equilibrium requires that

$$c_1 + c_2 = e_1 + e_2$$

Assume the demand curves

$$c_i = (\Pi^\top \Pi)^{-1}(\Pi^\top b_i - \mu_i p)$$

Competitive equilibrium then requires that

$$e_1 + e_2 = (\Pi^\top \Pi)^{-1}(\Pi^\top (b_1 + b_2) - (\mu_1 + \mu_2)p)$$

which, after a line or two of linear algebra, implies that

$$(\mu_1 + \mu_2)p = \Pi^\top (b_1 + b_2) - \Pi^\top \Pi (e_1 + e_2) \tag{43.1}$$

We can normalize prices by setting $\mu_1 + \mu_2 = 1$ and then solving

$$\mu_i(p, e) = \frac{p^\top (\Pi^{-1} b_i - e_i)}{p^\top (\Pi^\top \Pi)^{-1} p} \tag{43.2}$$

for $\mu_i, i = 1, 2$.

Exercise 43.2.1

Show that, up to normalization by a positive scalar, the same competitive equilibrium price vector that you computed in the preceding two-consumer economy would prevail in a single-consumer economy in which a single **representative consumer** has utility function

$$-.5(\Pi c - b)^\top (\Pi c - b)$$

and endowment vector e, where

$$b = b_1 + b_2$$

and

$$e = e_1 + e_2.$$

43.3 Pure exchange economy

Let's further explore a pure exchange economy with n goods and m people.

43.3.1 Competitive equilibrium

We'll compute a competitive equilibrium.

To compute a competitive equilibrium of a pure exchange economy, we use the fact that

- Relative prices in a competitive equilibrium are the same as those in a special single person or representative consumer economy with preference Π and $b = \sum_i b_i$, and endowment $e = \sum_i e_i$.

We can use the following steps to compute a competitive equilibrium:

- First we solve the single representative consumer economy by normalizing $\mu = 1$. Then, we renormalize the price vector by using the first consumption good as a numeraire.

- Next we use the competitive equilibrium prices to compute each consumer's marginal utility of wealth:

$$\mu_i = \frac{-W_i + p^\top \left(\Pi^{-1} b_i - e_i \right)}{p^\top (\Pi^\top \Pi)^{-1} p}$$

- Finally we compute a competitive equilibrium allocation by using the demand curves:

$$c_i = \Pi^{-1} b_i - (\Pi^\top \Pi)^{-1} \mu_i p$$

43.3.2 Designing some Python code

Below we shall construct a Python class with the following attributes:

- **Preferences** in the form of

  - an $n \times n$ positive definite matrix Π

  - an $n \times 1$ vector of bliss points b

- **Endowments** in the form of

  - an $n \times 1$ vector e

  - a scalar "wealth" W with default value 0

The class will include a test to make sure that $b \gg \Pi e$ and raise an exception if it is violated (at some threshold level we'd have to specify).

- **A Person** in the form of a pair that consists of

  - **Preferences** and **Endowments**

- **A Pure Exchange Economy** will consist of

  - a collection of m **persons**

    * $m = 1$ for our single-agent economy

    * $m = 2$ for our illustrations of a pure exchange economy

  - an equilibrium price vector p (normalized somehow)

  - an equilibrium allocation $c_1, c_2, \ldots, c_m$ –a collection of m vectors of dimension $n \times 1$

Now let's proceed to code.

```python
class ExchangeEconomy:
    def __init__(self,
                 Π,
                 bs,
                 es,
                 Ws=None,
                 thres=1.5):
        """
        Set up the environment for an exchange economy

        Args:
            Π (np.array): shared matrix of substitution
            bs (list): all consumers' bliss points
            es (list): all consumers' endowments
            Ws (list): all consumers' wealth
            thres (float): a threshold set to test b >> Pi e violated
        """
        n, m = Π.shape[0], len(bs)

        # check non-satiation
        for b, e in zip(bs, es):
            if np.min(b / np.max(Π @ e)) <= thres:
                raise Exception('set bliss points further away')

        if Ws == None:
            Ws = np.zeros(m)
        else:
            if sum(Ws) != 0:
                raise Exception('invalid wealth distribution')

        self.Π, self.bs, self.es, self.Ws, self.n, self.m = Π, bs, es, Ws, n, m

    def competitive_equilibrium(self):
        """
        Compute the competitive equilibrium prices and allocation
        """
        Π, bs, es, Ws = self.Π, self.bs, self.es, self.Ws
        n, m = self.n, self.m
        slope_dc = inv(Π.T @ Π)
        Π_inv = inv(Π)

        # aggregate
        b = sum(bs)
        e = sum(es)
```

```python
        # compute price vector with mu=1 and renormalize
        p = Π.T @ b - Π.T @ Π @ e
        p = p / p[0]

        # compute marginal utility of wealth
        μ_s = []
        c_s = []
        A = p.T @ slope_dc @ p

        for i in range(m):
            μ_i = (-Ws[i] + p.T @ (Π_inv @ bs[i] - es[i])) / A
            c_i = Π_inv @ bs[i] - μ_i * slope_dc @ p
            μ_s.append(μ_i)
            c_s.append(c_i)

        for c_i in c_s:
            if any(c_i < 0):
                print('allocation: ', c_s)
                raise Exception('negative allocation: equilibrium does not exist')

        return p, c_s, μ_s
```

43.4 Implementation

Next we use the class `ExchangeEconomy` defined above to study

- a two-person economy without production,

- a dynamic economy, and

- an economy with risk and arrow securities.

43.4.1 Two-person economy without production

Here we study how competitive equilibrium p, c_1, c_2 respond to different b_i and e_i, $i \in \{1, 2\}$.

```python
Π = np.array([[1, 0],
              [0, 1]])


bs = [np.array([5, 5]),   # first consumer's bliss points
```

```
      np.array([5, 5])]  # second consumer's bliss points

es = [np.array([0, 2]),  # first consumer's endowment
      np.array([2, 0])]  # second consumer's endowment

EE = ExchangeEconomy(Π, bs, es)
p, c_s, μ_s = EE.competitive_equilibrium()

print('Competitive equilibrium price vector:', p)
print('Competitive equilibrium allocation:', c_s)
```

```
Competitive equilibrium price vector: [1. 1.]
Competitive equilibrium allocation: [array([1., 1.]), array([1., 1.])]
```

What happens if the first consumer likes the first good more and the second consumer likes the second good more?

```
EE.bs = [np.array([6, 5]),  # first consumer's bliss points
         np.array([5, 6])]  # second consumer's bliss points

p, c_s, μ_s = EE.competitive_equilibrium()

print('Competitive equilibrium price vector:', p)
print('Competitive equilibrium allocation:', c_s)
```

```
Competitive equilibrium price vector: [1. 1.]
Competitive equilibrium allocation: [array([1.5, 0.5]), array([0.5, 1.5])]
```

Let the first consumer be poorer.

```
EE.es = [np.array([0.5, 0.5]),  # first consumer's endowment
         np.array([1, 1])]  # second consumer's endowment

p, c_s, μ_s = EE.competitive_equilibrium()

print('Competitive equilibrium price vector:', p)
print('Competitive equilibrium allocation:', c_s)
```

```
Competitive equilibrium price vector: [1. 1.]
Competitive equilibrium allocation: [array([1., 0.]), array([0.5, 1.5])]
```

Now let's construct an autarky (i.e., no-trade) equilibrium.

```
EE.bs = [np.array([4, 6]),  # first consumer's bliss points
        np.array([6, 4])]  # second consumer's bliss points


EE.es = [np.array([0, 2]),  # first consumer's endowment
        np.array([2, 0])]  # second consumer's endowment


p, c_s, μ_s = EE.competitive_equilibrium()


print('Competitive equilibrium price vector:', p)
print('Competitive equilibrium allocation:', c_s)
```

```
Competitive equilibrium price vector: [1. 1.]
Competitive equilibrium allocation: [array([0., 2.]), array([2., 0.])]
```

Now let's redistribute endowments before trade.

```
bs = [np.array([5, 5]),  # first consumer's bliss points
      np.array([5, 5])]  # second consumer's bliss points


es = [np.array([1, 1]),  # first consumer's endowment
      np.array([1, 1])]  # second consumer's endowment


Ws = [0.5, -0.5]
EE_new = ExchangeEconomy(Π, bs, es, Ws)
p, c_s, μ_s = EE_new.competitive_equilibrium()


print('Competitive equilibrium price vector:', p)
print('Competitive equilibrium allocation:', c_s)
```

```
Competitive equilibrium price vector: [1. 1.]
Competitive equilibrium allocation: [array([1.25, 1.25]), array([0.75, 0.75])]
```

43.4.2 A dynamic economy

Now let's use the tricks described above to study a dynamic economy, one with two periods.

```
beta = 0.95


Π = np.array([[1, 0],
              [0, np.sqrt(beta)]])
```

```
bs = [np.array([5, np.sqrt(beta) * 5])]

es = [np.array([1, 1])]

EE_DE = ExchangeEconomy(Π, bs, es)
p, c_s, μ_s = EE_DE.competitive_equilibrium()

print('Competitive equilibrium price vector:', p)
print('Competitive equilibrium allocation:', c_s)
```

```
Competitive equilibrium price vector: [1.   0.95]
Competitive equilibrium allocation: [array([1., 1.])]
```

43.4.3 Risk economy with arrow securities

We use the tricks described above to interpret c_1, c_2 as "Arrow securities"that are state-contingent claims to consumption goods.

```
prob = 0.7

Π = np.array([[np.sqrt(prob), 0],
              [0, np.sqrt(1 - prob)]])

bs = [np.array([np.sqrt(prob) * 5, np.sqrt(1 - prob) * 5]),
      np.array([np.sqrt(prob) * 5, np.sqrt(1 - prob) * 5])]

es = [np.array([1, 0]),
      np.array([0, 1])]

EE_AS = ExchangeEconomy(Π, bs, es)
p, c_s, μ_s = EE_AS.competitive_equilibrium()

print('Competitive equilibrium price vector:', p)
print('Competitive equilibrium allocation:', c_s)
```

```
Competitive equilibrium price vector: [1.         0.42857143]
Competitive equilibrium allocation: [array([0.7, 0.7]), array([0.3, 0.3])]
```

43.5 Deducing a representative consumer

In the class of multiple consumer economies that we are studying here, it turns out that there exists a single **representative consumer** whose preferences and endowments can be deduced from lists of preferences and endowments for separate individual consumers.

Consider a multiple consumer economy with initial distribution of wealth W_i satisfying $\sum_i W_i = 0$

We allow an initial redistribution of wealth.

We have the following objects

- The demand curve:

$$c_i = \Pi^{-1} b_i - (\Pi^\top \Pi)^{-1} \mu_i p$$

- The marginal utility of wealth:

$$\mu_i = \frac{-W_i + p^\top \left(\Pi^{-1} b_i - e_i \right)}{p^\top (\Pi^\top \Pi)^{-1} p}$$

- Market clearing:

$$\sum c_i = \sum e_i$$

Denote aggregate consumption $\sum_i c_i = c$ and $\sum_i \mu_i = \mu$.

Market clearing requires

$$\Pi^{-1} \left(\sum_i b_i \right) - (\Pi^\top \Pi)^{-1} p \left(\sum_i \mu_i \right) = \sum_i e_i$$

which, after a few steps, leads to

$$p = \mu^{-1} \left(\Pi^\top b - \Pi^\top \Pi e \right)$$

where

$$\mu = \sum_i \mu_i = \frac{0 + p^\top \left(\Pi^{-1} b - e \right)}{p^\top (\Pi^\top \Pi)^{-1} p}.$$

Now consider the representative consumer economy specified above.

Denote the marginal utility of wealth of the representative consumer by $\tilde{\mu}$.

The demand function is

$$c = \Pi^{-1} b - (\Pi^\top \Pi)^{-1} \tilde{\mu} p$$

Substituting this into the budget constraint gives

$$\tilde{\mu} = \frac{p^\top \left(\Pi^{-1} b - e \right)}{p^\top (\Pi^\top \Pi)^{-1} p}$$

In an equilibrium $c = e$, so

$$p = \tilde{\mu}^{-1}(\Pi^\top b - \Pi^\top \Pi e)$$

Thus, we have verified that, up to the choice of a numeraire in which to express absolute prices, the price vector in our representative consumer economy is the same as that in an underlying economy with multiple consumers.

Part XIII

Estimation

SIMPLE LINEAR REGRESSION MODEL

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

The simple regression model estimates the relationship between two variables x_i and y_i

$$y_i = \alpha + \beta x_i + \epsilon_i, i = 1, 2, ..., N$$

where ϵ_i represents the error between the line of best fit and the sample values for y_i given x_i.

Our goal is to choose values for α and β to build a line of "best" fit for some data that is available for variables x_i and y_i.

Let us consider a simple dataset of 10 observations for variables x_i and y_i:

	y_i	x_i
1	2000	32
2	1000	21
3	1500	24
4	2500	35
5	500	10
6	900	11
7	1100	22
8	1500	21
9	1800	27
10	250	2

Let us think about y_i as sales for an ice-cream cart, while x_i is a variable that records the day's temperature in Celsius.

```
x = [32, 21, 24, 35, 10, 11, 22, 21, 27, 2]
y = [2000,1000,1500,2500,500,900,1100,1500,1800, 250]
df = pd.DataFrame([x,y]).T
df.columns = ['X', 'Y']
df
```

```
      X      Y
0    32   2000
1    21   1000
2    24   1500
3    35   2500
4    10    500
5    11    900
6    22   1100
7    21   1500
8    27   1800
9     2    250
```

We can use a scatter plot of the data to see the relationship between y_i (ice-cream sales in dollars (\$'s)) and x_i (degrees Celsius).

```
ax = df.plot(
    x='X',
    y='Y',
    kind='scatter',
    ylabel='Ice-cream sales ($\'s)',
    xlabel='Degrees celcius'
)
```

as you can see the data suggests that more ice-cream is typically sold on hotter days.

To build a linear model of the data we need to choose values for α and β that represents a line of "best" fit such that

$$\hat{y}_i = \hat{\alpha} + \hat{\beta} x_i$$

Let's start with $\alpha = 5$ and $\beta = 10$

```
α = 5
β = 10
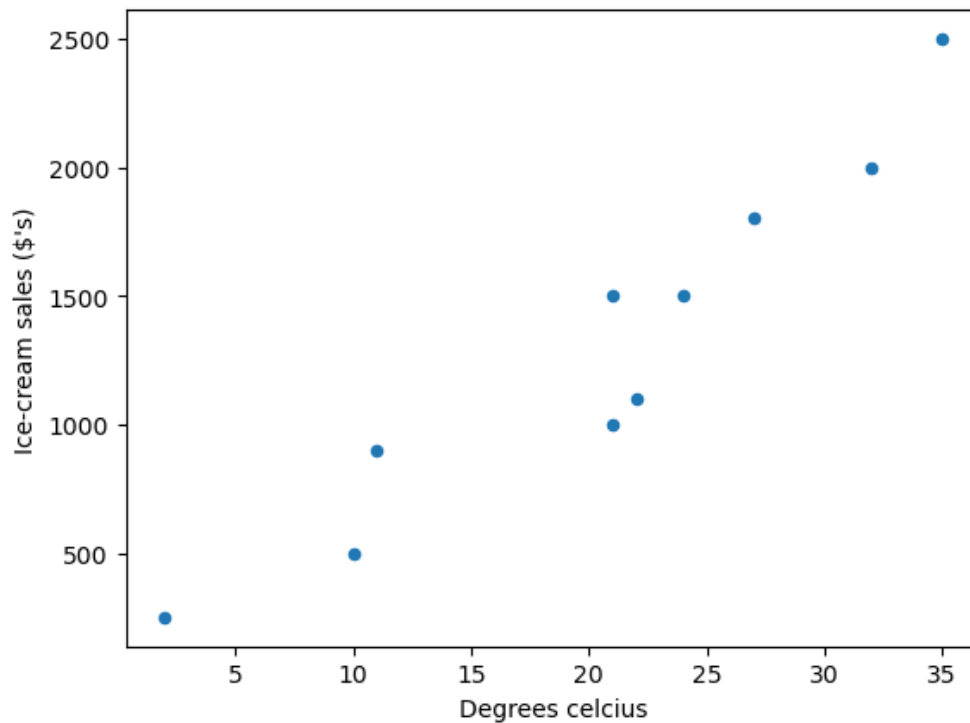df['Y_hat'] = α + β * df['X']
```

Fig. 44.1: Scatter plot

```
fig, ax = plt.subplots()
ax = df.plot(x='X',y='Y', kind='scatter', ax=ax)
ax = df.plot(x='X',y='Y_hat', kind='line', ax=ax)
plt.show()
```

We can see that this model does a poor job of estimating the relationship.

We can continue to guess and iterate towards a line of "best"fit by adjusting the parameters

```
β = 100
df['Y_hat'] = α + β * df['X']
```

```
fig, ax = plt.subplots()
ax = df.plot(x='X',y='Y', kind='scatter', ax=ax)
ax = df.plot(x='X',y='Y_hat', kind='line', ax=ax)
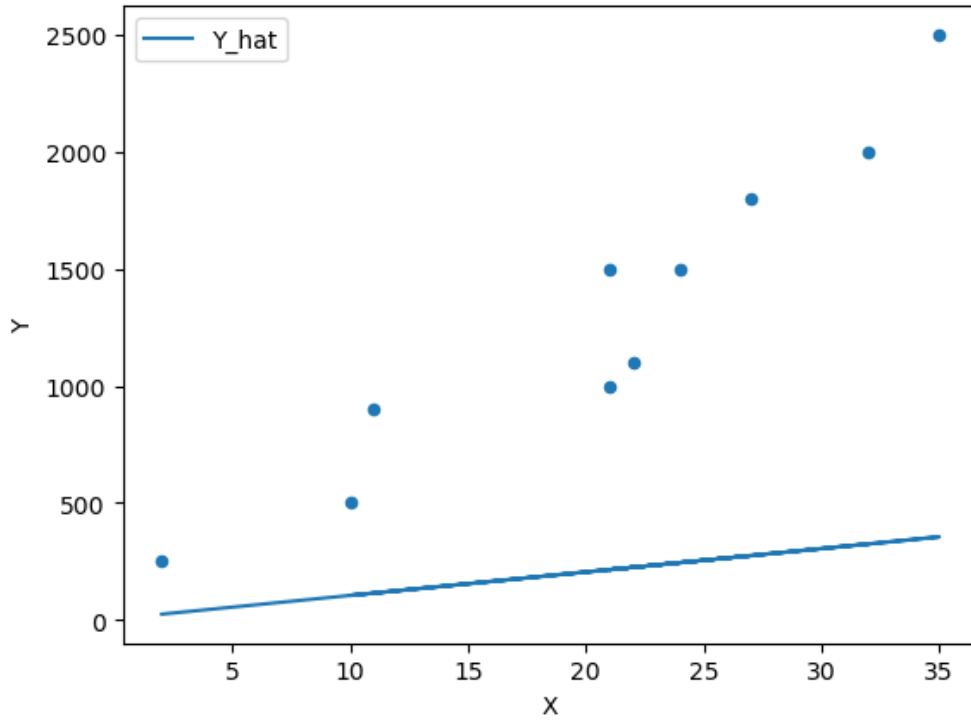plt.show()
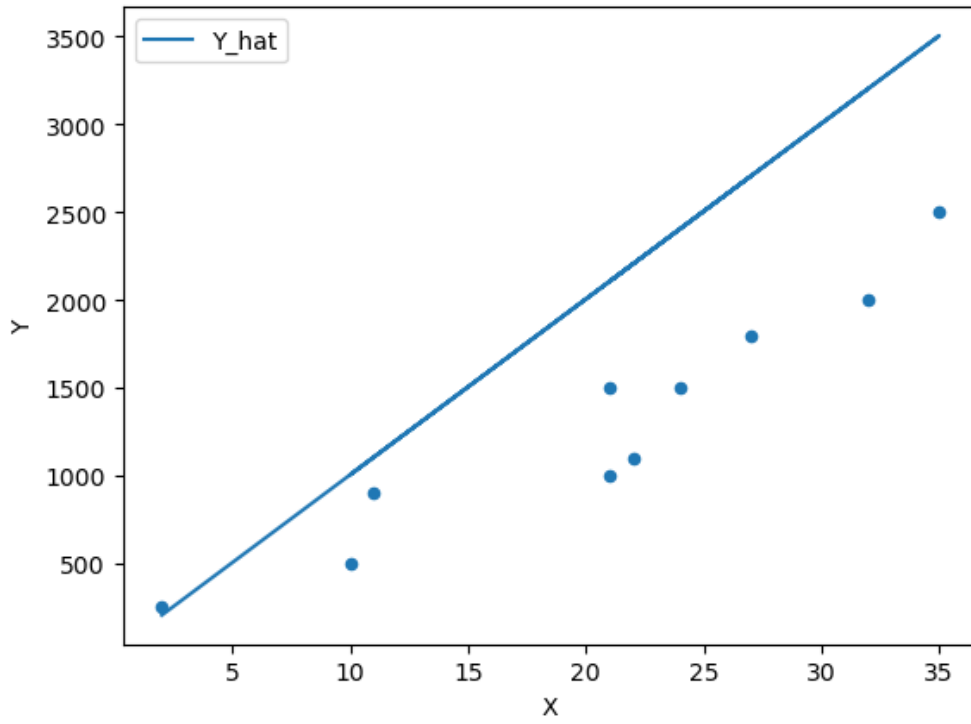```

Fig. 44.2: Scatter plot with a line of fit



Fig. 44.3: Scatter plot with a line of fit #2

```
β = 65
df['Y_hat'] = α + β * df['X']
```

```
fig, ax = plt.subplots()
ax = df.plot(x='X',y='Y', kind='scatter', ax=ax)
ax = df.plot(x='X',y='Y_hat', kind='line', ax=ax, color='g')
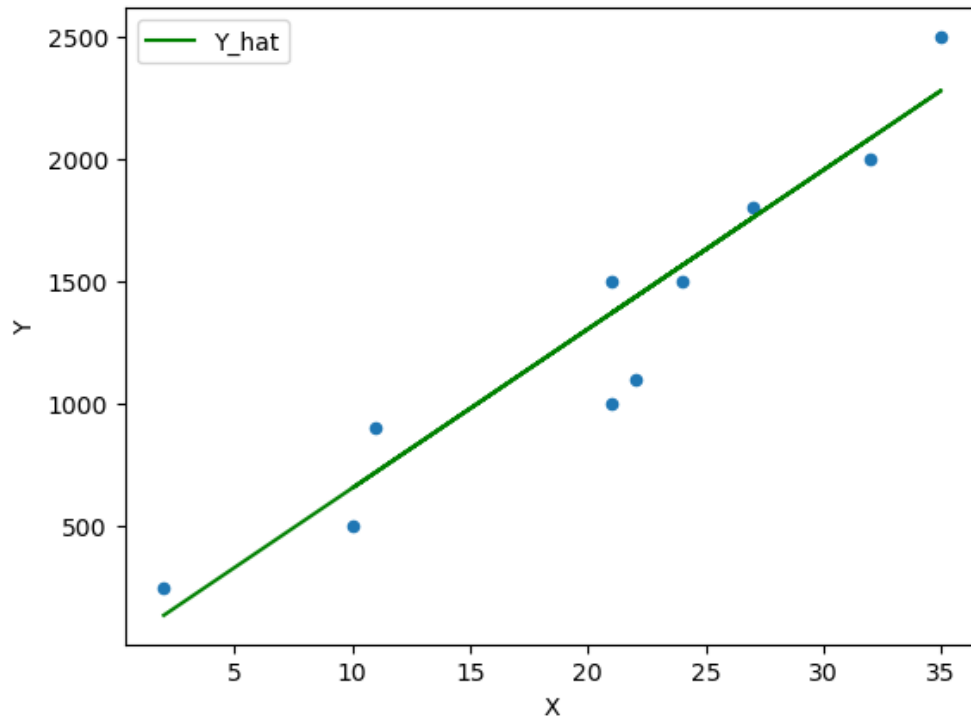plt.show()
```


Fig. 44.4: Scatter plot with a line of fit #3

However we need to think about formalizing this guessing process by thinking of this problem as an optimization problem.

Let's consider the error ϵ_i and define the difference between the observed values y_i and the estimated values $\hat{y}_i$ which we will call the residuals

$$\hat{e}_i = y_i - \hat{y}_i$$
$$= y_i - \hat{\alpha} - \hat{\beta}x_i$$

```
df['error'] = df['Y_hat'] - df['Y']
```

```
df
```

```
      X      Y  Y_hat  error
0    32   2000   2085     85
1    21   1000   1370    370
2    24   1500   1565     65
3    35   2500   2280   -220
4    10    500    655    155
5    11    900    720   -180
6    22   1100   1435    335
7    21   1500   1370   -130
8    27   1800   1760    -40
9     2    250    135   -115
```

```python
fig, ax = plt.subplots()
ax = df.plot(x='X',y='Y', kind='scatter', ax=ax)
ax = df.plot(x='X',y='Y_hat', kind='line', ax=ax, color='g')
plt.vlines(df['X'], df['Y_hat'], df['Y'], color='r')
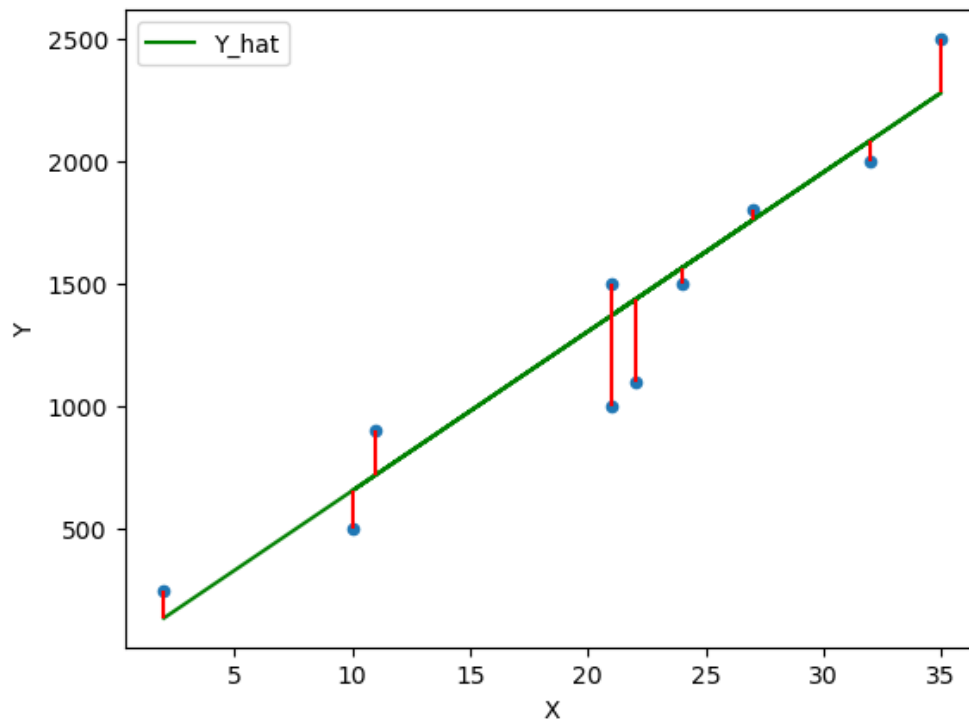plt.show()
```


Fig. 44.5: Plot of the residuals

The Ordinary Least Squares (OLS) method chooses α and β in such a way that **minimizes** the sum of the squared residuals (SSR).

$$\min_{\alpha,\beta} \sum_{i=1}^{N} \hat{e}_i^2 = \min_{\alpha,\beta} \sum_{i=1}^{N} (y_i - \alpha - \beta x_i)^2$$

Let's call this a cost function

$$C = \sum_{i=1}^{N} (y_i - \alpha - \beta x_i)^2$$

that we would like to minimize with parameters α and β.

44.1 How does error change with respect to α and β

Let us first look at how the total error changes with respect to β (holding the intercept α constant)

We know from *the next section* the optimal values for α and β are:

```
β_optimal = 64.38
α_optimal = -14.72
```

We can then calculate the error for a range of β values

```
errors = {}
for β in np.arange(20,100,0.5):
    errors[β] = abs((α_optimal + β * df['X']) - df['Y']).sum()
```

Plotting the error

```
ax = pd.Series(errors).plot(xlabel='β', ylabel='error')
plt.axvline(β_optimal, color='r');
```

Now let us vary α (holding β constant)

```
errors = {}
for α in np.arange(-500,500,5):
    errors[α] = abs((α + β_optimal * df['X']) - df['Y']).sum()
```

Plotting the error

```
ax = pd.Series(errors).plot(xlabel='α', ylabel='error')
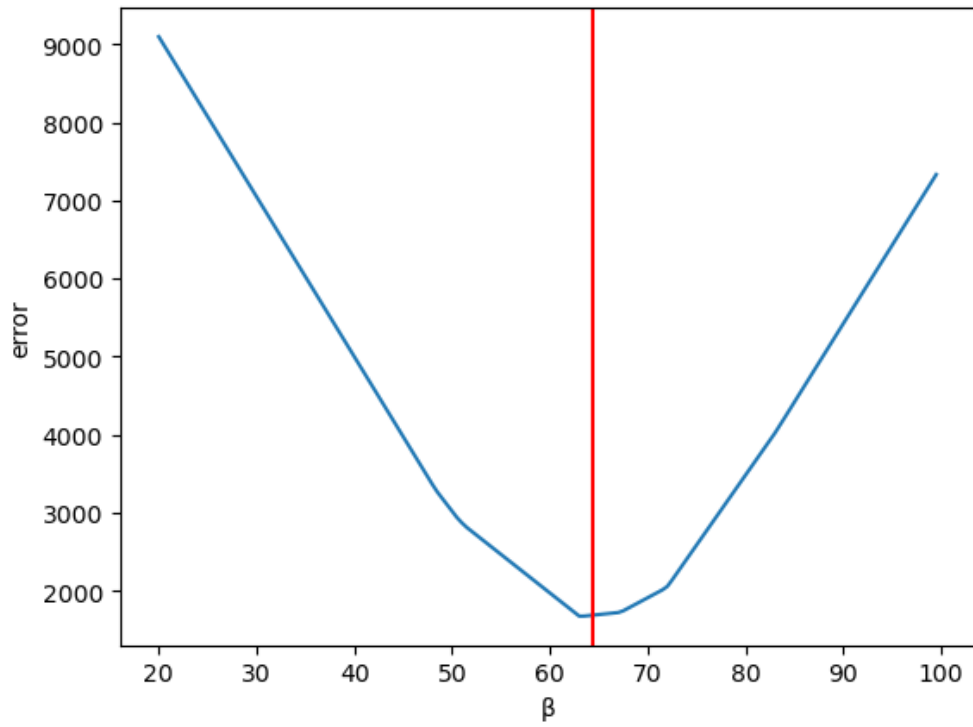plt.axvline(α_optimal, color='r');
```

Fig. 44.6: Plotting the error
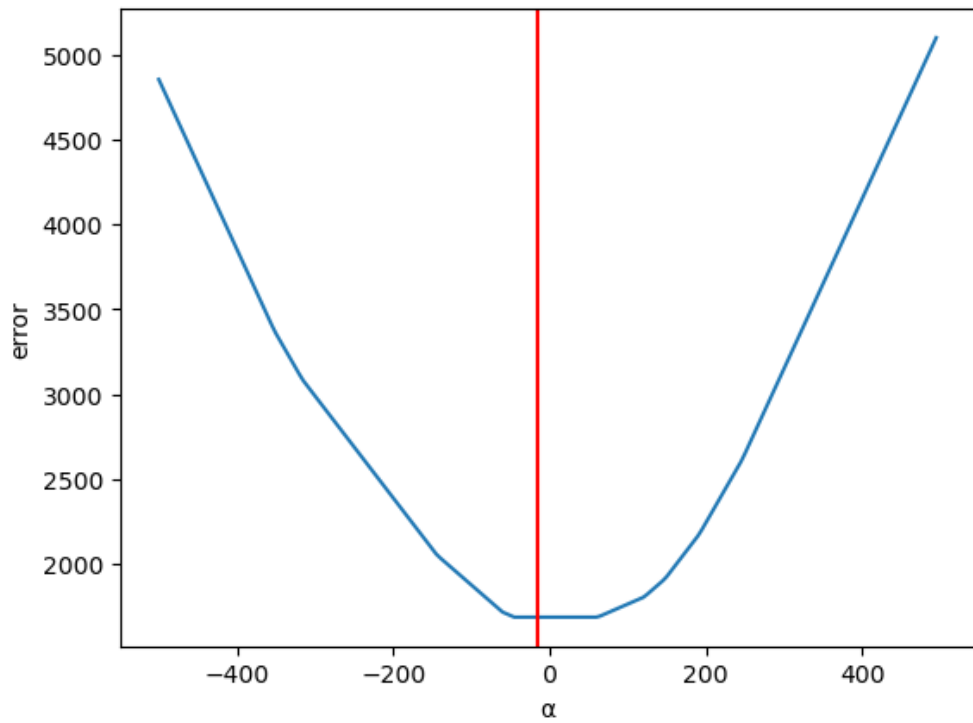
Fig. 44.7: Plotting the error (2)

44.2 Calculating optimal values

Now let us use calculus to solve the optimization problem and compute the optimal values for α and β to find the ordinary least squares solution.

First taking the partial derivative with respect to α

$$\frac{\partial C}{\partial \alpha}[\sum_{i=1}^{N}(y_i - \alpha - \beta x_i)^2]$$

and setting it equal to 0

$$0 = \sum_{i=1}^{N} -2(y_i - \alpha - \beta x_i)$$

we can remove the constant -2 from the summation by dividing both sides by -2

$$0 = \sum_{i=1}^{N}(y_i - \alpha - \beta x_i)$$

Now we can split this equation up into the components

$$0 = \sum_{i=1}^{N} y_i - \sum_{i=1}^{N} \alpha - \beta \sum_{i=1}^{N} x_i$$

The middle term is a straight forward sum from $i = 1, ...N$ by a constant α

$$0 = \sum_{i=1}^{N} y_i - N * \alpha - \beta \sum_{i=1}^{N} x_i$$

and rearranging terms

$$\alpha = \frac{\sum_{i=1}^{N} y_i - \beta \sum_{i=1}^{N} x_i}{N}$$

We observe that both fractions resolve to the means $\bar{y}_i$ and $\bar{x}_i$

$$\alpha = \bar{y}_i - \beta \bar{x}_i \qquad (44.1)$$

Now let's take the partial derivative of the cost function C with respect to β

$$\frac{\partial C}{\partial \beta}[\sum_{i=1}^{N}(y_i - \alpha - \beta x_i)^2]$$

and setting it equal to 0

$$0 = \sum_{i=1}^{N} -2x_i(y_i - \alpha - \beta x_i)$$

we can again take the constant outside of the summation and divide both sides by -2

$$0 = \sum_{i=1}^{N} x_i (y_i - \alpha - \beta x_i)$$

which becomes

$$0 = \sum_{i=1}^{N} (x_i y_i - \alpha x_i - \beta x_i^2)$$

now substituting for α

$$0 = \sum_{i=1}^{N} (x_i y_i - (\bar{y}_i - \beta \bar{x}_i) x_i - \beta x_i^2)$$

and rearranging terms

$$0 = \sum_{i=1}^{N} (x_i y_i - \bar{y}_i x_i - \beta \bar{x}_i x_i - \beta x_i^2)$$

This can be split into two summations

$$0 = \sum_{i=1}^{N} (x_i y_i - \bar{y}_i x_i) + \beta \sum_{i=1}^{N} (\bar{x}_i x_i - x_i^2)$$

and solving for β yields

$$\beta = \frac{\sum_{i=1}^{N} (x_i y_i - \bar{y}_i x_i)}{\sum_{i=1}^{N} (x_i^2 - \bar{x}_i x_i)} \tag{44.2}$$

We can now use (**??**) and (**??**) to calculate the optimal values for α and β

Calculating β

```
df = df[['X','Y']].copy()   # Original Data

# Calculate the sample means
x_bar = df['X'].mean()
y_bar = df['Y'].mean()
```

Now computing across the 10 observations and then summing the numerator and denominator

```
# Compute the Sums
df['num'] = df['X'] * df['Y'] - y_bar * df['X']
df['den'] = pow(df['X'],2) - x_bar * df['X']
β = df['num'].sum() / df['den'].sum()
print(β)
```

```
64.37665782493369
```

Calculating α

```
α = y_bar - β * x_bar
print(α)
```

```
-14.72148541114052
```

Now we can plot the OLS solution

```
df['Y_hat'] = α + β * df['X']
df['error'] = df['Y_hat'] - df['Y']

fig, ax = plt.subplots()
ax = df.plot(x='X',y='Y', kind='scatter', ax=ax)
ax = df.plot(x='X',y='Y_hat', kind='line', ax=ax, color='g')
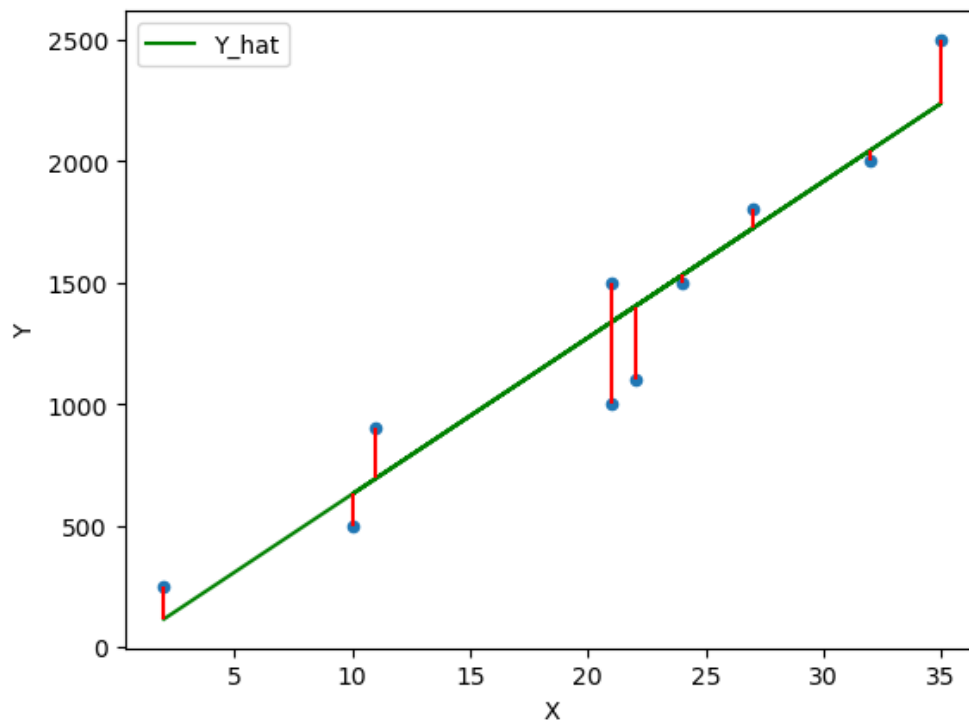plt.vlines(df['X'], df['Y_hat'], df['Y'], color='r');
```


Fig. 44.8: OLS line of best fit

Exercise 44.2.1

Now that you know the equations that solve the simple linear regression model using OLS you can now run your own regressions to build a model between y and x.

Let's consider two economic variables GDP per capita and Life Expectancy.

1. What do you think their relationship would be?

2. Gather some data from our world in data

3. Use `pandas` to import the `csv` formatted data and plot a few different countries of interest

4. Use (**??**) and (**??**) to compute optimal values for α and β

5. Plot the line of best fit found using OLS

6. Interpret the coefficients and write a summary sentence of the relationship between GDP per capita and Life Expectancy

---

Solution to Exercise 44.2.1

Q2: Gather some data from our world in data

You can download a copy of the data here if you get stuck

Q3: Use `pandas` to import the `csv` formatted data and plot a few different countries of interest

```
data_url = "https://github.com/QuantEcon/lecture-python-intro/raw/main/lectures/_
 ↪static/lecture_specific/simple_linear_regression/life-expectancy-vs-gdp-per-capita.
 ↪csv"
df = pd.read_csv(data_url, nrows=10)
```

```
df
```

```
         Entity      Code  Year  Life expectancy at birth (historical)   \
0       Abkhazia  OWID_ABK  2015                                    NaN
1    Afghanistan       AFG  1950                                   27.7
2    Afghanistan       AFG  1951                                   28.0
3    Afghanistan       AFG  1952                                   28.4
4    Afghanistan       AFG  1953                                   28.9
5    Afghanistan       AFG  1954                                   29.2
6    Afghanistan       AFG  1955                                   29.9
7    Afghanistan       AFG  1956                                   30.4
```

```
8   Afghanistan       AFG  1957                                  30.9
9   Afghanistan       AFG  1958                                  31.5

    GDP per capita  417485-annotations  Population (historical estimates)  \
0             NaN                  NaN                                NaN
1          1156.0                  NaN                          7480464.0
2          1170.0                  NaN                          7571542.0
3          1189.0                  NaN                          7667534.0
4          1240.0                  NaN                          7764549.0
5          1245.0                  NaN                          7864289.0
6          1246.0                  NaN                          7971933.0
7          1278.0                  NaN                          8087730.0
8          1253.0                  NaN                          8210207.0
9          1298.0                  NaN                          8333827.0

    Continent
0        Asia
1         NaN
2         NaN
3         NaN
4         NaN
5         NaN
6         NaN
7         NaN
8         NaN
9         NaN
```

You can see that the data downloaded from Our World in Data has provided a global set of countries with the GDP per capita and Life Expectancy Data.

It is often a good idea to at first import a few lines of data from a csv to understand its structure so that you can then choose the columns that you want to read into your DataFrame.

You can observe that there are a bunch of columns we won't need to import such as `Continent`

So let's built a list of the columns we want to import

```
cols = ['Code', 'Year', 'Life expectancy at birth (historical)', 'GDP per capita']
df = pd.read_csv(data_url, usecols=cols)
df
```

```
          Code  Year  Life expectancy at birth (historical)  GDP per capita
0    OWID_ABK  2015                                    NaN             NaN
```

```
1          AFG  1950                            27.7        1156.0
2          AFG  1951                            28.0        1170.0
3          AFG  1952                            28.4        1189.0
4          AFG  1953                            28.9        1240.0
...        ...  ...                             ...            ...
62151      ZWE  1946                            NaN            NaN
62152      ZWE  1947                            NaN            NaN
62153      ZWE  1948                            NaN            NaN
62154      ZWE  1949                            NaN            NaN
62155      ALA  2015                            NaN            NaN

[62156 rows x 4 columns]
```

Sometimes it can be useful to rename your columns to make it easier to work with in the DataFrame

```
df.columns = ["cntry", "year", "life_expectancy", "gdppc"]
df
```

```
         cntry  year  life_expectancy   gdppc
0      OWID_ABK  2015              NaN     NaN
1           AFG  1950             27.7  1156.0
2           AFG  1951             28.0  1170.0
3           AFG  1952             28.4  1189.0
4           AFG  1953             28.9  1240.0
...         ...  ...              ...     ...
62151       ZWE  1946              NaN     NaN
62152       ZWE  1947              NaN     NaN
62153       ZWE  1948              NaN     NaN
62154       ZWE  1949              NaN     NaN
62155       ALA  2015              NaN     NaN

[62156 rows x 4 columns]
```

We can see there are `NaN` values which represents missing data so let us go ahead and drop those

```
df.dropna(inplace=True)
```

```
df
```

```
      cntry  year  life_expectancy       gdppc
1       AFG  1950             27.7  1156.0000
```

```
2       AFG  1951         28.0  1170.0000
3       AFG  1952         28.4  1189.0000
4       AFG  1953         28.9  1240.0000
5       AFG  1954         29.2  1245.0000
...     ...  ...           ...        ...
61960   ZWE  2014         58.8  1594.0000
61961   ZWE  2015         59.6  1560.0000
61962   ZWE  2016         60.3  1534.0000
61963   ZWE  2017         60.7  1582.3662
61964   ZWE  2018         61.4  1611.4052


[12445 rows x 4 columns]
```

We have now dropped the number of rows in our DataFrame from 62156 to 12445 removing a lot of empty data relationships.

Now we have a dataset containing life expectancy and GDP per capita for a range of years.

It is always a good idea to spend a bit of time understanding what data you actually have.

For example, you may want to explore this data to see if there is consistent reporting for all countries across years

Let's first look at the Life Expectancy Data

```
le_years = df[['cntry', 'year', 'life_expectancy']].set_index(['cntry', 'year']).
 ↪unstack()['life_expectancy']
le_years
```

```
year   1543  1548  1553  1558  1563  1568  1573  1578  1583  1588  ...  2009  \
cntry                                                               ...
AFG    NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   ...  60.4
AGO    NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   ...  55.8
ALB    NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   ...  77.8
ARE    NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   ...  78.0
ARG    NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   ...  75.9
...    ...   ...   ...   ...   ...   ...   ...   ...   ...   ...   ...   ...
VNM    NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   ...  73.5
YEM    NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   ...  67.2
ZAF    NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   ...  57.4
ZMB    NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   ...  55.3
ZWE    NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN   ...  48.1


year   2010  2011  2012  2013  2014  2015  2016  2017  2018
cntry
```

```
AFG    60.9   61.4   61.9   62.4   62.5   62.7   63.1   63.0   63.1
AGO    56.7   57.6   58.6   59.3   60.0   60.7   61.1   61.7   62.1
ALB    77.9   78.1   78.1   78.1   78.4   78.6   78.9   79.0   79.2
ARE    78.3   78.5   78.7   78.9   79.0   79.2   79.3   79.5   79.6
ARG    75.7   76.1   76.5   76.5   76.8   76.8   76.3   76.8   77.0
...     ...    ...    ...    ...    ...    ...    ...    ...    ...
VNM    73.5   73.7   73.7   73.8   73.9   73.9   73.9   74.0   74.0
YEM    67.3   67.4   67.3   67.5   67.4   65.9   66.1   66.0   64.6
ZAF    58.9   60.7   61.8   62.5   63.4   63.9   64.7   65.4   65.7
ZMB    56.8   57.8   58.9   59.9   60.7   61.2   61.8   62.1   62.3
ZWE    50.7   53.3   55.6   57.5   58.8   59.6   60.3   60.7   61.4


[166 rows x 310 columns]
```

As you can see there are a lot of countries where data is not available for the Year 1543!

Which country does report this data?

```
le_years[~le_years[1543].isna()]
```

```
year    1543    1548    1553    1558    1563    1568    1573    1578   1583    1588   \
cntry
GBR    33.94   38.82   39.59   22.38   36.66   39.67   41.06   41.56   42.7   37.05

year   ...  2009   2010   2011   2012   2013   2014   2015   2016   2017   2018
cntry  ...
GBR    ...  80.2   80.4   80.8   80.9   80.9   81.2   80.9   81.1   81.2   81.1


[1 rows x 310 columns]
```

You can see that Great Britain (GBR) is the only one available

You can also take a closer look at the time series to find that it is also non-continuous, even for GBR.

```
le_years.loc['GBR'].plot()
```

```
<Axes: xlabel='year'>
```
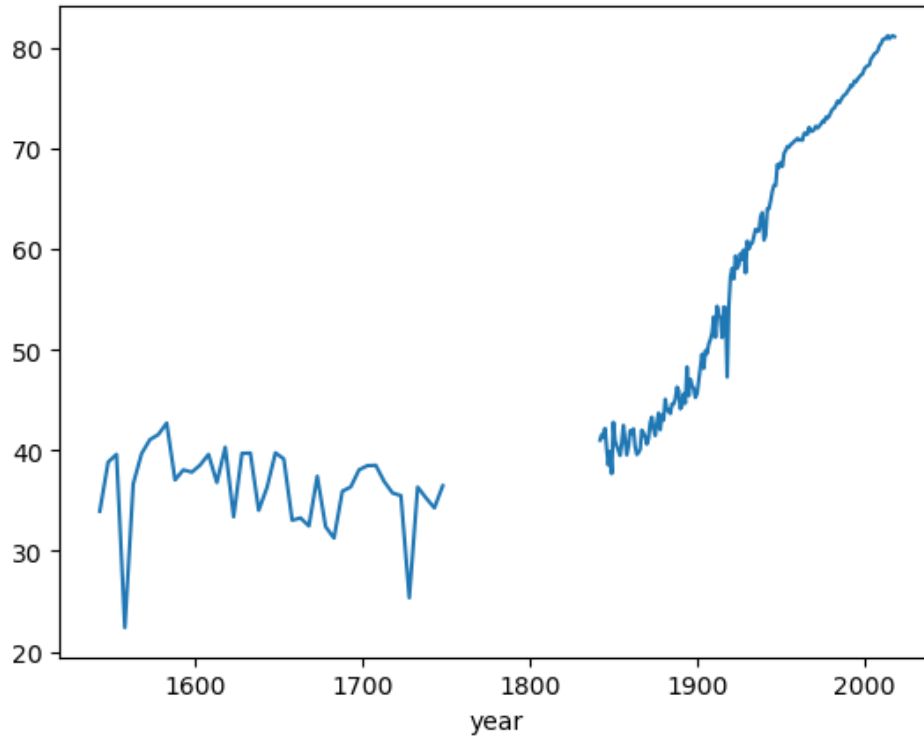
In fact we can use pandas to quickly check how many countries are captured in each year

```
le_years.stack().unstack(level=0).count(axis=1).plot(xlabel="Year", ylabel="Number of␣
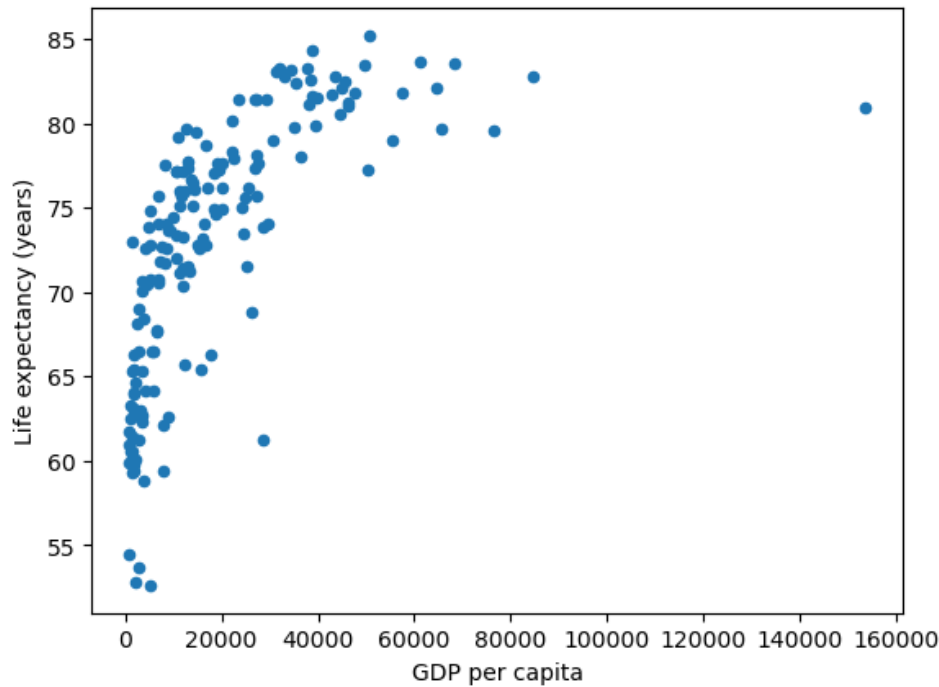 ↪countries");
```
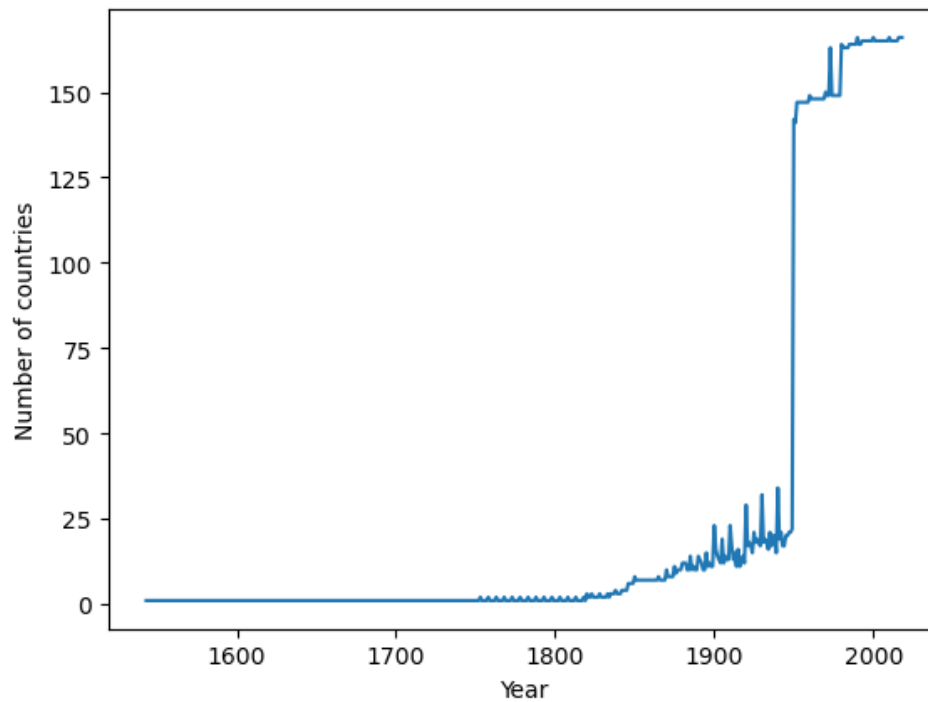
So it is clear that if you are doing cross-sectional comparisons then more recent data will include a wider set of countries

Now let us consider the most recent year in the dataset 2018

```
df = df[df.year == 2018].reset_index(drop=True).copy()
```

```
df.plot(x='gdppc', y='life_expectancy', kind='scatter',  xlabel="GDP per capita",␣
 ↪ylabel="Life expectancy (years)",);
```


This data shows a couple of interesting relationships.

1. there are a number of countries with similar GDP per capita levels but a wide range in Life Expectancy

2. there appears to be a positive relationship between GDP per capita and life expectancy. Countries with higher GDP per capita tend to have higher life expectancy outcomes

Even though OLS is solving linear equations –one option we have is to transform the variables, such as through a log transform, and then use OLS to estimate the transformed variables.

By specifying `logx` you can plot the GDP per Capita data on a log scale

```
df.plot(x='gdppc', y='life_expectancy', kind='scatter',  xlabel="GDP per capita",␣
 ↪ylabel="Life expectancy (years)", logx=True);
```
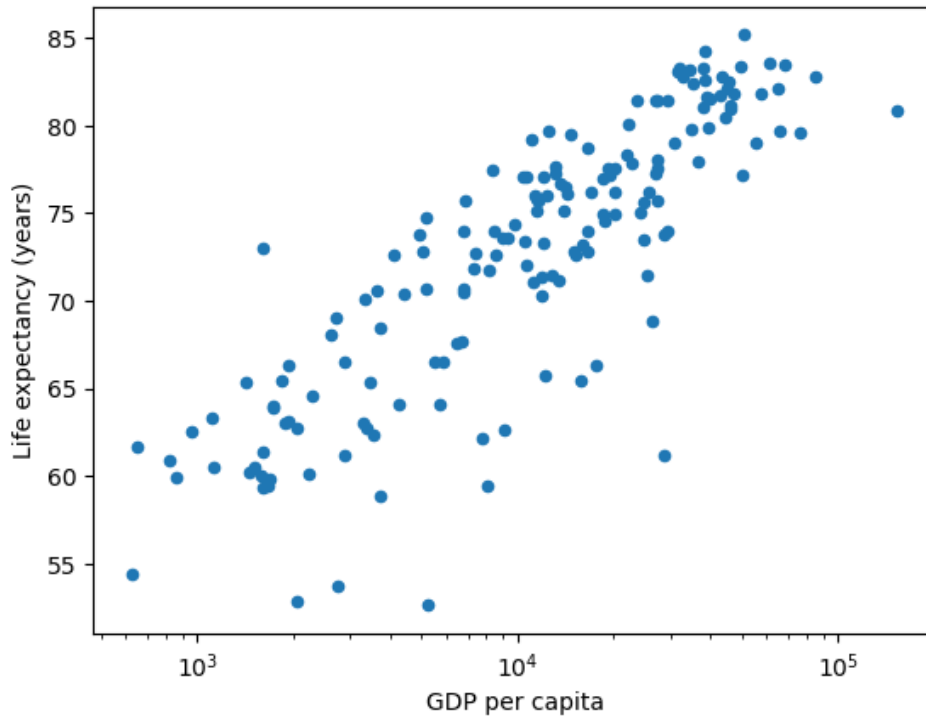
As you can see from this transformation – a linear model fits the shape of the data more closely.

```
df['log_gdppc'] = df['gdppc'].apply(np.log10)
```

```
df
```

```
        cntry  year  life_expectancy       gdppc  log_gdppc
0         AFG  2018             63.1   1934.5550   3.286581
1         ALB  2018             79.2  11104.1660   4.045486
2         DZA  2018             76.1  14228.0250   4.153145
3         AGO  2018             62.1   7771.4420   3.890502
4         ARG  2018             77.0  18556.3830   4.268493
..        ...   ...              ...         ...        ...
161       VNM  2018             74.0   6814.1420   3.833411
162  OWID_WRL  2018             72.6  15212.4150   4.182198
163       YEM  2018             64.6   2284.8900   3.358865
164       ZMB  2018             62.3   3534.0337   3.548271
165       ZWE  2018             61.4   1611.4052   3.207205

[166 rows x 5 columns]
```

Q4: Use (**??**) and (**??**) to compute optimal values for α and β

```
data = df[['log_gdppc', 'life_expectancy']].copy()  # Get Data from DataFrame


# Calculate the sample means
x_bar = data['log_gdppc'].mean()
y_bar = data['life_expectancy'].mean()
```

```
data
```

```
      log_gdppc  life_expectancy
0      3.286581             63.1
1      4.045486             79.2
2      4.153145             76.1
3      3.890502             62.1
4      4.268493             77.0
..          ...              ...
161    3.833411             74.0
162    4.182198             72.6
163    3.358865             64.6
164    3.548271             62.3
165    3.207205             61.4

[166 rows x 2 columns]
```

```
# Compute the Sums
data['num'] = data['log_gdppc'] * data['life_expectancy'] - y_bar * data['log_gdppc']
data['den'] = pow(data['log_gdppc'],2) - x_bar * data['log_gdppc']
β = data['num'].sum() / data['den'].sum()
print(β)
```

```
12.643730292819704
```

```
α = y_bar - β * x_bar
print(α)
```

```
21.702096701389053
```

Q5: Plot the line of best fit found using OLS

```
data['life_expectancy_hat'] = α + β * df['log_gdppc']
data['error'] = data['life_expectancy_hat'] - data['life_expectancy']
```

```
fig, ax = plt.subplots()
data.plot(x='log_gdppc',y='life_expectancy', kind='scatter', ax=ax)
data.plot(x='log_gdppc',y='life_expectancy_hat', kind='line', ax=ax, color='g')
plt.vlines(data['log_gdppc'], data['life_expectancy_hat'], data['life_expectancy'],↵
 ↪color='r')
```

```
<matplotlib.collections.LineCollection at 0x12eeaad10>
```
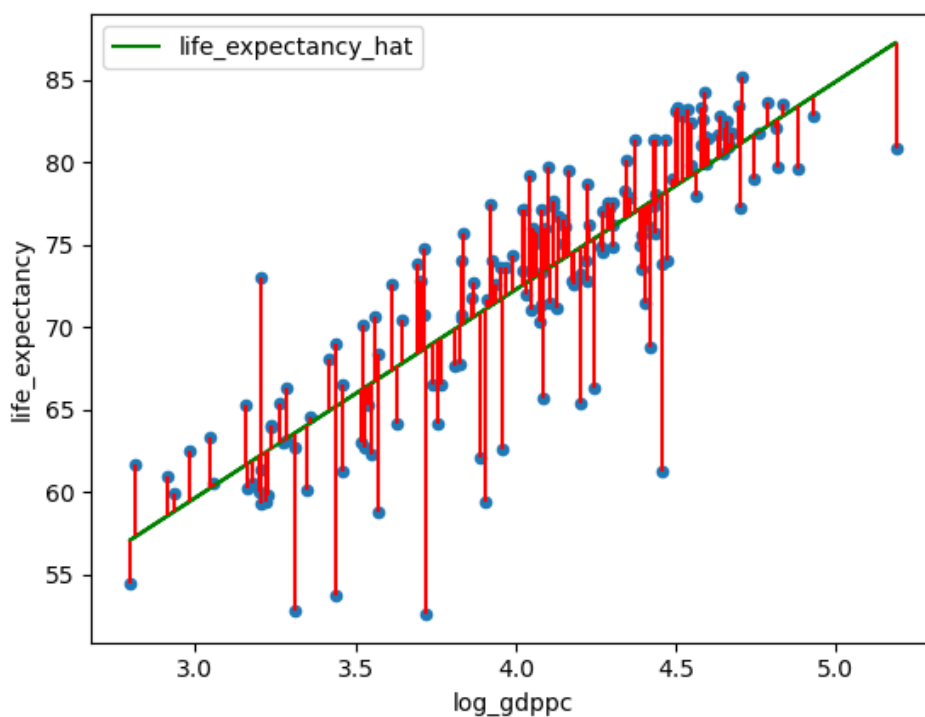

Exercise 44.2.2

Minimizing the sum of squares is not the **only** way to generate the line of best fit.

For example, we could also consider minimizing the sum of the **absolute values**, that would give less weight to outliers.

Solve for α and β using the least absolute values

MAXIMUM LIKELIHOOD ESTIMATION

```python
from scipy.stats import lognorm, pareto, expon
import numpy as np
from scipy.integrate import quad
import matplotlib.pyplot as plt
import pandas as pd
from math import exp
```

45.1 Introduction

Consider a situation where a policymaker is trying to estimate how much revenue a proposed wealth tax will raise.

The proposed tax is

$$h(w) = \begin{cases} aw & \text{if } w \leq \bar{w} \\ a\bar{w} + b(w - \bar{w}) & \text{if } w > \bar{w} \end{cases}$$

where w is wealth.

Example

For example, if $a = 0.05$, $b = 0.1$, and $\bar{w} = 2.5$, this means

- a 5% tax on wealth up to 2.5 and

- a 10% tax on wealth in excess of 2.5.

The unit is 100,000, so $w = 2.5$ means 250,000 dollars.

Let's go ahead and define h:

```
def h(w, a=0.05, b=0.1, w_bar=2.5):
    if w <= w_bar:
        return a * w
    else:
        return a * w_bar + b * (w - w_bar)
```

For a population of size N, where individual i has wealth w_i, total revenue raised by the tax will be

$$T = \sum_{i=1}^{N} h(w_i)$$

We wish to calculate this quantity.

The problem we face is that, in most countries, wealth is not observed for all individuals.

Collecting and maintaining accurate wealth data for all individuals or households in a country is just too hard.

So let's suppose instead that we obtain a sample $w_1, w_2, \cdots, w_n$ telling us the wealth of n randomly selected individuals.

For our exercise we are going to use a sample of $n = 10,000$ observations from wealth data in the US in 2016.

```
n = 10_000
```

The data is derived from the Survey of Consumer Finances (SCF).

The following code imports this data and reads it into an array called `sample`.

Let's histogram this sample.

```
fig, ax = plt.subplots()
ax.set_xlim(-1, 20)
density, edges = np.histogram(sample, bins=5000, density=True)
prob = density * np.diff(edges)
plt.stairs(prob, edges, fill=True, alpha=0.8, label=r"unit: $\$100,000$")
plt.ylabel("prob")
plt.xlabel("net wealth")
plt.legend()
plt.show()
```
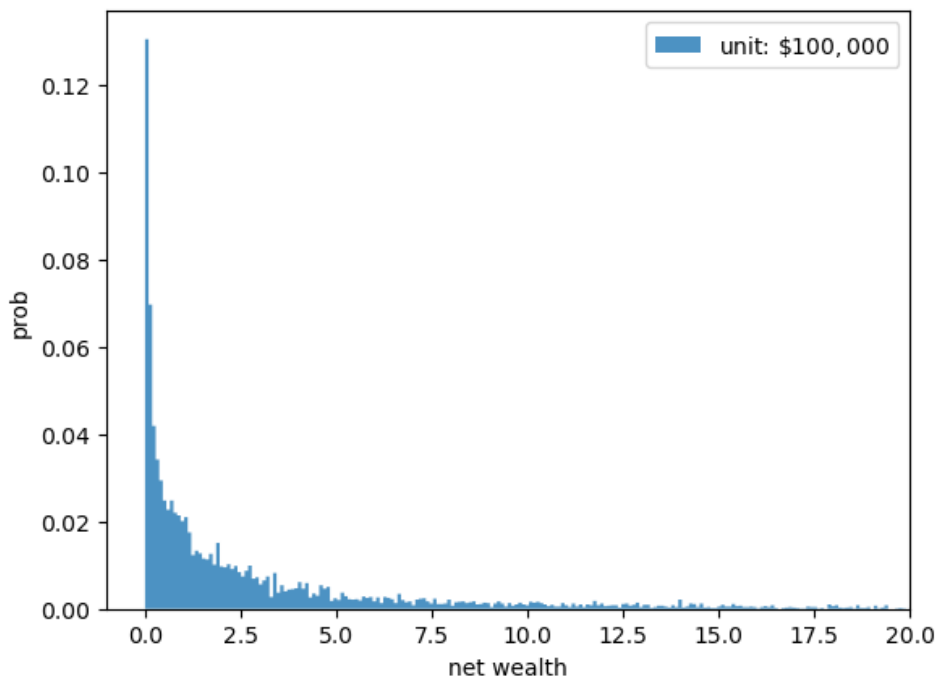
The histogram shows that many people have very low wealth and a few people have very high wealth.

We will take the full population size to be

```
N = 100_000_000
```

How can we estimate total revenue from the full population using only the sample data?

Our plan is to assume that wealth of each individual is a draw from a distribution with density f.

If we obtain an estimate of f we can then approximate T as follows:

$$T = \sum_{i=1}^{N} h(w_i) = N \frac{1}{N} \sum_{i=1}^{N} h(w_i) \approx N \int_{0}^{\infty} h(w) f(w) dw \tag{45.1}$$

(The sample mean should be close to the mean by the law of large numbers.)

The problem now is: how do we estimate f?

45.2 Maximum likelihood estimation

Maximum likelihood estimation is a method of estimating an unknown distribution.

Maximum likelihood estimation has two steps:

1. Guess what the underlying distribution is (e.g., normal with mean μ and standard deviation σ).

2. Estimate the parameter values (e.g., estimate μ and σ for the normal distribution)

One possible assumption for the wealth is that each w_i is log-normally distributed, with parameters $\mu \in (-\infty, \infty)$ and $\sigma \in (0, \infty)$.

(This means that $\ln w_i$ is normally distributed with mean μ and standard deviation σ.)

You can see that this assumption is not completely unreasonable because, if we histogram log wealth instead of wealth, the picture starts to look something like a bell-shaped curve.

```
ln_sample = np.log(sample)
fig, ax = plt.subplots()
ax.hist(ln_sample, density=True, bins=200, histtype='stepfilled', alpha=0.8)
plt.show()
```
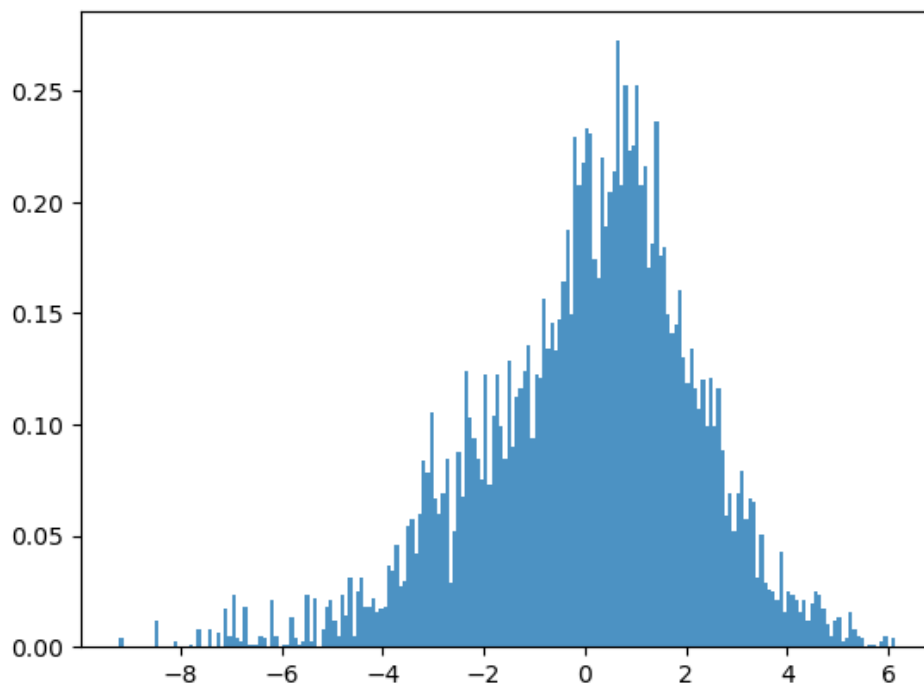

Now our job is to obtain the maximum likelihood estimates of μ and σ, which we denote by $\hat{\mu}$ and $\hat{\sigma}$.

These estimates can be found by maximizing the likelihood function given the data.

The pdf of a lognormally distributed random variable X is given by:

$$f(x, \mu, \sigma) = \frac{1}{x} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(\frac{-1}{2}\left(\frac{\ln x - \mu}{\sigma}\right)\right)^2$$

For our sample $w_1, w_2, \cdots, w_n$, the likelihood function is given by

$$L(\mu, \sigma | w_i) = \prod_{i=1}^{n} f(w_i, \mu, \sigma)$$

The likelihood function can be viewed as both

- the joint distribution of the sample (which is assumed to be IID) and

- the "likelihood" of parameters (μ, σ) given the data.

Taking logs on both sides gives us the log likelihood function, which is

$$
\begin{aligned}
\ell(\mu, \sigma | w_i) &= \ln \left[\prod_{i=1}^{n} f(w_i, \mu, \sigma) \right] \\
&= -\sum_{i=1}^{n} \ln w_i - \frac{n}{2} \ln(2\pi) - \frac{n}{2} \ln \sigma^2 - \frac{1}{2\sigma^2} \sum_{i=1}^{n} (\ln w_i - \mu)^2
\end{aligned}
$$

To find where this function is maximised we find its partial derivatives wrt μ and σ^2 and equate them to 0.

Let's first find the maximum likelihood estimate (MLE) of μ

$$
\begin{aligned}
\frac{\delta \ell}{\delta \mu} &= -\frac{1}{2\sigma^2} \times 2 \sum_{i=1}^{n} (\ln w_i - \mu) = 0 \\
&\implies \sum_{i=1}^{n} \ln w_i - n\mu = 0 \\
&\implies \hat{\mu} = \frac{\sum_{i=1}^{n} \ln w_i}{n}
\end{aligned}
$$

Now let's find the MLE of σ

$$
\begin{aligned}
\frac{\delta \ell}{\delta \sigma^2} &= -\frac{n}{2\sigma^2} + \frac{1}{2\sigma^4} \sum_{i=1}^{n} (\ln w_i - \mu)^2 = 0 \\
&\implies \frac{n}{2\sigma^2} = \frac{1}{2\sigma^4} \sum_{i=1}^{n} (\ln w_i - \mu)^2 \\
&\implies \hat{\sigma} = \left(\frac{\sum_{i=1}^{n} (\ln w_i - \hat{\mu})^2}{n} \right)^{1/2}
\end{aligned}
$$

Now that we have derived the expressions for $\hat{\mu}$ and $\hat{\sigma}$, let's compute them for our wealth sample.

```
μ_hat = np.mean(ln_sample)
μ_hat
```

```
0.0634375526654064
```

```
num = (ln_sample - μ_hat)**2
σ_hat = (np.mean(num))**(1/2)
σ_hat
```

```
2.1507346258433424
```

Let's plot the lognormal pdf using the estimated parameters against our sample data.

```
dist_lognorm = lognorm(σ_hat, scale = exp(μ_hat))
x = np.linspace(0,50,10000)

fig, ax = plt.subplots()
ax.set_xlim(-1,20)

ax.hist(sample, density=True, bins=5_000, histtype='stepfilled', alpha=0.5)
ax.plot(x, dist_lognorm.pdf(x), 'k-', lw=0.5, label='lognormal pdf')
ax.legend()
plt.show()
```
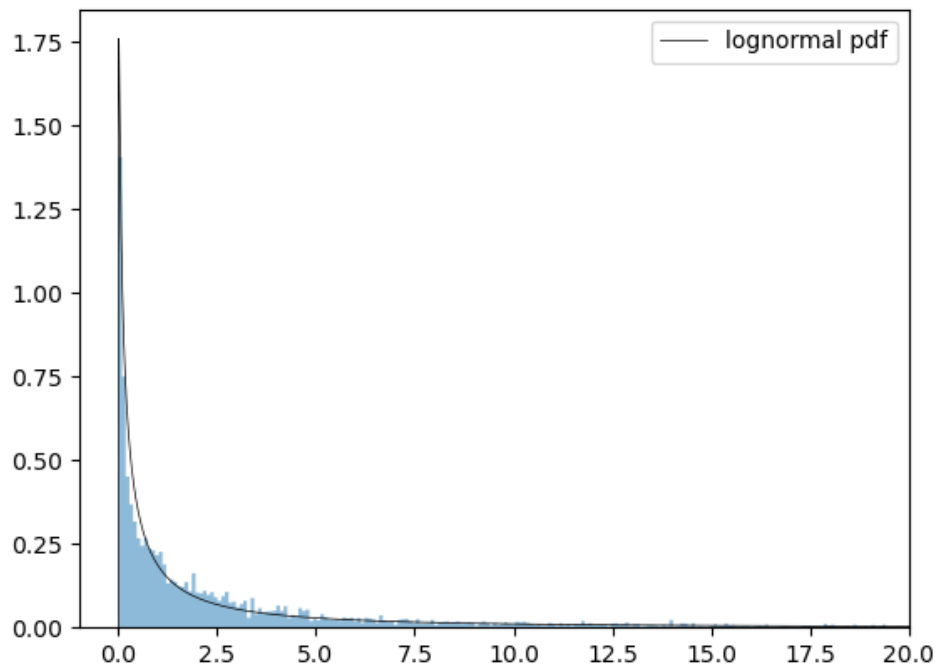

Our estimated lognormal distribution appears to be a reasonable fit for the overall data.

We now use (**??**) to calculate total revenue.

We will compute the integral using numerical integration via SciPy's quad function

```
def total_revenue(dist):
    integral, _ = quad(lambda x: h(x) * dist.pdf(x), 0, 100_000)
    T = N * integral
    return T
```

```
tr_lognorm = total_revenue(dist_lognorm)
tr_lognorm
```

```
101105326.82814859
```

(Our unit was 100,000 dollars, so this means that actual revenue is 100,000 times as large.)

45.3 Pareto distribution

We mentioned above that using maximum likelihood estimation requires us to make a prior assumption of the underlying distribution.

Previously we assumed that the distribution is lognormal.

Suppose instead we assume that w_i are drawn from the Pareto Distribution with parameters b and x_m.

In this case, the maximum likelihood estimates are known to be

$$\hat{b} = \frac{n}{\sum_{i=1}^{n} \ln(w_i/\hat{x_m})} \quad \text{and} \quad \hat{x}_m = \min_i w_i$$

Let's calculate them.

```
xm_hat = min(sample)
xm_hat
```

```
0.0001
```

```
den = np.log(sample/xm_hat)
b_hat = 1/np.mean(den)
b_hat
```

```
0.10783091940803055
```

Now let's recompute total revenue.

```
dist_pareto = pareto(b = b_hat, scale = xm_hat)
tr_pareto = total_revenue(dist_pareto)
tr_pareto
```

```
12933168365.762566
```

The number is very different!

```
tr_pareto / tr_lognorm
```

```
127.91777418162562
```

We see that choosing the right distribution is extremely important.

Let's compare the fitted Pareto distribution to the histogram:

```
fig, ax = plt.subplots()
ax.set_xlim(-1, 20)
ax.set_ylim(0,1.75)

ax.hist(sample, density=True, bins=5_000, histtype='stepfilled', alpha=0.5)
ax.plot(x, dist_pareto.pdf(x), 'k-', lw=0.5, label='Pareto pdf')
ax.legend()

plt.show()
```
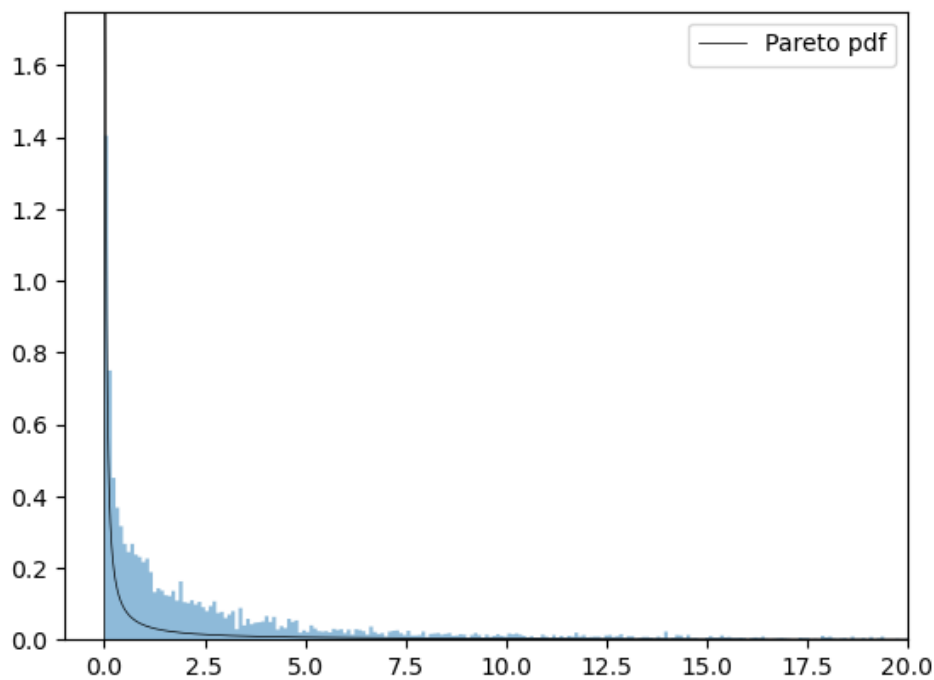

We observe that in this case the fit for the Pareto distribution is not very good, so we can probably reject it.

45.4 What is the best distribution?

There is no "best" distribution —every choice we make is an assumption.

All we can do is try to pick a distribution that fits the data well.

The plots above suggested that the lognormal distribution is optimal.

However when we inspect the upper tail (the richest people), the Pareto distribution may be a better fit.

To see this, let's now set a minimum threshold of net worth in our dataset.

We set an arbitrary threshold of $500,000 and read the data into `sample_tail`.

Let's plot this data.

```
fig, ax = plt.subplots()
ax.set_xlim(0,50)
ax.hist(sample_tail, density=True, bins=500, histtype='stepfilled', alpha=0.8)
plt.show()
```
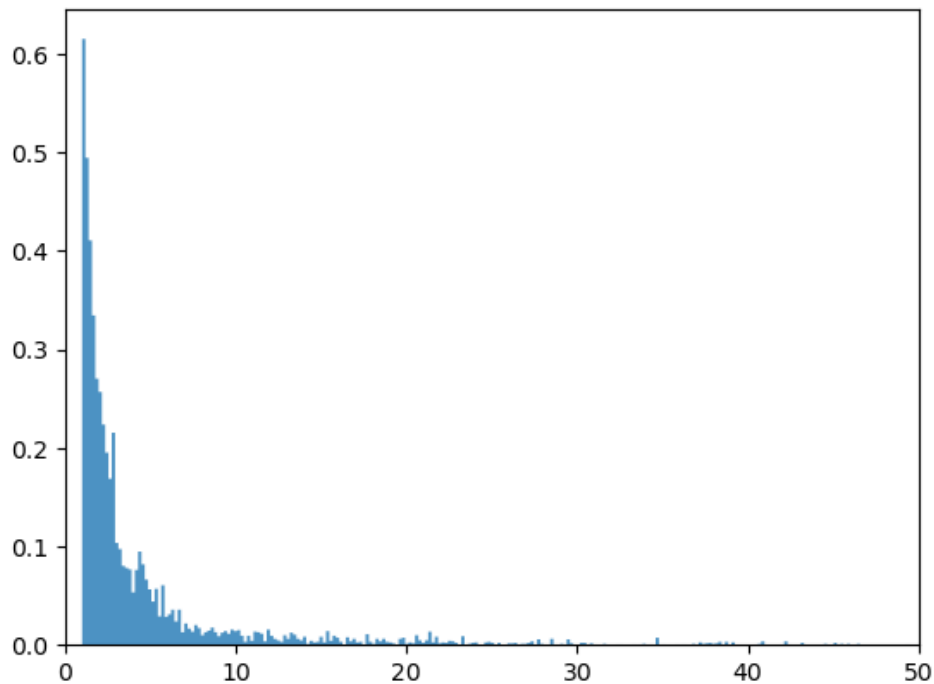

Now let's try fitting some distributions to this data.

45.4.1 Lognormal distribution for the right hand tail

Let's start with the lognormal distribution

We estimate the parameters again and plot the density against our data.

```
ln_sample_tail = np.log(sample_tail)
μ_hat_tail = np.mean(ln_sample_tail)
num_tail = (ln_sample_tail - μ_hat_tail)**2
σ_hat_tail = (np.mean(num_tail))**(1/2)
dist_lognorm_tail = lognorm(σ_hat_tail, scale = exp(μ_hat_tail))

fig, ax = plt.subplots()
ax.set_xlim(0,50)
ax.hist(sample_tail, density=True, bins=500, histtype='stepfilled', alpha=0.5)
ax.plot(x, dist_lognorm_tail.pdf(x), 'k-', lw=0.5, label='lognormal pdf')
ax.legend()
plt.show()
```
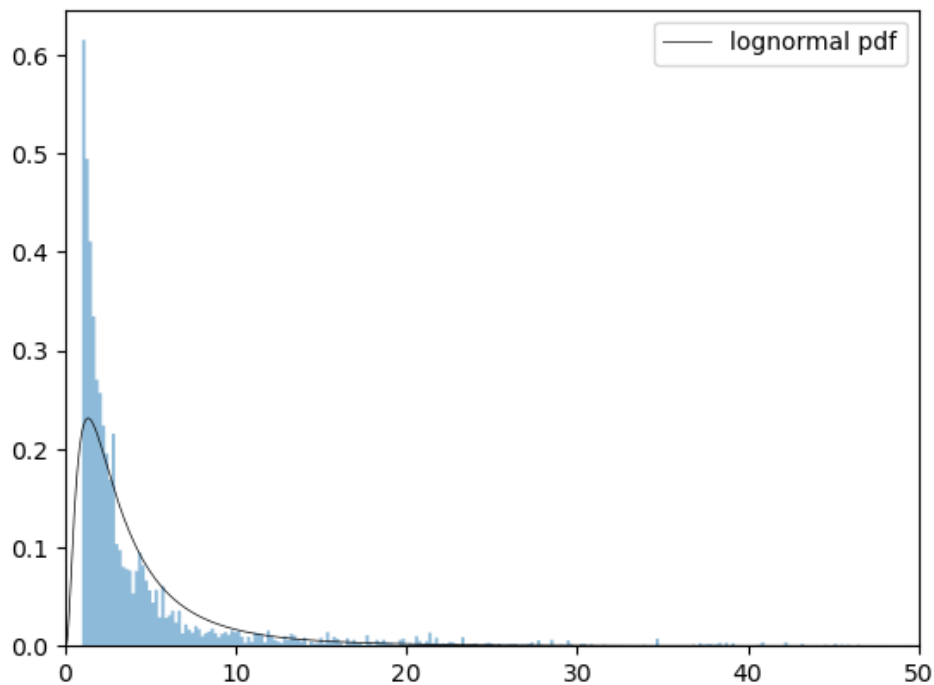

While the lognormal distribution was a good fit for the entire dataset, it is not a good fit for the right hand tail.

45.4.2 Pareto distribution for the right hand tail

Let's now assume the truncated dataset has a Pareto distribution.

We estimate the parameters again and plot the density against our data.

```
xm_hat_tail = min(sample_tail)
den_tail = np.log(sample_tail/xm_hat_tail)
b_hat_tail = 1/np.mean(den_tail)
dist_pareto_tail = pareto(b = b_hat_tail, scale = xm_hat_tail)

fig, ax = plt.subplots()
ax.set_xlim(0, 50)
ax.set_ylim(0,0.65)
ax.hist(sample_tail, density=True, bins= 500, histtype='stepfilled', alpha=0.5)
ax.plot(x, dist_pareto_tail.pdf(x), 'k-', lw=0.5, label='pareto pdf')
plt.show()
```
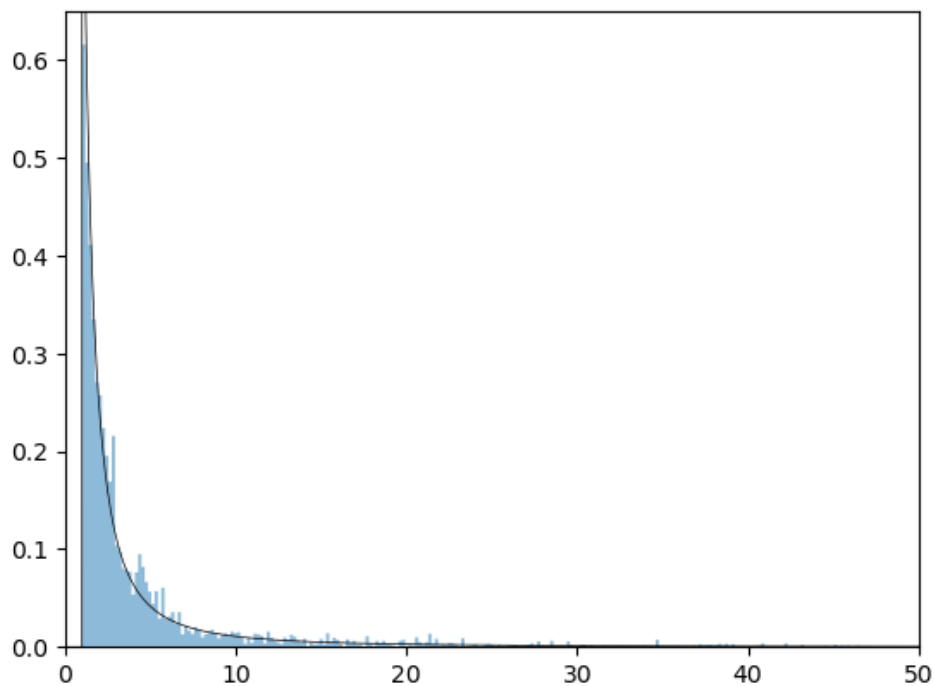

The Pareto distribution is a better fit for the right hand tail of our dataset.

45.4.3 So what is the best distribution?

As we said above, there is no "best" distribution —each choice is an assumption.

We just have to test what we think are reasonable distributions.

One test is to plot the data against the fitted distribution, as we did.

There are other more rigorous tests, such as the Kolmogorov-Smirnov test.

We omit such advanced topics (but encourage readers to study them once they have completed these lectures).

45.5 Exercises

Exercise 45.5.1

Suppose we assume wealth is exponentially distributed with parameter $\lambda > 0$.

The maximum likelihood estimate of λ is given by

$$\hat{\lambda} = \frac{n}{\sum_{i=1}^{n} w_i}$$

1. Compute $\hat{\lambda}$ for our initial sample.

2. Use $\hat{\lambda}$ to find the total revenue

Solution to Exercise 45.5.1

```
λ_hat = 1/np.mean(sample)
λ_hat
```

```
0.15234120963403971
```

```
dist_exp = expon(scale = 1/λ_hat)
tr_expo = total_revenue(dist_exp)
tr_expo
```

```
55246978.53427645
```

Exercise 45.5.2

Plot the exponential distribution against the sample and check if it is a good fit or not.

Solution to Exercise 45.5.2

```
fig, ax = plt.subplots()
ax.set_xlim(-1, 20)

ax.hist(sample, density=True, bins=5000, histtype='stepfilled', alpha=0.5)
ax.plot(x, dist_exp.pdf(x), 'k-', lw=0.5, label='exponential pdf')
ax.legend()

plt.show()
```
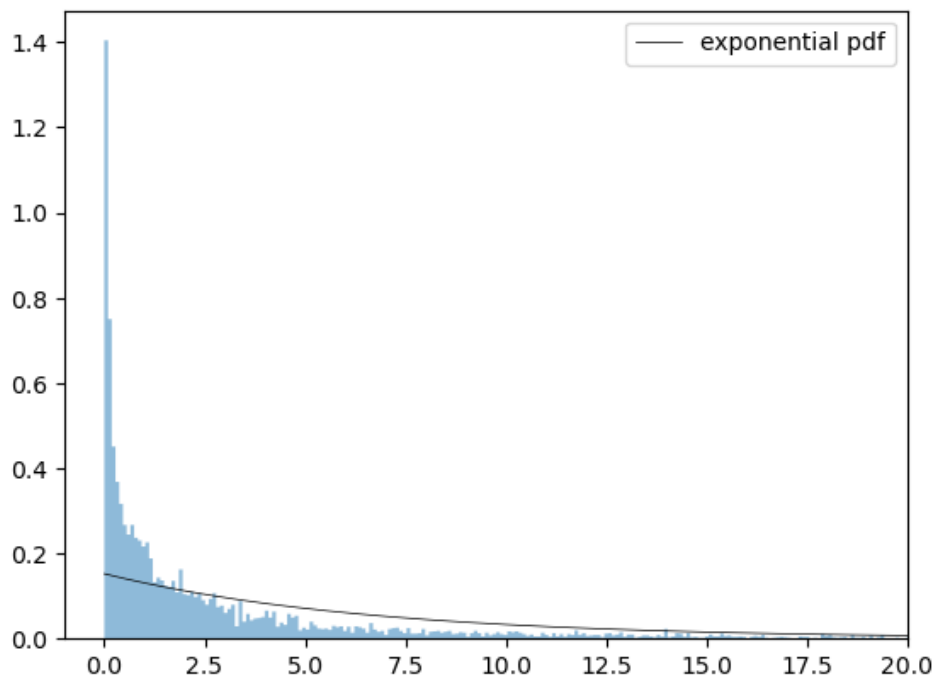

Clearly, this distribution is not a good fit for our data.

Part XIV

Other

FORTYSIX

TROUBLESHOOTING

This page is for readers experiencing errors when running the code from the lectures.

46.1 Fixing your local environment

The basic assumption of the lectures is that code in a lecture should execute whenever

1. it is executed in a Jupyter notebook and

2. the notebook is running on a machine with the latest version of Anaconda Python.

You have installed Anaconda, haven't you, following the instructions in this lecture?

Assuming that you have, the most common source of problems for our readers is that their Anaconda distribution is not up to date.

Here's a useful article on how to update Anaconda.

Another option is to simply remove Anaconda and reinstall.

You also need to keep the external code libraries, such as QuantEcon.py up to date.

For this task you can either

• use conda install -y quantecon on the command line, or

• execute !conda install -y quantecon within a Jupyter notebook.

If your local environment is still not working you can do two things.

First, you can use a remote machine instead, by clicking on the Launch Notebook icon available for each lecture

Second, you can report an issue, so we can try to fix your local set up.

We like getting feedback on the lectures so please don't hesitate to get in touch.

46.2 Reporting an issue

One way to give feedback is to raise an issue through our issue tracker.

Please be as specific as possible. Tell us where the problem is and as much detail about your local set up as you can provide.

Another feedback option is to use our discourse forum.

Finally, you can provide direct feedback to contact@quantecon.org

FORTYSEVEN

REFERENCES

EXECUTION STATISTICS

This table contains the latest execution statistics.

Document	Modified	Method	Run Time (s)	Status	
ar1_processes	2024-09-16 05:24	cache	5.34		✓
business_cycle	2024-09-16 01:26	cache	18.64		✓
cagan_adaptive	2024-09-16 05:24	cache	2.11		✓
cagan_ree	2024-09-16 05:24	cache	3.46		✓
cobweb	2024-09-16 05:24	cache	2.13		✓
commod_price	2024-09-16 05:24	cache	7.48		✓
complex_and_trig	2024-09-16 05:24	cache	2.19		✓
cons_smooth	2024-09-16 05:25	cache	3.43		✓
eigen_I	2024-09-16 05:25	cache	3.94		✓
eigen_II	2024-09-16 05:25	cache	4.57		✓
equalizing_difference	2024-09-16 05:25	cache	1.99		✓
french_rev	2024-09-16 05:55	cache	9.7		✓
geom_series	2024-09-16 05:25	cache	2.95		✓
greek_square	2024-09-16 05:25	cache	2.23		✓
heavy_tails	2024-09-16 05:25	cache	16.16		✓
inequality	2024-09-16 05:26	cache	40.97		✓
inflation_history	2024-09-16 05:26	cache	8.61		✓
input_output	2024-09-16 05:26	cache	11.27		✓
intro	2024-09-16 01:22	cache	0.97		✓
intro_supply_demand	2024-09-16 05:26	cache	2.17		✓
laffer_adaptive	2024-09-16 05:26	cache	1.93		✓
lake_model	2024-09-16 05:26	cache	2.08		✓
linear_equations	2024-09-16 05:26	cache	1.66		✓
lln_clt	2024-09-16 05:28	cache	104.03		✓
long_run_growth	2024-09-16 01:22	cache	10.42		✓

Table 48.1 – continued from previous page

Document	Modified	Method	Run Time (s)	Status
lp_intro	2024-09-16 05:28	cache	4.8	✓
markov_chains_I	2024-09-16 05:28	cache	8.97	✓
markov_chains_II	2024-09-16 05:28	cache	4.56	✓
mle	2024-09-16 05:29	cache	16.01	✓
money_inflation	2024-09-16 05:29	cache	2.09	✓
money_inflation_nonlinear	2024-09-16 05:29	cache	1.97	✓
monte_carlo	2024-09-16 05:31	cache	110.05	✓
networks	2024-09-16 05:31	cache	6.97	✓
olg	2024-09-16 05:31	cache	2.17	✓
prob_dist	2024-09-16 05:31	cache	6.78	✓
pv	2024-09-16 05:31	cache	1.55	✓
scalar_dynam	2024-09-16 05:31	cache	2.32	✓
schelling	2024-09-16 05:31	cache	8.8	✓
short_path	2024-09-16 05:31	cache	0.94	✓
simple_linear_regression	2024-09-16 05:31	cache	4.23	✓
solow	2024-09-16 05:31	cache	3.1	✓
status	2024-09-16 01:38	cache	4.62	✓
supply_demand_heterogeneity	2024-09-16 05:31	cache	0.97	✓
supply_demand_multiple_goods	2024-09-16 05:31	cache	1.41	✓
time_series_with_matrices	2024-09-16 05:31	cache	2.9	✓
troubleshooting	2024-09-16 01:22	cache	0.97	✓
unpleasant	2024-09-16 05:31	cache	1.57	✓
zreferences	2024-09-16 01:22	cache	0.97	✓

These lectures are built on `linux` instances through `github actions`.

These lectures are using the following python version

```
!python --version
```

```
Python 3.11.7
```

and the following package versions

```
!conda list
```

BIBLIOGRAPHY

[AR02]   Daron Acemoglu and James A. Robinson. The political economy of the Kuznets curve. *Review of Development Economics*, 6(2):183–203, 2002.

[AKM+18] SeHyoun Ahn, Greg Kaplan, Benjamin Moll, Thomas Winberry, and Christian Wolf. When inequality matters for macro and macro matters for inequality. *NBER Macroeconomics Annual*, 32(1):1–75, 2018.

[Axt01]   Robert L Axtell. Zipf distribution of us firm sizes. *science*, 293(5536):1818–1820, 2001.

[Bar79]   Robert J Barro. On the Determination of the Public Debt. *Journal of Political Economy*, 87(5):940–971, 1979.

[BB18]   Jess Benhabib and Alberto Bisin. Skewed wealth distributions: theory and empirics. *Journal of Economic Literature*, 56(4):1261–91, 2018.

[BBL19]   Jess Benhabib, Alberto Bisin, and Mi Luo. Wealth Distribution and Social Mobility in the US: A Quantitative Approach. *American Economic Review*, 109(5):1623–1647, May 2019.

[Ber97]   J. N. Bertsimas, D. & Tsitsiklis. *Introduction to linear optimization*. Athena Scientific, 1997.

[BEGS18] Anmol Bhandari, David Evans, Mikhail Golosov, and Thomas J Sargent. Inequality, business cycles, and monetary-fiscal policy. Technical Report, National Bureau of Economic Research, 2018.

[BEJ18]   Stephen P Borgatti, Martin G Everett, and Jeffrey C Johnson. *Analyzing social networks*. Sage, 2018.

[BF90]   Michael Bruno and Stanley Fischer. Seigniorage, operating rules, and the high inflation trap. *The Quarterly Journal of Economics*, 105(2):353–374, 1990.

[BW84]   John Bryant and Neil Wallace. A price discrimination analysis of monetary policy. *The Review of Economic Studies*, 51(2):279–288, 1984.

[Bur23]   Jennifer Burns. *Milton Friedman: The Last Conservative by Jennifer Burns*. Farrar, Straus, and Giroux, New York, 2023.

[Cag56]   Philip Cagan. The monetary dynamics of hyperinflation. In Milton Friedman, editor, *Studies in the Quantity Theory of Money*, pages 25–117. University of Chicago Press, Chicago, 1956.

[CB96]   Marcus J Chambers and Roy E Bailey. A theory of commodity price fluctuations. *Journal of Political Economy*, 104(5):924–957, 1996.

[Coc23]   John H Cochrane. *The Fiscal Theory of the Price Level*. Princeton University Press, Princeton, New Jersey, 2023.

[Cos21]   Michele Coscia. The atlas for the aspiring network scientist. *arXiv preprint arXiv:2101.00863*, 2021.

[DL92]   Angus Deaton and Guy Laroque. On the behavior of commodity prices. *The Review of Economic Studies*, 59:1–23, 1992.

[DL96]   Angus Deaton and Guy Laroque. Competitive storage and commodity price dynamics. *Journal of Political Economy*, 104(5):896–923, 1996.

[DSS58]   Robert Dorfman, Paul A. Samuelson, and Robert M. Solow. *Linear Programming and Economic Analysis: Revised Edition*. McGraw Hill, New York, 1958.

[EK+10]   David Easley, Jon Kleinberg, and others. *Networks, crowds, and markets*. Volume 8. Cambridge university press Cambridge, 2010.

[Fri56]   M. Friedman. *A Theory of the Consumption Function*. Princeton University Press, 1956.

[FK45]   Milton Friedman and Simon Kuznets. *Income from Independent Professional Practice*. National Bureau of Economic Research, New York, 1945.

[FDGA+04] Yoshi Fujiwara, Corrado Di Guilmi, Hideaki Aoyama, Mauro Gallegati, and Wataru Souma. Do pareto–zipf and gibrat laws hold true? an analysis with european firms. *Physica A: Statistical Mechanics and its Applications*, 335(1-2):197–216, 2004.

[Gab16]   Xavier Gabaix. Power laws in economics: an introduction. *Journal of Economic Perspectives*, 30(1):185–206, 2016.

[GSS03]   Edward Glaeser, Jose Scheinkman, and Andrei Shleifer. The injustice of inequality. *Journal of Monetary Economics*, 50(1):199–222, 2003.

[Goy23]   Sanjeev Goyal. *Networks: An economics approach*. MIT Press, 2023.

[Hal78]   Robert E Hall. Stochastic Implications of the Life Cycle-Permanent Income Hypothesis: Theory and Evidence. *Journal of Political Economy*, 86(6):971–987, 1978.

[Ham05]   James D Hamilton. What's real about the business cycle? *Federal Reserve Bank of St. Louis Review*, pages 435–452, 2005.

[Har60]   Arthur A. Harlow. The hog cycle and the cobweb theorem. *American Journal of Agricultural Economics*, 42(4):842–853, 1960. doi:https://doi.org/10.2307/1235116.

[Hu18]   Y. Hu, Y. & Guo. *Operations research*. Tsinghua University Press, 5th edition, 2018.

[Haggstrom02] Olle Häggström. *Finite Markov chains and algorithmic applications*. Volume 52. Cambridge University Press, 2002.

[IT23]   Patrick Imam and Jonathan RW Temple. Political institutions and output collapses. *IMF Working Paper*, 2023.

[Jac10]   Matthew O Jackson. *Social and economic networks*. Princeton university press, 2010.

[Key40]   John Maynard Keynes. How to pay for the war. In *Essays in persuasion*, pages 367–439. Springer, 1940.

[KLS18]   Illenin Kondo, Logan T Lewis, and Andrea Stella. On the us firm and establishment size distributions. Technical Report, SSRN, 2018.

[KF39]   Simon Kuznets and Milton Friedman. Incomes from independent professional practice, 1929-1936. *National Bureau of Economic Research Bulletin*, 1939.

[Lev19]   Malcolm Levitt. Why did ancient states collapse?: the dysfunctional state. *Why Did Ancient States Collapse?*, pages 1–56, 2019.

[Man63]   Benoit Mandelbrot. The variation of certain speculative prices. *The Journal of Business*, 36(4):394 – 419, 1963.

[MN03]   Albert Marcet and Juan P Nicolini. Recurrent hyperinflations and learning. *American Economic Review*, 93(5):1476–1498, 2003.

[MS89]   Albert Marcet and Thomas J Sargent. Least squares learning and the dynamics of hyperinflation. In William Barnett, John Geweke and Karl Shell, editors, *Sunspots, Complexity, and Chaos*. Cambridge University Press, 1989.

[MFD20]   Filippo Menczer, Santo Fortunato, and Clayton A Davis. *A first course in network science*. Cambridge University Press, 2020.

[MT09]   S P Meyn and R L Tweedie. *Markov Chains and Stochastic Stability*. Cambridge University Press, 2009.

[New18]   Mark Newman. *Networks*. Oxford university press, 2018.

[NW89]   Douglass C North and Barry R Weingast. Constitutions and commitment: the evolution of institutions governing public choice in seventeenth-century england. *The journal of economic history*, 49(4):803–832, 1989.

[Rac03]   Svetlozar Todorov Rachev. *Handbook of heavy tailed distributions in finance: Handbooks in finance*. Volume 1. Elsevier, 2003.

[RRGM11]   Hernán D Rozenfeld, Diego Rybski, Xavier Gabaix, and Hernán A Makse. The area and population of cities: new insights from a different perspective on cities. *American Economic Review*, 101(5):2205–25, 2011.

[Rus04]   Bertrand Russell. *History of western philosophy*. Routledge, 2004.

[Sam58]   Paul A Samuelson. An exact consumption-loan model of interest with or without the social contrivance of money. *Journal of political economy*, 66(6):467–482, 1958.

[Sam71]   Paul A Samuelson. Stochastic speculative price. *Proceedings of the National Academy of Sciences*, 68(2):335–337, 1971.

[Sam39]   Paul A. Samuelson. Interactions between the multiplier analysis and the principle of acceleration. *Review of Economic Studies*, 21(2):75–78, 1939.

[SWZ09]   Thomas Sargent, Noah Williams, and Tao Zha. The conquest of south american inflation. *Journal of Political Economy*, 117(2):211–256, 2009.

[Sar82]   Thomas J Sargent. The ends of four big inflations. In Robert E Hall, editor, *Inflation: Causes and effects*, pages 41–98. University of Chicago Press, 1982.

[Sar13]   Thomas J Sargent. *Rational Expectations and Inflation*. Princeton University Press, Princeton, New Jersey, 2013.

[SS22]   Thomas J Sargent and John Stachurski. Economic networks: theory and computation. *arXiv preprint arXiv:2203.11972*, 2022.

[SS23]   Thomas J Sargent and John Stachurski. Economic networks: theory and computation. *arXiv preprint arXiv:2203.11972*, 2023.

[SV95]   Thomas J Sargent and Francois R Velde. Macroeconomic features of the french revolution. *Journal of Political Economy*, 103(3):474–518, 1995.

[SV02]   Thomas J Sargent and François R Velde. *The Big Problem of Small Change*. Princeton University Press, Princeton, New Jersey, 2002.

[SW81]   Thomas J Sargent and Neil Wallace. Some unpleasant monetarist arithmetic. *Federal reserve bank of minneapolis quarterly review*, 5(3):1–17, 1981.

[SS83]   Jose A Scheinkman and Jack Schechtman. A simple competitive model with production and storage. *The Review of Economic Studies*, 50(3):427–441, 1983.

[Sch69]   Thomas C Schelling. Models of Segregation. *American Economic Review*, 59(2):488–493, 1969.

[ST19]   Christian Schluter and Mark Trede. Size distributions reconsidered. *Econometric Reviews*, 38(6):695–710, 2019.

[Smi10]   Adam Smith. *The Wealth of Nations: An inquiry into the nature and causes of the Wealth of Nations*. Harriman House Limited, 2010.

[Too14]   Adam Tooze. The deluge: the great war, america and the remaking of the global order, 1916–1931. 2014.

[Vil96]   Pareto Vilfredo. Cours d'économie politique. *Rouge, Lausanne*, 1896.

[Wau64]   Frederick V. Waugh. Cobweb models. *Journal of Farm Economics*, 46(4):732–750, 1964.

[WW82]   Brian D Wright and Jeffrey C Williams. The economic role of commodity storage. *The Economic Journal*, 92(367):596–614, 1982.

[Zha12]   Dongmei Zhao. *Power Distribution and Performance Analysis for Wireless Communication Networks*. SpringerBriefs in Computer Science. Springer US, Boston, MA, 2012. ISBN 978-1-4614-3283-8 978-1-4614-3284-5. URL: https://link.springer.com/10.1007/978-1-4614-3284-5 (visited on 2023-02-03), doi:10.1007/978-1-4614-3284-5.