動態マクロ経済学

大阪府立大学 現代システム科学域
マネジメント学類
2021 年度 講義ノート

佐藤 健治

目次

はじめに

対象

この講義ノートでは，時間を通じて変化するマクロ経済の様子を分析するための考え方を紹介する。マクロ経済学という分野を整理して理解する上では,「経済成長」と「経済変動」という 2 つの視点から経済環境を眺めることが有用である。すなわち，次のように考える：マクロ経済（＝国の経済全体）は，長期的には経済環境が好転していくという一定の傾向（トレンド）に従っているように見える一方で，短期の視野で見れば景気の浮き沈みが人々の暮らし向きを大きく左右している。長期的な「経済成長」と，短期的な「経済変動」の 2 つを合成することで，現実経済の複雑な振る舞いの理解に近づこうというのがマクロ経済分析の基本方針だと言える。

成長と変動のいずれに注目して分析するにしても,「時間」という要素が欠かせないことがすぐに分かるだろう。時間の止まった世界では，成長も景気の浮沈も起こりようがないからだ。したがって，マクロ経済学の重要な結論をきちんと理解するためには，時間を通じた変化を扱うための数学的・経済学的なモデルを構築・分析する手法を学ぶ必要がある。

時間を扱うための数学は微分方程式や差分方程式である。本講義では，経済変数が離散的な時間軸上で定義されていると考える差分方程式モデルを主に用いる。どちらかが正解とか簡単ということではないが，念のため選択を正当化しておこう。現実の経済がどのような振る舞いをするかについてのシンプルな理解を得たいというのが私たちの目標なのだけれども，複雑な経済現象をうまく表現するために巧妙な数理モデルを操作した結果，難解な数学的表現をまとった答えから先に進めなくなってしまう，ということがしばしば起こる。そのようなときには，コンピュータ・シミュレーションを用いたモデルの定量的な評価や可視化が大いに役に立つ。そして，シミュレーションの観点から言えば，離散時間のモデルは数学的な表現とコンピュータのコードとの対応関係が見つけやすいという意味でやや易しいと言えるかもしれない。これが本書で離散時間のモデルを用いる理由である。それ以上の

ものではない。

この講義で扱う内容は次の 3 つのポイントに集約される。

- 時間を考慮したマクロ経済モデル（数理モデル）を構築する方法
- 構築した数理モデルを解く方法
- 解いた結果をコンピュータで可視化する方法

この一連の操作に習熟すれば，マクロ経済モデルの結果を解釈・評価することができるようになる。

マクロ経済学が長期の成長と短期の変動の二本柱からなると言っても，実は最近のマクロ経済モデルの多くは，Ramsey モデルや世代重複モデルといった成長モデルを基礎に組み立てられている。これらのモデルは，単純な 2 期間の最適消費モデル（Fisher の異時点間の最適消費モデル）と大きくは変わらないので，恐れずにチャレンジしてほしい。

想定読者

想定する読者像は，標準的な学部中級レベルのミクロ経済学・マクロ経済学を履修済みの学部学生・大学院生である。中級のマクロ経済学を復習しながら，シミュレーションの手法を学んで，より上級のマクロ経済学に進むための橋渡しをすることを目的としている。

マネジメント学類開講科目のうち，

- マクロ経済学入門，マクロ経済学 I，マクロ経済学 II
- ミクロ経済学 I，ミクロ経済学 II

で扱う内容を習得していれば十分であろう。自習をするとすれば，以下の 3 冊を一通り読んでおくとよい。

- ハル・ヴァリアン（佐藤隆三・監訳）『入門 ミクロ経済学』勁草書房
- N・グレゴリー・マンキュー（足立英之他・訳）『マクロ経済学 I 入門編』東洋経済新報社
- N・グレゴリー・マンキュー（足立英之他・訳）『マクロ経済学 II 応用編』東洋経済新報社

高校数学や大学初年次で学ぶ微分積分学や線形代数学を超える数学を使うことはほとんどないはずだが，必要になった場合には詳しく説明することを心がけるので，根気よく式変形を追ってほしい。不安な読者は，制約条件付き最適化問題を解くた

めのラグランジュ未定乗数法などといった応用数学のテクニックを先に身につけておくと，スムーズに読むことができる。

特徴

この講義では読者が中級レベルまでのミクロ経済学およびマクロ経済学を履修済みであることを想定し，その知識の土台の上に，より高度な分析手法やコンピュータ・シミュレーションの技法を身につけることを目標としている。シミュレーションに用いる Python 3 の使用経験は問わないという方針のもと，講義で扱うトピックの順序は経済学的に自然であることをある程度は意識しつつも多くの例外が発生する。プログラミングのテクニックという観点から難易度が徐々に難しくなるように配慮したというのがその理由である。もし経済学に関する説明が直線的でないことに困難を感じる読者は，まず，上で紹介したミクロ・マクロの教科書をじっくり読むことをすすめる。

各章は，概要・理論・プログラミングの 3 つの部分から成る。理論，プログラミングについては複数の節にまたがることがある。

- 「概要」には，各回で理解してもらいたいトピックを列挙している。他の章との対応関係はこの部分に書くようにするで，基礎知識が足りないと感じた場合は，概要に記載されているパートを読んでみてほしい。この講義ノートは前から順番に読むようには書かれていないので，自由に動き回ってもらってよい。章の全体を一通り読み終えたら，概要に書かれている内容を自分で説明できるようになっているかを時間をとって考えてみてほしい。
- 「理論」には，数学的・経済学的な分析手法の紹介を書く。数学が苦手な読者は躓いても気にせずに読み進めて，プログラミングの分析まで進んでほしい。
- 「プログラミング」には,「理論」で紹介した分析をコンピュータ上で再現・確認するための方法を書いている。すべてのコードを読者が自分で打ち込んで実行することが期待されている。冗談かと思うかもしれないが，**プログラミングは指の筋肉を使って学ぶもの**である。相当な上級者になるまでは，コードを読むだけで理解しようなどとは考えないほうがいい。数字を変えたり，条件を変えたりしつつコードを実行しながら学ぶように心がけてほしい。

Python 3 の経験があるか他のプログラミング言語にかなり習熟しているが経済学の経験が少ない読者にとっては，標準的な経済学の教程に従った順序で読む方がスムーズに読み進められるかもしれない。そのように感じた読者は,「概要」を読ん

で，先にどの章を読むべきかを判断してほしい。

コンピュータ・シミュレーションの原理はプログラミング言語によらず共通のものであるが，特定の言語を固定することなく，この共通原理について説明することは大変に困難な仕事である。したがって，この講義では多くの講義と同様に 1 つの言語（すなわち，Python 3）を固定することにした。

プログラミング言語に Python を選んだのに特別な理由はない。しいて言えば，Python はフリーで人気があるプログラミング言語だからだ。Python にこだわらなくても，以下の機能を持つ言語であればなんでもよい。もっと言えば，1 つの言語に限定しなくても複数の環境を組み合わせることもできる。

- ベクトル計算を実行できること。
- 線形代数計算ができること。
- データ可視化が容易にできること。
- 関数の最大化・最小化をする方法を備えていること。
- 代数的に微分ができること。

Python 本体ではこのようなことはできないが，**NumPy**, **SciPy**, **Matplotlib**, **SymPy** というサードパーティ製のライブラリを用いて実現できる。**Pandas** という人気のあるデータ処理用のライブラリを用いれば，生データ，シミュレーション結果の処理を効率的に実行できる。本書では，**Pandas** の使い方についても簡単に紹介する。

構成

本講義ノートは 15 のセッションからなる講義のために書かれている。講義の構成は下表の通りである。

開発環境

この書籍のコードは，表 2 に示される環境で確認されている。必ずしもバージョンをすべて揃える必要はないが，バージョンが異なることで生じる問題は原則的に読者の責任で解決していただきたい。

表 1.: 講義の構成

講	理論	プログラミング
1	複利計算と成長	Python を関数電卓として使う
2	物価指数とインフレーション	ベクトルの表現と演算，簡単な可視化
3	GDP の成長と格差	繰り返し計算，制御構文
4	時系列データのモデリングと行列計算	線形システムのシミュレーションと AR モデル
5	成長と変動	Pandas による時系列データ取得，可視化
6	ソロー・モデルと成長会計	ユーザー定義関数，非線形システムのシミュレーション
7	2 期間最適消費モデル	SymPy による代数計算と数値最適化
8	ラムゼー・モデル (1)	非線形システムの線形化。フォワードルッキング・モデルのシミュレーション (1)
9	ラムゼー・モデル (2)	ダイナミック・プログラミング
10	リアル・ビジネスサイクル・モデル	フォワードルッキング・モデルのシミュレーション (2)
11	動学的 AD-AS モデル	
12	ニュー・ケインジアン・フィリップス曲線	フォワードルッキング・モデルのシミュレーション (3)
13	ニュー・ケインジアン・モデル	フォワードルッキング・モデルのシミュレーション (4)

表 2.: Python およびライブラリのバージョン

	Version
Python	3.8.5
NumPy	1.19.2
SciPy	1.5.2
SymPy	1.6.2
Pandas	1.1.3
Matplotlib	3.3.2
statsmodels	0.12.0
pandas-datareader	0.9.0

謝辞

本文中で紹介した Python およびサードパーティライブラリ（numpy, scipy, matplotlib, pandas）の他にも多くのオープンソースプロジェクトのお世話になった。LaTeX 2$_\varepsilon$ を用いたタイプセットに LyX を使用し，Python コードと実行結果を本文中に埋め込むために R (R Core Team, 2019), **knitr** (Xie, 2015, 2019) および **reticulate** (Ushey et al., 2019) を用いた。Python, R, LaTeX を取り巻くコミュニティに感謝する。

1. 変化率と複利計算

本講義のタイトルは「動態マクロ経済学」である。時間を通じて変化するマクロ経済の分析ということだ。この章では，変化量を記述する数学について基本的な部分を整理する。

1.1. 概要

経済学に限らず時間を通じて変化する対象を分析するには，変化の相対的な大きさを分析の対象にすると便利なことが多い。例えば,「5 年で年収を 500 万円増やす」という目標の達成難易度は当初年収によって異なる。計画時点の年収が 500 万円の人が年収 1000 万円になるには並々ならぬ努力と転職を伴うかもしれない。しかし，初めから 5000 万円の年収を受け取っている人が年収 5500 万円になることは,（これは私の想像に過ぎないけれども）現在の仕事の延長線上として達成できそうな気がする。同じ 500 万円増であっても，前者は 100% の年収アップ，後者は 10% アップであるから，変化率を見れば大きな違いがあることが分かるだろう。

現在の年収が 500 万円であるとしよう。もう少し現実的な目標として次のような 2 つの計画を立てて，比較する。

1. 毎年 10 万円ずつの年収増
2. 毎年 2% ずつの年収増

年収が倍の 1000 万円になるには何年かかるだろうか。1 つ目のプランは年収の増分が加法的に作用する一方で，2 つ目のプランは乗法的に作用する。最初のうちは大きな違いはないが，時間が経過するにつれて差が大きくなっていく。

以下では，簡単な数値例を用いて計算方法に習熟するとともに，数式や記号を用いた抽象的な思考に慣れよう。この章では，次のことを学ぶ。

- 時間を表す変数
- 時間変化の表示方法
- 平均変化率

- 複利計算

1.2. 理論

1.2.1. 時間を表す変数と時間変化する変数

2017 年の GDP が 500 兆円，2018 年の GDP が 510 兆円だったとしよう [1]。2017 年から 2018 年にかけての GDP の成長は

$$\frac{510-500}{500} = \frac{10}{500} = 0.02$$

によって計算できる。少し視点を変えてみよう。2017 年の 500 兆円と，2018 年の前年度からの成長率が 2% であることが分かっていれば，2018 年の GDP は次のようにして計算できる。

$$500 \times (1 + 0.02) = 510$$

ここでちょっとした記号を導入する。少しずつ抽象化していくので，まずは腕試しだ。2017 年の GDP を GDP_{17}, 2018 年の GDP を GDP_{18} と書くことにする。

$$\mathrm{GDP}_{17} = 500, \qquad \mathrm{GDP}_{18} = 510$$

また，2017 年から 2018 年にかけての**成長率**を $g_{18/17}$ と書くと，

$$g_{18/17} = \frac{\mathrm{GDP}_{18} - \mathrm{GDP}_{17}}{\mathrm{GDP}_{17}} = \frac{\mathrm{GDP}_{18}}{\mathrm{GDP}_{17}} - 1$$

および

$$\mathrm{GDP}_{18} = \left(1 + g_{18/17}\right) \mathrm{GDP}_{17}$$

と書ける。言語的な表現に近い記法を用いると読みやすい式が書ける。一方で，少し煩わしさを感じる読者もいるかもしれない。次に進もう。なお，GDP については成長率という言葉を用いることが多いが，より一般的な**増減率**とか**変化率**と同じものである。

数学モデルを構築するときにはさらに簡略化した表記が便利である。言語的な表現から離れるので，記号が何を意味しているかを常に意識しながら読む必要が生

[1] GDP とは Gross Domestic Product の略で，日本語では国内総生産と訳される。ある一定期間に国内で行われた生産活動の規模を測る指標である。詳しくは第 3 章で詳しく説明する。

じるものの，コンパクトな表現が数式の操作を容易にしてくれるというご利益がある。時間を表現する変数には t を用いるのが慣例である。$t = 0$ が分析の起点で，$t = 1, 2, 3, \ldots$ と数字が増えるにつれて特定の定まった間隔で時間が進行する。例えば，2010 年を分析の起点として，1 年毎の観測をもとに分析を進めるとすれば，

$$t = 0 \Leftrightarrow 2010 \text{ 年}, \quad t = 1 \Leftrightarrow 2011 \text{ 年}, \quad \cdots$$

のような対応関係を作ることができる。なお，整数値を取る時間軸上に構築された数理モデルを「**離散時間モデル**」と呼ぶ。

GDP に対して記号 y を使うとしよう。すると，y_0 は 2010 年の GDP，y_1 は 2011 年の GDP のように理解される。この記法を用いれば，前述の数値例は

$$y_7 = 500, \qquad y_8 = 510$$

と書ける。「2017 年の GDP は 500」というかわりに，「7 期の GDP は 500」という。特定の期を固定することなく，一般の t のままで議論を進められる方が数学的には何かと都合がよいことが多い。 単に，「t 期の GDP は y_t，$(t+1)$ 期の GDP は y_{t+1} である」のように言ったときには，$t = 10$ でも $t = 208$ でも何でもよいと考えているのである。普通は t が取る値の範囲まで示して，「$y_t, t = 0, 1, 2, \ldots,$ は t 期の GDP である」のように書く [2]。y_t のように，一定時間おきの観測値を表すデータを**時系列**とか**時系列データ**という。

問題 1.1. 分析の起点を 1951 年 ($t = 0$) とする。

1. 第 58 期 ($t = 58$) は西暦に換算すると何年か？
2. 一般の t 期は西暦何年か？ t を用いた公式を導きなさい。

[2] ここで，$t = 0, 1, 2, \ldots$ のようにピリオドを 3 つ並べた記法は以下略という意味である，省略されている内容が文脈から読み取れるとき以外には使わない。ある特定の T が t の最大値であると考えている場合には，$t = 0, 1, 2, \ldots, T$ と書く。

注意 1.1. 分析の開始期を $t = 0$ としたが，これを $t = 1$ としてもよい。$t = 3$ とか $t = 2000$ を起点としてもあまり嬉しいことはない。本書では配列のインデックスがゼロから始まる Python を用いたので，$t = 0$ を起点とする方が使いやすい。 □

注意 1.2. Python で表形式データを扱うためのライブラリ Pandas を用いると，実際の西暦や日付等をインデックスとして利用することができるので，実データ分析に限って言えば前述のような抽象化を使わずに済む場合も多い。しかし，数理モデルを使ったシミュレーションや高度な分析を行う場合には NumPy の配列を直接操作する方が都合がよいこともあるので，ゼロから始まる整数のインデックスを使う考え方に習熟しておこう。 □

問題 1.2. 1 年間の GDP ではなく四半期 GDP を分析したいとしよう。2000 年の第 1 四半期（1–3 月期）を分析の起点 $t = 0$ とする。

1. $t = 1, 2, \ldots$ と四半期 GDP $y_1, y_2, \ldots$ が表す内容を下表に書き下しなさい。

記号	意味
$t = 0$	2000 年の第 1 四半期（1–3 月期）
y_0	2000 年の第 1 四半期の四半期 GDP
$t = 1$	
y_1	
$t = 2$	
y_2	

2. $t = 10, t = 35$ はそれぞれ何年の第何四半期にあたるか？

注意 1.3. 問題 1.2 では暦年について 4 つの四半期を考えた。1 月始まりでない一般の会計年度を用いる場合には，例えば,「2019 年度第 3 四半期」が表しているものが何かをきちんと意識する必要がある。会計年度の始まりが 4 月であれば，日本で育った人にとっての標準的な理解は「2019 年の 10–12 月期」ということになる

だろう。しかし，2018 年の 10–12 月期と表すということがあるらしい。というよりむしろ後者が国際的なスタンダードのようだ。つまり,「2019 年度」というのは「その会計年度の終了月が 2019 年にある」ということになる。 □

1.2.2. 時間変化の大きさ

変数 y の，t 期から $(t+1)$ 期にかけての変化を表す基本的な方法は**差分** (difference) を取ることである。

$$\Delta y_{t+1} = y_{t+1} - y_t.$$

しかし，冒頭で述べたように Δy_{t+1} の大きさが意味する内容（容易に起こりそうな変化か，なかなか起こりそうにない変化か）は y_t の大きさによって異なる。この問題は変化率を計算することで解決できる。t 期から $(t+1)$ 期にかけての y の変化率（あるいは成長率）を g_{t+1} と書くことにしよう。これは,

$$g_{t+1} = \frac{\Delta y_{t+1}}{y_t} = \frac{y_{t+1} - y_t}{y_t} = \frac{y_{t+1}}{y_t} - 1$$

と定義される。もちろん,

$$y_{t+1} = (1 + g_{t+1}) y_t$$

である。変数 y が前節の GDP を表すとすれば,

$$g_8 = 0.02$$

となる。 ここでは，g_8 は 8 期の GDP の前年度からの成長率と解釈される。なお,

$$1 + g_{t+1} = \frac{y_{t+1}}{y_t}$$

は**粗成長率**（gross growth rate）という。

注意 1.4. 記号の定義は毎回必ずチェックする必要があることに注意しよう。例えば,

$$g_t = \frac{y_{t+1} - y_t}{y_t}$$

と成長率を定義する人がいても不思議ではない。逆に言えば，**数式を使った分析を実行するのであれば，すべての記号の定義を述べるのが読者に対する最低限の礼儀**

である [3]。y だから GDP とか，t だから時間だといった思い込みでさえ邪魔である。排除しよう。

注意 1.5. ただし，定義しさえすればどんな記号を使ってもよいという訳ではない。慣例的に使われている記号から逸脱することは避けたほうがよい。例えば，GDP を g として，GDP 成長率を r にするような選択をしたら大混乱を招くだろう。

問題 1.3. ある連続する 2 期の四半期 GDP の原系列 y_t, y_{t+1} が与えられているとする [4]。この 2 期間に渡って「成長率」

$$\frac{y_{t+1} - y_t}{y_t}$$

を計算することの問題点を指摘しなさい。（ヒント：購買行動の季節変化を考える）

隣接するデータ同士で成長率を計算するにはデータから季節変動を除去しなければならない（**季節調整**という）。本講義では，季節調整済みのデータのみを扱い，この問題は解決しているものとする。季節調整法の詳細に関心のある読者は北川 (2005) を参照せよ。

1.2.3. 累積成長と平均成長率

$t = 0, 1, 2, \ldots$ を期を表す変数，y_t を t 期の GDP とする。$n = 1, 2, \ldots$ に対して，n 期間の成長は次のように計算される。

$$\frac{y_{t+n} - y_t}{y_t} = \frac{y_{t+n}}{y_t} - 1$$

これは n 期間で起こる成長の大きさを測る指標だから，1 期分の成長とはスケールが異なることに注意しよう。1 期が 1 年の場合の変化率には「年率」などの便利な表現があるが，期間の長さが一般の場合にも使える一般的な日本語表現が定まって

[3] 私が未定義の記号を無断で使ってしまっている場合には教えて下さい。訂正します。
[4] 原系列というのは変換操作を施していない生の時系列データのこと。

いないようなので，本書では「期間変化率」とか「期間成長率」という言葉を用いることにしよう。

例 1.1. ある y_t の水準から，年率 2% の成長が 3 年間続いたとすれば（1 年を 1 期としている），3 年後の GDP, y_{t+3}, は次のように計算できる。

$$y_{t+1} = 1.02y_t, \quad y_{t+2} = 1.02y_{t+1}, \quad y_{t+3} = 1.02y_{t+2}$$

だから，

$$y_{t+3} = 1.02^3 y_t \approx 1.061 y_t$$

である。1 年間の成長が 2% のとき，3 年間の成長は 6.1% と，およそ 3 倍の大きさになる。 □

例 1.2. 国民経済計算（GDP 統計）の四半期 GDP 速報では季節調整済み前期比が公開される。前期比の成長率が g （例えば，$g = 0.01$ とか $g = 0.005$）のとき，年率換算した GDP 成長率は次のように計算される。

$$\text{四半期 GDP 成長率（季節調整済み前期比）の年率換算値} = (1+g)^4 - 1$$

□

注意 1.6. 「1 期」が表す期間が 1 年なのか四半期なのか，あるいは 1ヶ月なのかといった違いは数理モデルの記述には現れない。それではモデルと現実をどうやって接続するのかというと，成長率や利子率の大きさと期間の長さの関係を用いることが多い。例えば，現実の GDP 成長率が年率にして 2% であるとしよう。このとき，期間成長率が 6% であるとしてモデルをセットアップするなら，1 期 ≈ 3 年 であると考えるのが自然だろう。 □

粗成長率 y_{t+n}/y_t を次のように書き換える。

$$\begin{aligned}\frac{y_{t+n}}{y_t} &= \frac{y_{t+n}}{y_{t+n-1}} \times \frac{y_{t+n-1}}{y_{t+n-2}} \times \cdots \times \frac{y_{t+2}}{y_{t+1}} \times \frac{y_{t+1}}{y_t} \\ &= \underbrace{(1+g_{t+n}) \times (1+g_{t+n-1}) \times \cdots \times (1+g_{t+2}) \times (1+g_{t+1})}_{n \text{ 個}}.\end{aligned}$$

t 期と $t+n$ 期の間の y の**平均成長率**あるいは**平均変化率**とは，当該期間で一定の期間変化率を保って y_t から y_{t+n} に変化するとしたときの，その一定の期間変化

率のことである。平均成長率 $\bar{g}$ は次の性質を持つ。

$$\frac{y_{t+n}}{y_t} = \underbrace{(1+\bar{g})\times(1+\bar{g})\times\cdots\times(1+\bar{g})\times(1+\bar{g})}_{n\text{ 個}} = (1+\bar{g})^n .$$

したがって，$\bar{g}$ は次のように計算できる。

$$\bar{g} = \left(\frac{y_{t+n}}{y_t}\right)^{\frac{1}{n}} - 1$$

あるいは，隣接する 2 期の間の成長率 $g_{t+1}, g_{t+2}, \ldots, g_{t+n}$ を用いると，

$$\bar{g} = \{(1+g_{t+n})(1+g_{t+n-1})\cdots(1+g_{t+2})(1+g_{t+1})\}^{\frac{1}{n}} - 1. \tag{1.1}$$

問題 1.4. n 個の実数を適当に選び，$i = 1, 2, \ldots, n$ でインデックス付けしたものを $x_1, x_2, \ldots, x_n$ と書こう。これらの数をすべて足した値は，$\sum$ を使って次のように書く。

$$\sum_{i=1}^{n} x_i = x_1 + x_2 + \cdots + x_n.$$

この記法を使った次の等式を示しなさい。

$$y_{t+n} - y_t = \sum_{i=1}^{n} \Delta y_{t+i}.$$

問題 1.5. 加算記号 $\sum$ の乗算バージョンは $\prod$ である。この記号は指定されたすべての数の積を表す。

$$\prod_{i=1}^{n} x_i = x_1 x_2 \times \cdots \times x_n.$$

この記法を用いて，(1.1) の公式を書き直しなさい。

1.2.4. 成長率と対数関数

$b > 0$ かつ $b \neq 1$ とする。b を底とする対数関数 $\log_b(\cdot)$ は次の性質を持つ [5]。

1. 任意の $x, y > 0$ について,

$$\log_b xy = \log_b x + \log_b y, \qquad \log_b \frac{x}{y} = \log_b x - \log_b y$$

2. 任意の $x > 0$ と任意の実数 z について，$\log_b x^z = z \log_b x$.
3. 任意の $x > 0$ について，$b^{\log_b x} = x$.
4. $\log_b 1 = 0$.
5. $\log_b b = 1$.

底 b を変えることで生じるのは，対数関数の定数倍の違いである。

命題 1.1. $b, c > 0, b \neq 1 \neq c$ とする。任意の $x > 0$ に対して,

$$\log_b x = \log_b c \times \log_c x.$$

証明. これを示すには,

$$b^{\log_b x} = b^{\log_b c \times \log_c x}$$

を示せばよい。対数関数の性質から左辺は x である。右辺の方も x と一致することを次のようにして確認できる。

$$b^{\log_b c \times \log_c x} = \left(b^{\log_b c}\right)^{\log_c x} = c^{\log_c x} = x.$$

□

底としてよく使われるのは， $b = 2$ や 10 である。$\log_{10}(\cdot)$ を**常用対数**という。

[5] 関数の独立変数を囲むカッコは省略することが多い。

例 1.3. 変数を 2 倍にすると $\log_2(\cdot)$ を取った値は 1 だけ大きくなる。なぜなら，

$$\log_2 2x = \log_2 2 + \log_2 x = 1 + \log_2 x.$$

同様の方法で，変数を 10 倍にすると $\log_{10}(\cdot)$ は 1 だけ大きくなることを示せる。 □

$\log_2(\cdot)$ や $\log(\cdot)$ は人間が数字を評価するときには使いやすいが，数学的に最も便利という訳ではない。微分積分と最も親和性が高いのが**自然対数** $\log_e(\cdot)$ である。底 e は**ネイピア数**や**自然対数の底**と呼ばれ，次のように定義される。

$$e = \lim_{n \to +\infty} \left(1 + \frac{1}{n}\right)^n. \tag{1.2}$$

自然対数 $\log_e(\cdot)$ は通常，$\log(\cdot)$ のように e を省略して書くか，$\ln(\cdot)$ という特別な記号を用いて表現する。この講義では $\log(\cdot)$ という記法を採用する。

自然対数の微分

$$\big(\log x\big)' = \frac{1}{x}$$

は証明なしで認めておこう [6]。正値関数 $f(\cdot)$ の自然対数の微分もよく使われるので，覚えておこう。

$$\big(\log f(x)\big)' = \frac{f'(x)}{f(x)}.$$

次の性質はよく使われる。

命題 1.2. x がゼロに十分近いとき，次の近似式が成り立つ [7]。

$$\log(1 + x) \approx x$$

証明. テイラー展開を使って証明してみよう。

[6] 指数関数や対数関数をどのように定義するかによって議論が変わってくる。大学初年次で使った微分積分学の教科書を参考にせよ。

[7] 「十分近い」とか「十分大きい」という数学的な表現はよく使うので意味するところを覚えておこう。ここでは $|x| \to 0$ の極限で $|\log(1 + x) - x| \to 0$ が成り立つという意味である。

□

図 1.1 には $y = x$ と $y = \log(1 + x)$ のグラフが描かれている。$x = 0$ の周りで 2 つの関数が近接していることを確認してほしい。$|x|$ が大きくなるとグラフは次第に離れていく。つまり，$|x|$ が大きいときには命題 1.2 の近似は使えなくなるなる。

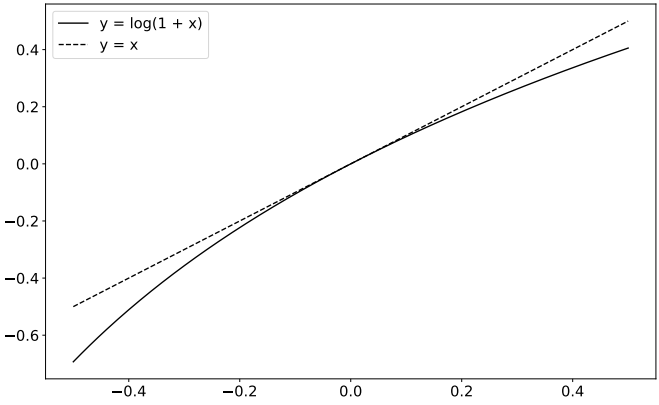


図 1.1.: 対数関数の近似

この命題を用いると「**対数差分は変化率を近似する**」ことが分かる。すなわち，

命題 1.3. $y_t > 0$ と $y_{t+1} > 0$ が十分近ければ，次の近似公式が成り立つ。

$$\Delta \log y_{t+1} = \log y_{t+1} - \log y_t \approx \frac{\Delta y_{t+1}}{y_t}$$

証明. 対数の差が商の対数になることに注意すると，

$$\begin{aligned}
\Delta \log y_{t+1} &= \log y_{t+1} - \log y_t \\
&= \log \frac{y_{t+1}}{y_t} \\
&= \log \left(1 + \frac{y_{t+1} - y_t}{y_t}\right) \\
&= \log \left(1 + \frac{\Delta y_{t+1}}{y_t}\right) \\
&\approx \frac{\Delta y_{t+1}}{y_t}.
\end{aligned}$$

□

例 1.4 (Rule of 70)**.** 年率 x %で成長している変数が 2 倍になるために必要な年数はおよそ $70/x$ 年である。これは「Rule of 70」と呼ばれる近似公式である。例えば，年率 2% で成長している変数は，35 年で 2 倍になる。証明を与えておこう。当初 y である変数が年率 x% で成長して T 年で 2 倍になるとすれば，y, x, T について次の式が成り立つ。

$$\left(1 + \frac{x}{100}\right)^T y = 2y$$

両辺の対数を取ると，

$$T \log \left(1 + \frac{x}{100}\right) + \log y = \log 2 + \log y$$

したがって，

$$T = \frac{\log 2}{\log \left(1 + \frac{x}{100}\right)} \approx \frac{\log 2}{x/100},$$

最後の近似には命題 1.3 を使っている。$\log 2 = 0.693147\cdots$ なので，

$$T \approx \frac{69.3}{x}$$

という近似が成り立つ。計算が容易になるように，$T \approx 70/x$ という公式がよく使われている。 □

1.2.5. 複利計算と指数関数

指数的成長

銀行に 100 万円を預けているとする。利息の年率を 6% とする。もちろん，計算を簡単にするために選んだ数字だ。追加の預け入れも引き出しもしなければ，1 年後には 106 万円になっている（諸々の税・手数料等を支払った後の利率が 6% としている）。さらに 1 年間預金に手を付けなければ預金残高は 112.36 万円となる。前節までで扱った成長の公式と同じ計算方式なので，すでに予想はついていると思うが，t 年後 $(t = 1, 2, 3, \ldots)$ の預金残高は $1.06^t \times 100$ 万円である。受け取った利息にも利息が付くので，t が増えるたび資産の増え方が早くなっていく。金融資産の評価の文脈では，このような計算方式を**複利計算**（compounding）と呼ぶ。

複利計算と似て非なるものとして単利計算という計算方式がある。これは，利息には利息がつかないという契約である。上記の例で利息を毎回引き出してタンス貯金すれば，t 年後の資産残高（銀行 + タンス）は $(100 + 6t)$ 万円になる [8]。複利計算と単利計算は短期的には大きな違いはないが，運用期間が長くなるにつれて差がどんどん大きくなる。図 1.2 で確認してほしい。

GDP 成長率の国際比較をするときに見かけ上は小さな差（例えば，1% か 2% か）を無視できないのは，小さな成長率の格差が長期的に大きな GDP の差につながることがあるからだ。複利計算のときのようにどんどん増えていくような状況を「指数的に増える」とか「幾何級数的に増える」などと表現する。

ネイピア数再び

さて，ここまでは利息が毎年一回受け取れるものとして計算を進めてきた。もっと頻繁に利息を受け取れるとすればどうなるだろう。例えば，毎月 1 回あるいは毎日 1 回利息が支払われるとすれば 1 年後の預金残高はいくらになっているだろうか。

各回の利息を計算するための基本の計算式は

$$\text{各回の利率} = \text{年率} \div \text{年間の受取回数}$$

[8] タンス預金にしなくても，このような状況を作ることができる。例えば，平均リターン 6% の投資信託を 100 万円分購入し，配当を再投資しないように設定すればよい。

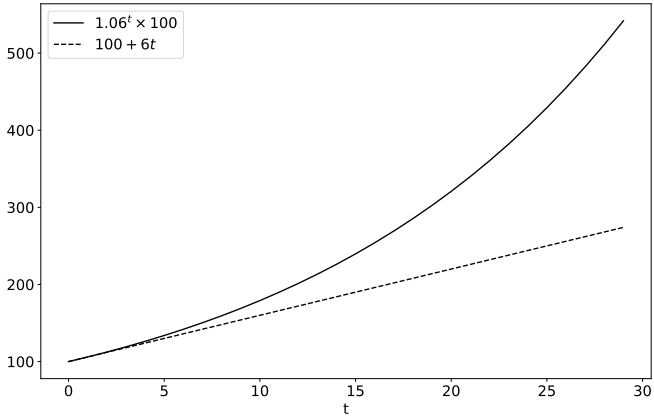


図 1.2.: 複利計算と単利計算

となる。つまり，年率 6% の預金で毎月 1 回利息を受け取れるとすれば，月率は $6\% \div 12 = 0.5\%$ となる。初期預金残高を 100 万円とすれば，1 年後，12 回目の利息を受け取った後には $1.005^{12} \times 100 = 106.1677\cdots$ 万円になっている。年率 6% の預金で毎日 1 回利息を受け取る場合 1 年後には，$(1+6\%\div 365)^{365} \times 100 = 1.06183\cdots$ 万円になる。当初資産額を S_0，利子率の年率を r，受け取り回数を N とすれば 1 年後の資産残高は

$$\left(1+\frac{r}{N}\right)^N S_0$$

となる。S_0 にかかる式を次のように変形する。

$$\left(1+\frac{r}{N}\right)^N = \left(1+\frac{1}{N/r}\right)^N = \left[\left(1+\frac{1}{N/r}\right)^{N/r}\right]^r$$

最右辺で r 乗される数が (1.2) と同じ形であることに注意してほしい：

$$n \longleftrightarrow \frac{N}{r}$$
$$\left(1+\frac{1}{n}\right)^n \longleftrightarrow \left(1+\frac{1}{N/r}\right)^{N/r}$$

$N \to \infty \Leftrightarrow N/r \to \infty$ だから，

$$\lim_{N\to\infty}\left(1+\frac{r}{N}\right)^N = \lim_{N/r\to\infty}\left[\left(1+\frac{1}{N/r}\right)^{N/r}\right]^r = e^r$$

となる。$N \to \infty$ というのは，すべての瞬間瞬間で利息を受け取れる仮想的な状況を表している。利息受け取りが時間的に連続した状況を考えているので，**連続複利計算**と呼ぶ。1 年後の資産残高は無限に大きくなる訳ではなく e^r 倍になることに注意しよう。なお，この 1 年間の変化

$$S_0 \longrightarrow S_0 e^r$$

について，対数差分を用いて変化率を求めると，

$$\begin{aligned}\log\left(S_0 e^r\right) - \log S_0 &= \log S_0 + \log e^r - \log S_0 \\ &= \log e^r \\ &= r\end{aligned}$$

となって利子率と一致する。瞬時的に受け取る利子率が（年率換算で）r であるとき（この利子率を「瞬時変化率」と呼んでおこう），1 年毎に資産は e^r 倍になる。対数差分は瞬時変化率 r に正確に一致する。

時間軸が飛び飛びの値を取る離散時間モデルに対して，時間軸が連続した値（つまり，実数値）を取るようモデル化の方法もある。そのようなモデルを「**連続時間モデル**」という。本書は主に離散時間モデルを扱うが，連続時間モデルの方が数学的な扱いが容易になることも多いため，離散時間・連続時間が状況に応じて使い分けられている。例えば，とある離散時間データの y_t から y_{t+1} への変化率を

$$\log y_{t+1} - \log y_t$$

で近似するのは，あたかも連続の時間軸上で成長したものと近似的に解釈して，瞬時変化率を計算していることになる。なお，瞬時変化率 r，あるいは同じことだが，離散時間の変化率 e^r が十分小さいときには，連続時間モデルと離散時間モデルの

違いは無視できるほどに小さくなる。なぜなら，

$$e^r = \sum_{n=0}^{\infty} \frac{r^n}{n!} = 1 + r + \frac{r^2}{2!} + \frac{r^3}{3!} + \cdots \approx 1 + r$$

が成り立つからである。

瞬時変化率を使うのが便利なのは，瞬時変化率に次のような性質があるからだ。

事実 1.1. 変数 x と y はそれぞれ瞬時的に g_x, g_y で変化している。このとき，

1. 積 xy の瞬時変化率は $g_x + g_y$，
2. 商 x/y の瞬時変化率は $g_x - g_y$，
3. 任意の実数 α に対して，べき x^α の瞬時変化率は αg_x 。

離散時間モデルではこのようなきれいな性質は成り立たない。例えば，$x_{t+1} = (1 + r_x)x_t, y_{t+1} = (1 + r_y)y_t$ とすると，

$$\frac{x_{t+1}y_{t+1} - x_t y_t}{x_t y_t} = r_x + r_y + r_x r_y$$

となり，余分な項 $r_x r_y$ が現れる。この余分な項は r_x, r_y がともに十分小さければ，無視できるほどに小さくなる。

注意 1.7. 離散時間的にしか観測されない経済現象を分析するにあたっては，次のような近似的なアプローチがよく使われる。

- 離散時間的に観測される現象を連続時間的に観測されるものと見做して連続時間モデルを構築する。数学的に使いやすいモデルが得られる。
- 離散時間モデルを構築した上で，成長率に関して連続時間モデルに類似した公式が成り立つものと近似して，数学的な議論を簡略化する。

□

1.3. プログラミング

この章では Python を関数電卓として使う方法を学ぶ。まだプログラミングと呼ぶほどのものでもないが，単一の式の計算が次章以降の基本になる。

Anaconda Prompt を開いて次のコマンドを入力しよう。なお，コマンドの最初に書かれた「>」は「プロンプト」と呼ばれる入力待ち状態を意味する記号である。読者はその後から入力すればよい。

```
ipython
```

次のような表示が見えれば成功だ。IPython のプロンプトは `In [n]:` という形式になっている。本書では IPython のプロンプトを省略している。 `In [n]:` の後にコードを入力して,「Enter」あるいは「Return」と書かれたキーを押せばコードが実行される。

```
Python 3.7.5 (default, Nov  1 2019, 02:16:23)  Type 'copyright',
'credits' or 'license' for more information IPython 7.11.0 --
An enhanced Interactive Python. Type '?' for help.
In [1]:
```

1.3.1. 基本の計算

加算 `+`, 減算`-`, 乗算`*`, 除算`/` の記号は使ったことがあるだろう。これらは `n + m` のように，数を 2 つ指定して使うので**二項演算子**（binary operator）と呼ばれる。

```
10 + 10.5
```

```
 20.5
```

```
3 - 5
```

```
 -2
```

```
2 * 3
```

```
 6
```

```
6 / 4
```

```
 1.5
```

```
2 ** 10
```

```
 1024
```

商（整数商）や剰余はそれぞれ二項演算子`//`, `%` を使う。

```
100 // 3
```

```
33
```

```
100 % 3
```

```
1
```

数 2 つで挟む二項演算子に対して，**単項演算子**（unary operator）は 1 つの数の前に置く。代表的なものには，負数を作る - がある。

```
-3 * 2
```

```
-6
```

```
2.5 * -2
```

```
-5.0
```

2 つ目のコードは，括弧を付けることで読みやすくできる。

```
2.5 * (-2)
```

```
-5.0
```

1.3.2. 優先順位と丸括弧

1 つの式の中に複数の演算子が使われているときは優先順位の高い順に計算が実行される。優先順位が同じ演算子が並んでいるときは左から順番に実行される。優先順位の詳細は，Python 公式ドキュメント「演算子の優先順位 [9]」の項を見てほしい。計算順序を変えたいときは丸括弧 ( ) で囲む（波括弧や角括弧は，この目的では使えない）。ほとんどは数学の慣習と同じなので，特に迷うこともないだろうと思う。例えば，次のような計算はすんなり理解できるだろう。

```
3 + 1 * 2
```

```
5
```

```
(3 + 1) * 2
```

[9] https://docs.python.org/ja/3/reference/expressions.html#operator-precedence

```
8
```

```
6 / 2 / 3
```

```
1.0
```

べき乗 **，負符号 -，積 *，和+ は，優先順位が高い順に並んでいる。したがって，次の 2 つのコードは同じ意味である。

```
1 + - 1 * - 3 ** 2
```

```
10
```

```
1 + (- 1) * (- (3 ** 2))
```

```
10
```

前者よりも後者のコードの方が読みやすいと感じる人が多いだろうと思う。優先順位表を知らなくても計算できるからだ。多少ムダであっても，適切に括弧を付けて人間が読みやすいコードを書くことを心がけてほしい。

なお，丸括弧にはもう 1 つ，長い計算の途中で改行を入れて可読性を高める，という使い方があることを覚えておいてほしい。次のような使用法である。

```
(1 + 2 + 3 + 4 + 5
 + 6 + 7 + 8 + 9 + 10)
```

```
55
```

同じ目的を達成するために括弧を使わず，バックスラッシュ \（日本語の環境では円マーク ¥）を使うこともできる。

```
1 + 2 + 3 + 4 + 5 \
  + 6 + 7 + 8 + 9 + 10
```

```
55
```

1.3.3. 変数

長い計算を実行するときには，重要な数字や計算途中の結果で意味があるものには名前を付けておくのが便利である。名前によって指し示そうとする対象（ここでは数字）をオブジェクト（object）と呼ぶ。オブジェクトに付けようとする名前のこ

とを慣例的に**オブジェクト名**とか**変数**（variable）と呼ぶ[10]。Python リファレンスにおける正式名称は**識別子**（identifier）あるいは**名前**（name）である。しばしば「変数に値を代入する」と表現される操作には= という二項演算子を使う。**右辺のオブジェクトに左辺の名前を付ける**，という操作である。

下の 2 行のコードが実行していることは，次の 2 つのことである。

1. 「10」という数字（整数）に x という名前をつける。
2. x という名前が指し示すものを参照する。

```
x = 10
x
```

```
10
```

今，私たちは IPython のコンソールで作業をしているので，x の中身である 10 が表示された。Python を実行している環境によっては，明示的に `print()` 関数を呼び出さないといけないかもしれない。

```
print(x)
```

```
10
```

変数は計算に使用することができる。

```
x * 10
```

```
100
```

成長の計算例

分析開始時点の GDP が 500 で，毎年 2% ずつ成長する経済を考えよう。3 年後の GDP を計算したい。

$$g = 0.02, \quad y_0 = 500, \quad t = 3$$

として，

$$y_t = (1 + g)^t y_0$$

である。素直に，数式をコードに置き換えて次のように書けばよい。

[10] ここに書いている定義はかなりいい加減なので，真に受けないように。

```
g = 0.02
y0 = 500
t = 3
yt = (1 + g) ** t * y0
yt
```

```
530.604
```

数式の表示ではわかりにくいという場合には，数字の意味を意識して次のようにしてもよい。

```
growth_rate = 0.02
GDP_0 = 500
years = 3
GDP_3 = (1 + growth_rate) ** years * GDP_0
GDP_3
```

```
530.604
```

どちらのコードがよりよいか，というのは一概には言えないのだが，数学的なモデルを数値的に分析するような状況では，数式との対応関係がはっきりと分かる前者のコードの方が保守がしやすい。一方，より汎用的なコードの開発という状況では，意味に応じた名前を付ける方が望ましいというケースもあるだろう。状況に応じた命名を心がけてほしい。

Python のオブジェクト名には次のルールがある。

- 大文字と小文字は区別する。
- 最初の文字として数字を使えない。
- 使える記号はアンダースコア _ だけ。
- Python のキーワードは変数名として使えない [11]。

問題 1.6. ネイピア数の定義

$$e = \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n$$

に基づいて，近似的に e を計算してみよう。n を $n = 100, 1000, 10000$ と大きくしていったとき，$\left(1+\frac{1}{n}\right)^n$ はどのような数字に近づいていくか。

[11] `True` , `False`, `import`, `from`, `as`, `if`, `else`, `for`, `in` など。全キーワードは Python の公式ドキュメントを参照のこと。 https://docs.python.org/ja/3/reference/lexical_analysis.html#keywords

1.3.4. エラー

Python はエラーが発生するとコードの実行を停止し，

〇〇 Error: エラーの理由

のような形式でユーザーにエラーに関する情報提供をするようになっている。「〇〇 Error」はエラーの種類ごとに付けられた名前である。例えば，未定義の名前を参照しようとすると，NameError が発生する。

```
print(abcde)
```

```
---------------------------------------------------------
NameError
Traceback (most recent call last)
<ipython-input-17-e8409d1cf4b6> in <module>
----> 1 print(abcde)
NameError: name 'abcde' is not defined
```

「NameError: name 'abcde' is not defined」というところを読めば，abcde というありもしない名前にアクセスしようとしていることが原因と分かる。このように，エラーメッセージをきちんと読めば自力で問題解決できることが多い。

あなたが Python の初心者であれば，まずは次の 2 つのエラーに慣れてしまおう。

- NameError: 存在しない名前を参照するエラー
  - 名前は定義されているか？[12]
  - スペルミスはないか？
  - 大文字・小文字の区別はできているか？

[12] 名前の定義の問題は Jupyter Notebook（後で紹介します）のような環境を使用する初心者を当惑させる原因になる。Jupyter Notebook を再起動して，途中からコードの実行を再開しようとするとこのエラーが出るのだ。確かに定義するコードはそこにあるのだけど，Python は知らないと言ってくる。再起動前に動いていたのであれば，コードを上から順番に実行し直させば解決するはずだ。

- SyntaxError: 書いたコードが Python の文法に則っていないというエラー
  - 名前のルールは守っているか？
  - キーワードを名前として使っていないか？
  - 開く括弧と閉じる括弧は正しく対応しているか？
  - その他，文法ルールは適切に守られているか？

これから多くのエラーに出会うことになるので，エラーメッセージを読む習慣を身につけよう。

問題 1.7. 次の 1〜5 のコードを実行したときに表示されるエラーメッセージに目を通し,「エラー名：エラーの理由」を書き取りなさい。エラーの原因について，自分の言葉で説明しなさい。

1. `3x = 10`
2. `True = 0`
3. `a-b = 0`
4. `print(abcde)`
5. `{3 * (2 + 4)} * 3`

1.3.5. 関数

脱線：関数とメソッド

複数の操作をまとめたり複雑な計算をするために「関数」を使うことができる。ユーザー（あなた）が関数を作ることもできる。複雑な計算を意味のある単位に分割し

て名前を付けておくことで，コードをクリーンに保つことができる。関数に定義された処理を実行することを「関数を呼び出す」とがある。2 つの関数呼び出しの方法があるので，一応ここで言及しておく。

- オブジェクトの外側から関数を作用させる方法。次の形式で呼び出す（obj はゼロ個でも，2 個以上でもよい。）

  ```
  function_name(obj)
  ```

- オブジェクトの内側から関数を呼び出す方法。ドットを使った次の形式で呼び出す。（other_obj はゼロ個でも，2 個以上でもよい。）

  ```
  obj.function_name(other_obj)
  ```

「オブジェクトとは何か」というような質問に対する正確な回答を本書には期待しないでほしいのだけれど，おおざっぱに言えば

「オブジェクト」=「データ」+「データに定義された動作」

というイメージを持っておけばよい。

オブジェクトが持つ「動作」はもちろん関数の一種であるが，これを特に，**メソッド**と呼ぶことが多い [13]。次のような例がある（何をやっているかはメソッドの名前を読めば分かる）。

```
1.5.as_integer_ratio()
```

```
(3, 2)
```

NumPy の関数

これ以上の例は今後のお楽しみということにしよう。Python は言語のコアの部分が非常に小さく，様々な興味のある処理を実行するためにライブラリと呼ばれる拡張機能を呼び出すことになる。ここで使うのは，**NumPy** という数値計算用のライブラリが持っている関数だ。Python の標準ライブラリ（必ずインストールされているライブラリ）には含まれていないが，数値計算を行うときの事実上の標準に

[13] なお，ドットが付けば必ずメソッドかというとそうでもない。オブジェクトの中にさらに変数（**属性**という）が格納されている場合もある。説明が必要になったときに紹介しよう。

なっている[14]。Anaconda を利用しているなら，**NumPy** はインストールされているはずだ。

ライブラリの拡張機能を呼び出すには，`import` 文を書く。いくつかの書き方があるが，多くの人は **NumPy** を次のようにロードする。

```
import numpy as np
```

このコードを実行しても見かけ上は何も起こらないが，**NumPy** の関数を `np.function_name()` の形式で呼び出すことができるようになる。例えば，自然対数は `np.log()`，指数関数は `np.exp()` である。

```
np.log(10)
```

```
2.302585092994046
```

```
np.exp(1)
```

```
2.718281828459045
```

ここで **NumPy** の数学関数を網羅することはできないので，必要に応じて公式リファレンスを調べてほしい[15]。種々の数学関数以外にも，重要な定数が定義されている。

```
np.pi
```

```
3.141592653589793
```

```
np.e
```

```
2.718281828459045
```

例 1.5. 成長率の近似公式（命題 1.3）を確認してみよう。ここでは，

$$y_0 = 300, \quad y_1 = 306$$

という数値例を使う。対数差分と通常の変化率の定義とが近い値を取ることを確認できる。

[14] Python をインストールすると必ずインストールされる標準ライブラリにも **math** や **statistics** という数値計算目的のライブラリがある。**NumPy** を使えない環境で仕事をしなければならなくなったときに使い方を調べればよい。

[15] https://docs.scipy.org/doc/numpy/reference/routines.math.html

```
y0 = 300
y1 = 306
np.log(y1) - np.log(y0)
```

```
0.019802627296179764
```

```
(y1 - y0) / y0
```

```
0.02
```

問題 1.8. 他の数値例を用いて例 1.5 と同様のことを確かめなさい。y_0 から y_1 への変化率がどの程度の大きさであれば，命題 1.3 の近似公式は実用上使えそうか。

2. 物価指数とインフレーション

2.1. 概要

この講義では以下のことを学ぶ。

- 理論
  - ベクトルの復習
  - 価格指数の定義
  - 連鎖指数の計算方法
- プログラミング
  - リスト，タプル
  - **NumPy** の配列
  - **NumPy** の関数
  - **Matplotlib** を用いた可視化

実質 GDP の定義は次章を参照のこと。

2.2. 理論：ベクトル

数の集合として実数全体の集合 $\mathbb{R}$ を考える。ひとまず，ベクトルとは一定の個数分だけ数字を並べたものと考えておこう。

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}, \qquad x_1, x_2, \dots, x_d \in \mathbb{R}$$

自然数 d は並べた数字の個数を表す。これは状況に応じて必要な数を固定する。縦に並べると余白が増えてしまうので，横に並べて省スペース化できる方が便利で

ある。

$$\boldsymbol{x} = (x_1, x_2, \ldots, x_d).$$

縦に数字を並べると要素ごとの足し算が見やすくなったり，サイズが $n \times 1$ である行列と同一視できるので行列との積の計算に大変都合がよかったりといった利点がある。本書では，状況に応じて都合のよい方を使う。どうしても「数を横に並べたベクトル（サイズが $1 \times n$ の行列）」を書く必要があるときには，

$$\begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}$$

のように，角括弧でコンマを付けずに書くことにしよう。なお，ベクトルを表す記号には太字を用いることが多いが，これも絶対に必要という訳ではない。

固定した d について，ベクトルの成分 $x_1, \ldots, x_d$ の組み合わせは無数に存在する。これらのベクトルを全部を集めた集合をベクトル空間と呼ぶ。$\mathbb{R}$ に含まれる数字が d 個並んでいるので，$\boldsymbol{x}$ を d 次元ベクトル，$\mathbb{R}^d$ を d 次元ベクトル空間という。d は次元（dimension）に由来する。

ベクトルの演算

ベクトル空間においては，和（足し算）とスカラー倍（定数倍）の 2 つの演算が基本的である。$\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^d$，$\alpha, \beta \in \mathbb{R}$ に対して，

$$\alpha\boldsymbol{x} + \beta\boldsymbol{y} = \alpha \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} + \beta \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_d \end{bmatrix} = \begin{bmatrix} \alpha x_1 + \beta y_1 \\ \alpha x_2 + \beta y_2 \\ \vdots \\ \alpha x_d + \beta y_d \end{bmatrix}$$

と定義する。最右辺も数字を d 個並べたベクトルであることに注意しよう。したがって，$\alpha\boldsymbol{x} + \beta\boldsymbol{y}$ も $\mathbb{R}^d$ の元である。

例 2.1. $d = 3$ としよう。

$$\boldsymbol{x} = \begin{bmatrix} 1 \\ 3 \\ -3 \end{bmatrix}, \quad \boldsymbol{y} = \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix}$$

に対して，

$$\boldsymbol{x}+\boldsymbol{y}=\begin{bmatrix}3\\3\\-2\end{bmatrix}$$

$\alpha=-1$ であれば，

$$\alpha\boldsymbol{x}=\begin{bmatrix}-1\\-3\\3\end{bmatrix}$$

ベクトル同士の引き算は，和とスカラー倍を組み合わせて，

$$\boldsymbol{x}-\boldsymbol{y}=\boldsymbol{x}+(-1)\boldsymbol{y}$$

とできる。これは成分ごとに引き算をしているだけである。

内積

ベクトル $\boldsymbol{x},\boldsymbol{y}\in\mathbb{R}^d$ に対して，

$$\boldsymbol{x}\cdot\boldsymbol{y}=x_1y_1+x_2y_2+\cdots+x_dy_d=\sum_{i=1}^{d}x_iy_i$$

で定義される $\boldsymbol{x}\cdot\boldsymbol{y}\in\mathbb{R}$ を内積という [1]。経済学の文脈では，価値を計算する場合に内積が使われる。

例 2.2. 経済に存在する財を 3 タイプに分類したとしよう。財 1，財 2，財 3 の価格を $\boldsymbol{p}=(p_1,p_2,p_3)$，取引量を $\boldsymbol{x}=(x_1,x_2,x_3)$ とする。このとき，取引総額は

$$\boldsymbol{p}\cdot\boldsymbol{x}=p_1x_1+p_2x_2+p_3x_3$$

である。

内積はベクトル同士の角度を表すのに使うことを覚えているかもしれない。内積はベクトルの直交関係を規定する重要な概念だが，そのような解釈は必要になったら説明することにしよう [2]。

[1] 分野によって，$(\boldsymbol{x},\boldsymbol{y})$, $\langle\boldsymbol{x},\boldsymbol{y}\rangle$, あるいは $\langle\boldsymbol{x}|\boldsymbol{y}\rangle$ などと書く。
[2] 今後，直交行列に関する議論が必要になるので，そのときに説明する。

注意

成分ごとの掛け算や割り算が使われることは少ないので，ここでも特別な記号は用意しない。しかし，**NumPy** のベクトル（後述）同士では乗算・除算が定義されていて，通常の乗算記号* や除算記号 / を用いる。これらは成分ごとの演算であることに注意しておこう。

2.3. 理論：価格指数

2.3.1. パーシェ指数とラスパイレス指数

物価指数計算の基本は，各財の価格変化率の加重平均である。パーシェ式とラスパイレス式という計算方法が代表的である。

財・サービスの種類を $i = 1, 2, \ldots, N$ の N 個の財グループに分類する。個々の財グループについては代表的な価格が定まっていると考える [3]。t 期の財グループ i の価格を $p_{t,i}$ と書く。1 つのグループについて，$t = 0$ 時点からの価格の粗変化率は

$$\frac{p_{t,i}}{p_{0,i}}$$

と書ける。

複数の財が存在する経済で，マクロ経済の価格を代表する「物価」，あるいはその変化，をどのように定義すればよいだろうか。ベクトルとベクトルの割り算は定義していないので，

$$\frac{\boldsymbol{p}_t}{\boldsymbol{p}_0} \qquad \text{(誤り !)}$$

とする訳にはいかないし，単純に算術平均

$$\frac{1}{N}\sum_{i=1}^{N}\frac{p_{t,i}}{p_{0,i}} \qquad \text{(一般的には正しくない !)}$$

を取ることでは平均的な物価の変化を捉えることもできない。マーケットの小さい財の価格が大幅に変化することを想像すれば問題点が明確になるだろう。マクロの

[3] 実務上は各品目について平均価格を計算するというステップと，全品目を総合してマクロ経済全体の価格指数を計算するというステップが必要である。ここでは，2 つ目のステップについての議論であると考えればよい。

経済活動にほとんど影響していない（マーケットが小さいから）にも関わらず，算術平均に大きな影響を与えてしまうからだ。

すべての財に共通の重み $1/N$ をかけていることが問題なので，マーケットの大きさに応じた重みを掛ければよい。つまり，

$$\sum_{i=1}^{N} w_i \frac{p_{t,i}}{p_{0,i}}$$

のように計算すればよい。なお，重み $w_i, i = 1, \ldots, N,$ は次の性質を満たす。

$$\sum_{i=1}^{N} w_i = 1.$$

パーシェ式とラスパイレス式の指数には重みの定義において違いがある。

ラスパイレス指数

財 i の $t = 0$ 時点における取引量を $x_{0,i}$，$i = 1, \ldots, N$ についてベクトルにまとめて $\boldsymbol{x}_0 = (x_{0,1}, \ldots, x_{0,N})$ と書く。この当時の 財 i の市場価値（取引額）は $p_{0,i}x_{0,i}$ だった。すべての財について価値を合計したものは，内積を用いて次のように書くことができる。

$$\sum_{j=1}^{N} p_{0,j}x_{0,j} = \boldsymbol{p}_0 \cdot \boldsymbol{x}_0$$

これは，分析対象とするすべての財について $t = 0$ 期の総取引額を計算したものである。この総取引額に占める財 i のシェアを w_i^L としよう。すなわち，

$$w_i^L = \frac{p_{0,i}x_{0,i}}{\sum_{j=1}^{N} p_{0,j}x_{0,j}} = \frac{p_{0,i}x_{0,i}}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_0}$$

この w_i^L は「重み」が持つべき性質を持っている。なぜなら，

$$\sum_{i=1}^{N} w_i^L = \sum_{i=1}^{N} \left(\frac{p_{0,i}x_{0,i}}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_0} \right) = \frac{\boldsymbol{p}_0 \cdot \boldsymbol{x}_0}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_0} = 1.$$

このように定義される重み $\boldsymbol{w}^L = (w_1^L, \ldots, w_N^L)$ を用いて計算する加重平均を**ラス**

パイレス指数 (Laspeyres index) と呼ぶ。

$$\begin{aligned}(\text{ ラスパイレス指数 }) &= \sum_{i=1}^{N} w_i^L \frac{p_{t,i}}{p_{0,i}} \\ &= \sum_{i=1}^{N} \left(\frac{p_{0,i}x_{0,i}}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_0} \right) \frac{p_{t,i}}{p_{0,i}} \\ &= \frac{\sum_{i=1}^{N} p_{t,i}x_{0,i}}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_0} \\ &= \frac{\boldsymbol{p}_t \cdot \boldsymbol{x}_0}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_0}. \end{aligned} \tag{2.1}$$

通常，このようにして得られた数に 100 を掛けて，基準化する。

パーシェ指数

財 i の t 期における取引量を $x_{t,i}$ と書くことにする。ベクトルにまとめて，$\boldsymbol{x}_t = (x_{t,1}, \ldots, x_{t,N})$ とする。 次に考える重みは次のようなものである。

$$w_i^P = \frac{p_{0,i}x_{t,i}}{\sum_{j=1}^{N} p_{0,j}x_{t,j}} = \frac{p_{0,i}x_{t,i}}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_t}.$$

これは t 期における市場の規模を，0 期の価格で評価したものになっている。この w_i^P は「重み」が持つべき性質を持っている。すなわち，

$$\sum_{i=1}^{N} w_i^P = \sum_{i=1}^{N} \left(\frac{p_{0,i}x_{t,i}}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_t} \right) = \frac{\boldsymbol{p}_0 \cdot \boldsymbol{x}_t}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_t} = 1.$$

このように定義される重み $\boldsymbol{w}^P = (w_1^P, \ldots, w_N^P)$ を用いて計算する加重平均を**パーシェ指数** (Paasche index) と呼ぶ。

$$
\begin{aligned}
(\text{パーシェ指数}) &= \sum_{i=1}^{N} w_i^P \frac{p_{t,i}}{p_{0,i}} \\
&= \sum_{i=1}^{N} \left(\frac{p_{0,i} x_{t,i}}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_t} \right) \frac{p_{t,i}}{p_{0,i}} \\
&= \frac{\boldsymbol{p}_t \cdot \boldsymbol{x}_t}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_t}
\end{aligned} \tag{2.2}
$$

通常，このようにして得られた数に 100 を掛けて，基準化する。

ラスパイレス指数とパーシェ指数の共通点と相違点

ラスパイレス指数とパーシェ指数ともに次のような形式をもつ物価変化の指標である。

$$
\text{物価変化率} = \frac{\boldsymbol{p}_t \cdot \square}{\boldsymbol{p}_0 \cdot \square}
$$

パーシェ式の物価指数は $\square$ の中に比較年（t）の数量を用いる。一方，ラスパイレス式の物価指数は $\square$ の中に基準年（0）の数量を用いる。パーシェとラスパイレスは次のようなニモニックで覚えるとよい [4]。

パーシェ式	Paasche	Present （今）の数量を使う
ラスパイレス式	Laspeyres	Long time ago （昔）の数量を使う

価格を評価するために補助的に用いる数量データの違いによって，2 つの指数には情報収集コストに違いが生じる。

問題 2.1. ラスパイレス式の物価指数はパーシェ式の物価指数よりも物価の変化を迅速に捉えることができる。これは，なぜか。情報収集のコストの観点から説明しなさい。

[4] E. Wayne Nafziger, *Economic Development*

2.3.2. バイアス

ラスパイレス式であってもパーシェ式であっても，特定の一時点の数量を用いて物価比較をする限り，価格の変化に対する消費の変化を完全に捉えることは容易ではない。なぜなら，価格が上昇した財の需要は減少して相対的に値下がりした別の財の需要が増加するのが通常の消費行動であるにも関わらず，上記の 2 つの物価指数はこのような代替効果を無視しているからだ。実際，ラスパイレス式の指数には物価の上昇を過大評価する傾向があり，パーシェ式の指数には過小評価する傾向があると言われている。

ある財 i が急激に値上がりしたとして，その財から別の財への代替が進んだとする。ラスパイレス指数の重み

$$w_i^L = \frac{p_{0,i}x_{0,i}}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_0}$$

には代替による効果は反映されないので，$p_{t,i}/p_{0,i}$ の影響が大きく残る。一方，パーシェ指数の重み

$$w_i^P = \frac{p_{0,i}x_{t,i}}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_t}$$

は，財 i の価格 $p_{t,i}$ が上昇したことで起こる需要減で $x_{t,i}$ が減少する効果は反映するものの，重みを計算する価格は $p_{0,i}$ という小さい数字のままである。したがって，$p_{t,i}/p_{0,i}$ に対する重みが小さくなってしまう。

フィッシャー指数

ラスパイレス指数の上方バイアス，パーシェ指数の下方バイアスという欠点を補うために，これらの平均を取った物価指数が考案されている。**フィッシャー指数**は

パーシェ式とラスパイレス式の相乗平均を用いて，次のように定義される。

$$(\text{フィッシャー指数}) = \sqrt{\underbrace{\frac{\boldsymbol{p}_t \cdot \boldsymbol{x}_t}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_t}}_{\text{パーシェ}} \times \underbrace{\frac{\boldsymbol{p}_t \cdot \boldsymbol{x}_0}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_0}}_{\text{ラスパイレス}}}.$$

通常，このようにして得られた数に 100 を掛けて，基準化する。

2.3.3. インフレーションとデフレーション

ここまでに計算した物価指数，時点 0 を基準にして，時点 t の物価がどれくらい高いかを示す指標である。$\boldsymbol{x}_* = \boldsymbol{x}_0$ or $\boldsymbol{x}_t$ として，価格指数を P_t と書こう。

$$P_t = \frac{\boldsymbol{p}_t \cdot \boldsymbol{x}_*}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_*}$$

このようにすれば，平均物価の変遷を表す時系列を作ることが出来る。

$$P_0 = 1, \quad P_1, \quad P_2, \quad \cdots, \quad P_t, \quad \cdots$$

P_t が継続的に上昇し続ける状況を**インフレーション**，継続的に下落し続ける状況を**デフレーション**と呼んでいる。マクロ経済に多数ある財が総じて値上げする状況でインフレーションが起こり，総じて値下げされる状況でデフレーションが起こる。「物価の安定」といったときには，インフレーションのスピード（**インフレ率**）

$$\frac{\Delta P_t}{P_{t-1}} = \frac{P_t - P_{t-1}}{P_{t-1}} = \frac{P_t}{P_{t-1}} - 1$$

がゼロに近い正の値になるような政策的な取り組みを表すことが多い [5]。

2.3.4. 固定基準年方式と連鎖方式

上のように定義した P_t は t が大きくなるにつれて，経済活動の実態に合わなくなってくる。例えば，ラスパイレス指数の場合，

$$P_t^{\text{Laspeyres}} = \frac{\boldsymbol{p}_t \cdot \boldsymbol{x}_0}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_0}$$

[5] インフレ率がゼロになってしまうと外的なショックの影響で不況になったときに金融緩和余地がなく苦境に立たされてしまう。

の分子にある $\boldsymbol{p}_t \cdot \boldsymbol{x}_0$ は価格 $\boldsymbol{p}_t$ のもとでは全く現実味のない数量ベクトル $\boldsymbol{x}_0$ を使ってしまっているかもしれない。パーシェ指数の場合にも，

$$P_t^{\text{Paasche}} = \frac{\boldsymbol{p}_t \cdot \boldsymbol{x}_t}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_t}$$

の分母にある $\boldsymbol{p}_0 \cdot \boldsymbol{x}_t$ が価格 $\boldsymbol{p}_0$ のもとでの経済活動では起こりそうにもない水準になるかもしれない。経済に新しい財がどんどん追加されたり，財の質的な向上と価格下落が著しい時代にあっては，このような問題が深刻になる。基準数量 $\boldsymbol{x}_0$ や基準価格 $\boldsymbol{p}_0$ を固定した前述のような計算方式を**固定基準年方式**と呼ぶ。このような問題に対処する 1 つの方法は，数量ベクトル $\boldsymbol{x}_0$ を頻繁に改定することである。例えば，代表的な物価指数である消費者物価指数では，マクロ経済における標準的な購入量を 5 年ごと見直して，ラスパイレス指数を計算している。基準年を変更することを，基準改定という。改定前と改定後の数値は直接接続しないことに注意をしなければならない。

固定基準年方式に代わる方式として，連鎖方式と呼ばれる算式が用いられることもある。連鎖方式の指数は，前年度の連鎖指数に前年度基準の価格指数を掛けることで計算される。つまり，

$$\text{ラスパイレス式連鎖指数}_t = \text{ラスパイレス式連鎖指数}_{t-1} \times \frac{\boldsymbol{p}_t \cdot \boldsymbol{x}_{t-1}}{\boldsymbol{p}_{t-1} \cdot \boldsymbol{x}_{t-1}}$$

$$\text{パーシェ式連鎖指数}_t = \text{パーシェ式連鎖指数}_{t-1} \times \frac{\boldsymbol{p}_t \cdot \boldsymbol{x}_t}{\boldsymbol{p}_{t-1} \cdot \boldsymbol{x}_t}$$

$$\text{フィッシャー式連鎖指数}_t = \text{フィッシャー式連鎖指数}_t \times \sqrt{\frac{\boldsymbol{p}_t \cdot \boldsymbol{x}_{t-1}}{\boldsymbol{p}_{t-1} \cdot \boldsymbol{x}_{t-1}} \times \frac{\boldsymbol{p}_t \cdot \boldsymbol{x}_t}{\boldsymbol{p}_{t-1} \cdot \boldsymbol{x}_t}}$$

GDP デフレーターはパーシェ式の連鎖指数である。なお，連鎖指数の初期値（$t = 0$ の値）は 1 と設定する（指数を 100 で基準化する場合は 100）。

$$\text{連鎖指数}_0 = 1$$

関連する言葉の整理をしておこう。

比較年 物価の水準を観察したい年のこと。上の式では t に相当する。

参照年 指数を 1（または 100）に基準化する時点のこと。上の例では，0 期に相当する。

基準年 比較年の物価上昇率を計算するための基準となる年のこと。固定基準年方

式の場合は参照年と同じで，0 期を指す。連鎖方式の場合は比較年の前年，つまり $t-1$ が基準年である。

GDP 統計には，さらに，体系基準年という概念がある。GDP 統計を作成するための基礎統計（産業連関表，国勢統計など）の更新に合わせて GDP 統計も基準数量が変更される。基礎統計が作成された年が体系基準年である。

2.4. プログラミング: NumPy 入門

この章では，複数の数字をまとめたオブジェクトである，リストとタプルを導入し，その後 **NumPy** の **ndarray** オブジェクトを説明する。数学的なベクトルや行列を **ndarray** で表現できるので，Python を用いた数値計算では必須の話題である。

2.4.1. リストとタプル

Python でベクトルや時系列を表現するための基礎となるデータ形式はリストやタプルと呼ばれるオブジェクトである。「基礎となる」と言ったのは，これらをそのまま使う訳ではないからだ。しかし，リストやタプルをすっ飛ばしてベクトルの表現を説明することはできないので，しばらく辛抱してほしい。

リスト

リストは複数の種類のデータをひとまとめにしたデータ形式である。コンマでつないで角括弧で括る。

```
x = [5, 8, 9, -1]
x
```

```
[5, 8, 9, -1]
```

和やスカラー倍は期待通りに動かないので，これはベクトルのようには使えない。

```
x + x
```

```
[5, 8, 9, -1, 5, 8, 9, -1]
```

```
3 * x
```

```
[5, 8, 9, -1, 5, 8, 9, -1, 5, 8, 9, -1]
```

問題 2.2. リスト同士の和，リストと自然数の積は Python ではどのように定義されているか。色々な入力を試して実験し，自分の言葉で説明しなさい。

問題 2.3. リストと小数の積を実行しようとするとどのような結果になるだろうか。結果を予想して，実行し，結果を記録しなさい [6]。

```
3.5 * x
```

Python のリストを数学的なベクトルを表現するために使いにくい理由は，リストはベクトルよりも遥かに柔軟なデータ構造だからだ。数学的なベクトルは同種の数字しか並べることができないが [7]，Python のリストは様々な種類のオブジェクトを並べることができる。

```
y = [10, 4.3, "Hello", [1, 2, 3]]
y
```

```
[10, 4.3, 'Hello', [1, 2, 3]]
```

上で定義したリスト y は次のような要素を持つ。

- 左から 1 つ目の要素は，整数 10
- 2 つ目の要素は小数 4.3

[6] 面倒かもしれないが,「結果を予想」するというステップを飛ばさないでほしい。あなた自身が自分のためのコードを書くときに予想する能力が必要になる。

[7] 前節の解説では「同種」の部分は完全に割愛している。具体的には実数，複素数，有理数などである。気になった読者は線形代数学の教科書を参照せよ。

- 3 つ目の要素はテキスト "Hello"（1 重引用符や 2 重引用符で囲ったテキストは「文字列」と呼ばれる種類のオブジェクトになる）
- 4 つ目の要素はリスト [1, 2, 3]。そのリストには整数が 3 つ並んでいる。

要素に何が入るのかが分からないのであれば，要素ごとの加算や乗算を定義できないことは容易に想像が付くだろう。リストは意味のあるひとまとまりのデータを「記録」するために使うことができる。しかし，同種の数字だけ並べて「計算」したいならもっとよい方法がある。

個別のデータを取得する方法が気になることだろう。角括弧を使って，インデックス（データにおける要素の位置）を次のように指定すればよい。

```
x[0]
```

```
5
```

```
x[-1]
```

```
-1
```

```
y[-2]
```

```
'Hello'
```

```
y[3][1]
```

```
2
```

次のことを覚えておこう。インデックスと中身を見比べてみよう。

- インデックスは 0 から始まる整数である。
- リストの最後の要素から数える場合には，-1, -2, ... という負のインデックスを使うことができる。
- リストの長さは len() で取得できる。

```
len(y)
```

```
4
```

リストの内容を変更するときには要素取得と同じ記法を用いる。

```
x[0] = 0
x
```

```
[0, 8, 9, -1]
```

オブジェクトに対応付けられた x や y という記号は，オブジェクトそのものではなくオブジェクトを呼びだすための名前に過ぎない。これは次のような実験からよく分かる。オリジナルの x には手を触れずにコピーした z だけを編集しようとしたのだけど，x にまで修正が及んでしまっている。このコードは失敗だ。

正しくは，次のように書かなければいけない。コロンは「スライス」と呼ばれる操作に関連している。いずれ詳しく説明することになるだろう。

なお，要素のないリストや，単一要素のリストも作ることができる。

```
[]
```

```
[]
```

```
[1]
```

```
[1]
```

タプル

タプルとリストの違いとして次の 2 点を押さえておけばよい。

- 丸括弧で括る。
- あとから要素を変更できない。

```
u = (1, 2, 3)
u
```

```
(1, 2, 3)
```

```
u[2]
```

```
3
```

丸括弧は計算順序の変更という他の意味があるので，単一の要素を持つタプルを作成するときには注意が必要である。

```
()
```

```
()
```

```
(1,)
```

```
(1,)
```

問題 2.4. タプルの要素を変更しようとするとどのような結果になるだろうか。下のコードを実行し，結果を記録しなさい。

```
u[0] = 0
```

数値計算だけに限定すれば，タプルを自分で作らなければならない状況というのは少ないかもしれない[8]。しかし，出力結果がタプルとなるケースは非常に多いので出力には慣れておこう。

2.4.2. NumPy の配列

さて，準備が整ったので本題に入ろう。

配列（array）というのは，同じ種類の要素（通常は，数字）を並べたリスト様のオブジェクトである。難しいことを特に意識する必要もないが，要素がすべて同じ種類でなければならないという制約があるおかげで要素の取得や要素に対する演算を効率的に実行できるということだけ覚えておこう。

Python で数値計算をするときには **NumPy** の **ndarray** という形式の配列を用いる。前章で紹介したとおり，**NumPy** を使えるようにするにはいつものインポート文を実行する必要がある。繰り返し書いているうちに無意識に打てるようになる。

[8] 多くの場合はリストで置き換えられる。リストでなく必ずタプルを使わないといけないケースには，辞書と呼ばれるデータ形式のキーにしたい場合などがある。本書では，このようなケースは（多分）出てこないと思う。

```
import numpy as np
```

ベクトルを表現する配列を作るには，np.array() をという関数をリストに適用すればよい。すべての要素に小数点がついていることに注意しよう。配列は同じ種類の数字が並んだものなので，整数と小数が混ざっている場合には小数に変換される。

```
x = np.array([1, 2, 3.0])
x
```

```
array([1., 2., 3.])
```

ndarray の nd は N-dimensional （N 次元）の略である。数学的なベクトル空間の次元と，配列の次元は意味合いが異なるので注意が必要である。例えば，

- 0 次元配列はスカラー
- 1 次元配列はベクトル
- 2 次元配列は行列

のようになる。数学的に N 次元ベクトルと呼ばれるものは，**NumPy** では要素数 N の 1 次元配列を用いて表現できる。配列の次元は ndim, 数学的な次元（サイズ）は shape で調べることができる。

```
x.ndim
```

```
1
```

```
x.shape
```

```
(3,)
```

ドットの後に ndim や shape と書いて呼び出している変数は**属性**（attribute）と呼ばれるオブジェクトの追加情報である。**ndarray** の **shape** 属性がタプルになっていることに注意しよう。

配列の演算

NumPy 配列同士の演算はおおむね普通の算術演算と同様に実行できるので，混乱はほとんどないだろう。原則的に要素ごとに計算される。

```
x = np.array([1, 2, 3.])
y = np.array([-1, 1, 4.])
2 * x + 0.5 * y
```

```
array([1.5, 4.5, 8. ])
```

スカラーとの演算もうまく定義されている。

```
np.array([1, 2, 3.]) + 10
```

```
array([11., 12., 13.])
```

低次元配列と高次元配列の演算にはブロードキャスティングという処理が実行されて shape が揃えられる。例えば，1 次元配列（ベクトル）と 2 次元配列（行列）の足し算は次のように振る舞う。

```
A = np.array([[0.1, 0.2, 0.3],
              [0.4, 0.5, 0.6],
              [0.7, 0.8, 0.9]])
A
```

```
array([[0.1, 0.2, 0.3],
       [0.4, 0.5, 0.6],
       [0.7, 0.8, 0.9]])
```

```
x + A
```

```
array([[1.1, 2.2, 3.3],
       [1.4, 2.5, 3.6],
       [1.7, 2.8, 3.9]])
```

低次元配列がスカラー以外の場合には最細の注意が必要であるが，ここで詳しく扱うには複雑すぎるので省略する（行列の定義の仕方を見ておけばよい。）。演算を行う前に十分テストをしたほうがよい。はじめから次元を合わせておくともっと安全だろう。

なお，ブロードキャスティングが適用できる場合を除いて，サイズの異なる配列同士の計算はできない。

問題 2.5. 次の計算を実行しようとするとどのような結果になるだろうか。下のコードを実行し，結果を記録しなさい。

```
np.array([1, 2, 3.]) + np.array([1, 2, 3., 4])
```

数学的には定義されていないベクトル間の演算も NumPy 配列に対して使用できる場合がある。例えば，乗算や除算などが要素ごとの演算として定義されている。

```
x * y
```

```
array([-1.,  2., 12.])
```

```
x / y
```

```
array([-1.  ,  2.  ,  0.75])
```

配列に作用する関数

配列を入力に取る数学関数には，次のようなタイプのものがある。

- 入力配列の要素ごとに関数値を計算し，入力と同じサイズの配列を出力するもの。
- 入力配列全体をひとまとめにして関数値を計算し，単一の数を出力するもの。
- 入力配列全体をひとまとめにして関数値を計算し，入力と同じサイズの配列を出力するもの。

1 つ目のタイプの関数には `np.log()` や `np.exp()` などがある。

```
np.log(x)
```

```
array([0.        , 0.69314718, 1.09861229])
```

```
np.exp(A)
```

```
array([[1.10517092, 1.22140276, 1.34985881],
```

```
       [1.4918247 , 1.64872127, 1.8221188 ],
       [2.01375271, 2.22554093, 2.45960311]])
```

2 つ目のタイプの関数の代表例は np.sum() や np.max() などである。

```
np.sum(x)
```

```
6.0
```

```
np.max(x)
```

```
3.0
```

このタイプの関数は,（すべてかどうかは分からないが）同名のメソッドを持っているので，次のように呼び出すこともできる。

```
x.max()
```

```
3.0
```

```
x.argmax()
```

```
2
```

2 次元以上の配列にこのタイプの関数を適用すれば，特定の次元にだけ適用して情報を集約するために使うことができる。例えば，次の 2 次元配列が 3 人の学生の英語と数学の試験の成績と解釈しよう。行方向（横方向）に見れば，各学生の英語・数学の 2 つの数字が並んでいる。列方向（縦方向）に見れば，各科目の 3 人の成績が並んでいる。

```
B = np.array([[90,  90],
              [100, 60],
              [80,  90]])
B
```

```
array([[ 90,  90],
       [100,  60],
       [ 80,  90]])
```

次のコードは各科目の平均得点（axis=0 と書くと，列（第 0 次元）の平均を計算する）と各学生の平均得点（axis=1 と書くと，行（第 1 次元）の平均を計算する）を計算している。

```
B.mean(axis=0)
```

```
array([90., 80.])
```

```
B.mean(axis=1)
```

```
array([90., 80., 85.])
```

3 つ目のタイプの関数には，累積和 cumsum()，累積積 cumprod() などがある。このタイプの関数も配列に付随するメソッドとして呼び出せる場合がある。

```
np.cumsum(x)
```

```
array([1., 3., 6.])
```

```
A.cumprod(axis=0)
```

```
array([[0.1  , 0.2  , 0.3  ],
       [0.04 , 0.1  , 0.18 ],
       [0.028, 0.08 , 0.162]])
```

2 つの配列に作用する数学関数

ここまでは単一の配列に対する操作を紹介した。ベクトルの内積やデータの共分散など，複数の配列（ベクトル）に対して関数を適用したいケースもある。出力は単一の数字，入力と同じサイズのベクトル，1 つ次元の高い配列（行列）になる場合がある。

```
np.dot(x, y)
```

```
13.0
```

```
np.maximum(x, y)
```

```
array([1., 2., 4.])
```

```
np.minimum(x, y)
```

```
array([-1.,  1.,  3.])
```

```
np.cov(x, y)
```

```
array([[1.         , 2.5        ],
       [2.5        , 6.33333333]])
```

なお，dot() を用いた計算はここではうまく内積を計算できたが，これはどちらも 1 次元の配列だからだ。x, y が同じシェイプを持つ 2 次元配列（ベクトル）になっている場合には，行列の積を計算しようとして失敗する。私はこのような振る舞いに使いにくさを感じているし，行列積を計算するだけなら@ 演算子を使う方が美しく書ける。今後本書で dot() に出会うことはもうないだろう。さようなら。

問題 2.6. dot() を使わずに内積を計算する方法を説明しなさい。

配列を生成する関数

基本の関数　配列を作成するために毎回リストを書かなければいけないのは手間が大きいので，よく使う配列を生成する関数が用意されている。以下の関数はタプルまたは単一の数を入力として受け取って配列のサイズとして用いる。それぞれどのような配列が出力されるかを確認しよう。

```
np.ones(2)
```

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

```
np.zeros((2, 3))
```

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

シミュレーション分析では配列の値を特に設定せずに，サイズだけ指定したい場合がある。empty() という関数を使う。値は適当に決まる [9]。

[9] このような関数はシミュレーション結果を保存する配列を作るために用いる。結局はすべての値が上書きされることになるので，最初に初期化する（例えばゼロを代入する）作業を省いても問題にならない。

```
np.empty((2, 3))
```

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

問題 2.7. `np.ones_like()`, `np.zeros_like()`, `np.empty_like()` はどのように使用するか。使用方法を調べて使ってみなさい。そして，それぞれの用途，使用方法を自分の言葉で説明しなさい。ヒント：IPython では関数名の前か後に ? をつければ関数のドキュメントを読むことができる。例えば，`np.empty_like()` は次のようにして表示できる。ドキュメントから抜けるにはキーボードの「q」を押下する。

```
np.empty_like?
```

乱数の生成　ランダムに数字を生成して配列を生成したい場合がある。**numpy.random** というモジュールで定義された関数を使うことができる。

$[0, 1)$ の一様乱数を生成するには，`np.random.random()` を用いる [10]。

```
np.random.random((3, 2, 2))
```

```
array([[[0.54340494, 0.27836939],
        [0.42451759, 0.84477613]],
```

[10]「乱数」というのは，コンピュータ上で確率変数の実現値を擬似的に計算したものである。あたかもランダムに見える数列というくらいに考えておけばよい。なお，プログラムの開発中にはランダムさが邪魔になる場合がある。`np.random.seed()` という関数を呼び出せば乱数の種（数列の初期値，パラメータ）が固定される。本書では，`np.random.seed(1000)` を実行した後に，続く乱数生成のコードが順次実行されている。

```
       [[0.00471886, 0.12156912],
        [0.67074908, 0.82585276]],

       [[0.13670659, 0.57509333],
        [0.89132195, 0.20920212]]])
```

標準正規分布に従う乱数を生成する方法はいくつかある。上で紹介した関数は，入力としてタプルを渡す使い方をしているので，これと同じ使用法である `np.random.standard_normal()` を紹介しよう。

```
np.random.standard_normal((3, 2))
```

```
array([[-0.45802699,  0.43516349],
       [-0.58359505,  0.81684707],
       [ 0.67272081, -0.10441114]])
```

特別な行列の生成　特別な行列を生成するときも，いくつかの便利な関数がある。対角成分に 1 が並ぶ単位行列を作るときは `eye()` を使う。

```
np.eye(3)
```

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

対角成分だけを指定して対角行列を作るには `diag()` を使う。

```
np.diag([1, -1, 3.])
```

```
array([[ 1.,  0.,  0.],
       [ 0., -1.,  0.],
       [ 0.,  0.,  3.]])
```

連続する数ベクトルの生成　`linspace()` は連続的な区間を出力サイズを指定して等間隔に離散化する際に用いる。非常によく使うので覚えておこう。入力に与えた数字は，左端・右端・出力サイズの順に並んでいる。

```
np.linspace(0, 1, 5)
```

```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

出力サイズではなく，隣り合う数の間の距離を指定したい場合には，arange() を用いる。右端の点が含まれないことには十分注意しよう。

```
np.arange(5.0, 7.0, 0.2)
```

```
array([5. , 5.2, 5.4, 5.6, 5.8, 6. , 6.2, 6.4, 6.6, 6.8])
```

```
np.arange(6)
```

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

配列の連結　2 つ以上の配列を連結して 1 つの配列を生成する方法を紹介しよう。

- np.c_[] は列（column）の方向に配列を拡張する。
- np.r_[] は行（row）方向に配列を拡張する。

丸括弧ではなく各括弧で呼び出すことに注意しよう [11]。

```
u = np.array([1, 2, 3, 4]).reshape((2, 2))
u
```

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

```
v = np.array([10, 11])
v
```

```
array([10, 11])
```

```
np.c_[u, v]
```

```
array([[ 1,  2, 10],
       [ 3,  4, 11]])
```

```
w = v.reshape((1, 2))
np.r_[u, w]
```

```
array([[ 1,  2],
       [ 3,  4],
```

[11] 関数でなければ気持ち悪いという場合は，hstack(), vstack(), concatenate() の使い方を調べてみよう。

```
       [10, 11]])
```

シェイプの変更

配列の変形を自由自在に行えると，後々都合がよいので，ここで説明しておこう。次のベクトル（1 次元配列） a を 2×4 行列（2 次元配列）に変換したいとする。

```
a = np.arange(8)
a
```

```
array([0, 1, 2, 3, 4, 5, 6, 7])
```

2 つの方法がある。

- reshape() メソッドを使って変換後のオブジェクトを生成する。この場合，元のオブジェクトは変更されない。
- resize() メソッドを使って変換前のオブジェクトを書き換える。このように，元のオブジェクトが変更される操作を「in-place」な操作とか，破壊的な操作と呼ぶ。

```
a.reshape((2, 4))
```

```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

```
a
```

```
array([0, 1, 2, 3, 4, 5, 6, 7])
```

```
a.resize((2, 4))
a
```

```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

行列に変換されるときには行方向に数字が埋められていくことに注意しよう [12]。転置行列を作るには，transpose() あるいは T 属性を用いる。

[12] Python と同じくデータ分析でよく用いられる R 言語は列方向に数字を埋めていく。すでに R に慣れている人にとってはしばらく混乱するかもしれない。3 次元配列の作り方にも微妙が違いがあるので，色々と比較をしてみてほしい。

```
a.transpose()
```

```
array([[0, 4],
       [1, 5],
       [2, 6],
       [3, 7]])
```

```
a.T
```

```
array([[0, 4],
       [1, 5],
       [2, 6],
       [3, 7]])
```

サイズが 1 の次元を除去するメソッド squeeze() は多用される（私が多用する）ので，覚えておいてほしい。以下の例は，3 つのベクトル時系列

$$b_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad b_2 = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \quad b_3 = \begin{bmatrix} 4 \\ 5 \end{bmatrix}$$

をシェイプが $(3, 2, 1)$ であるような 3 次元配列として表現した b と，それを表形式で表現し直すために最後の次元を落としたものである。ベクトル時系列は 3 次元配列として表現する方がシミュレーション・コードを簡潔に保ちやすい。しかし，2 次元配列（行列）として表現しておく方が可視化や統計処理には便利である。一長一短があるので，どちらの表現も使えるようにしておこう [13]。

```
b = np.arange(6).reshape((3, 2, 1))
b
```

```
array([[[0],
        [1]],

       [[2],
        [3]],

       [[4],
        [5]]])
```

[13] 表形式から 3 次元配列にするには，np.expand_dims(a, 2) のようにすればよい。

```
b.squeeze()
```

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

2.4.3. 可視化

ここまで分かれば，簡単な可視化を実行することができる。ちょっと脱線して，作図の方法を紹介しよう。

可視化にも様々な方法があるが，もっともよく用いられるのは **Matplotlib** というライブラリだろう。次のインポート文はイディオム的に覚えてしまおう。

```
import matplotlib.pyplot as plt
```

関数（$y = f(x)$）を描くには，次の 3 ステップが基本である。

1. 関数値を計算したい横軸の値を決める。
2. 指定した横軸すべてについて関数値を計算する。
3. プロット関数を呼び出す。

Matplotlib のもっとも初歩的な使い方を用いれば，次のようになる（図 2.1）。なお，井桁記号（#）に続くテキストは人間のためのコメントであり，Python は無視する。ここでのコメントは私が読者のために書いたものなので，読者は入力する必要がない。

```
x = np.linspace(0, 5, 200)   # Step 1
y = np.exp(x)                # Step 2
plt.plot(x, y)               # Step 3
plt.show()
```

最後のコマンド `plt.show()` は描画したグラフを画面に表示するためのものである [14]。

[14] IPython で `%matplotlib` マジックコマンドを実行している場合には `plt.show()` は不要になる。Jupyter Notebook を用いている場合も，`plt.show()` を忘れても図が表示されるかもしれない。表示されない場合は，`%matplotlib inline` というマジックコマンドを実行して，作図のコードを再実行してみよう。

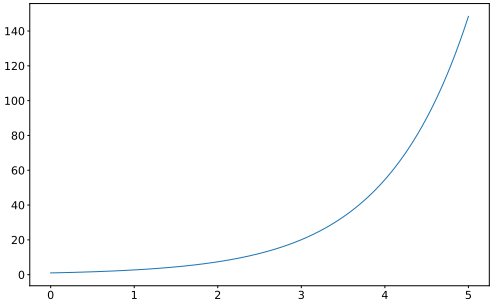


図 2.1.: **Maplotlib** の使用例

ランダム・ウォーク [15]

$$x_{t+1} = x_t + \varepsilon_{t+1}, \qquad \varepsilon_{t+1} \overset{\text{i.i.d}}{\sim} \mathcal{N}(0,1), \quad t = 0, 1, \ldots$$
$$x_0 = 0$$

をシミュレーションするには次のコードを実行するとよい（図 2.2）。1 つだけだと簡単過ぎて拍子抜けしてしまうかもしれないので，2 つの経路を描いておこう。

```
# Solid line
eps1 = np.r_[0, np.random.standard_normal(100)]
x1 = np.cumsum(eps1)
# Dashed line
eps2 = np.r_[0, np.random.standard_normal(100)]
x2 = np.cumsum(eps2)

plt.plot(x1, 'k-')
plt.plot(x2, 'k--')
plt.show()
```

`cumsum()` の使い方も分かっていただけただろうか。

[15] $\varepsilon_{t+1} \overset{\text{i.i.d}}{\sim} \mathcal{N}(0,1)$ は，ランダムな撹乱項 $\varepsilon_1, \varepsilon_2, \ldots$ が

- 互いに独立で同じ確率分布に従っている（i.i.d = independent and identically distributed）
- 確率分布は標準正規分布 $\mathcal{N}(0,1)$ である

という 2 つの性質を表現している。なお,「ランダム・ウォーク」と言った場合には普通は $\mathcal{N}(0,1)$ に限定する必要はないので，ここでは通常の定義よりも強い仮定を置いている。

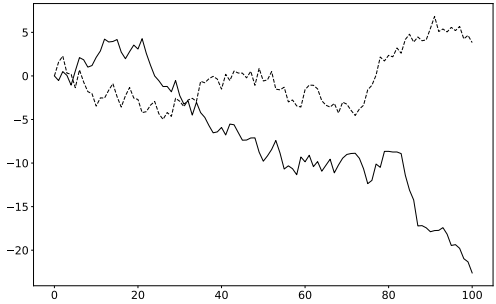


図 2.2.: ランダム・ウォークのシミュレーション

2.5. プログラミング: 価格指数の計算

乱暴な言い方をしてしまえば，価格指数の計算は

- ベクトルの内積
- 2 つの数の割算

の 2 つの計算をしているだけである。**NumPy** の内積を計算する方法は分かったので（問題 2.6），指数計算の準備は整っている。

公務員試験などでも出題される問題に次のようなものがある。

問題 (市役所・平成 21 年度)**.** 下の表はある国における A 財と B 財の価格と数量について，2000 年と 2001 年とで比較したものである。（以下略）

	A 財		B 財	
	価格	数量	価格	数量
2000 年	40	3	80	6
2001 年	80	5	30	7

次のようなコードを書けばラスパイレス指数，パーシェ指数（の 100 を掛ける前の数字）が得られる。式 (2.1), (2.2) とコードをよく見比べてほしい。

```
price00 = np.array([40, 80])
price01 = np.array([80, 30])
quantity00 = np.array([3, 6])
quantity01 = np.array([5, 7])
```

5

```
# Laspeyres
sum(price01* quantity00) / sum(price00 * quantity00)
```

```
0.7
```

```
# Paasche
sum(price01 * quantity01) / sum(price00 * quantity01)
```

```
0.8026315789473685
```

データ保存形式を工夫する

上のようなコードを書くときに，頭の中で元の表を次の形式に置き換えるという操作を実行している。太字にした部分が quantity00 に対応する。

価格	A 財	B 財
2000 年	40	80
2001 年	80	30

数量	A 財	B 財
2000 年	**3**	**6**
2001 年	5	7

上のコードを書いたときの気持ちは，各表の行（年）ごとに変数を作れば内積計算ができるだろうというものだった。非常に愚直に考えてコードを書いた訳だが，これではデータの年数が増えたときに大変困ったことになる。10 年分のデータがあると変数が 20 個になる。これでは管理が大変だ。そこで，この 2 つの表をそのままコードに変換してみよう。表形式のデータを行列と見なして，次のように書ける。

```
price = np.array([[40, 80],
                  [80, 30]])
price
```

```
array([[40, 80],
       [80, 30]])
```

```
quantity = np.array([[3, 6],
                     [5, 7]])
quantity
```

```
array([[3, 6],
       [5, 7]])
```

NumPy の 2 次元配列から一部分を取り出すには，

配列名 [行インデックス，列インデックス]

の形式を用いる[16]。特定の次元からすべての要素を抜き出すためにはコロン（:）をインデックスに指定する。したがって，quantity00 に相当する部分を配列 quantity から抜き出すためには,「0 行目のすべての列」を指定すればよい。

```
quantity[0, :]
```

```
array([3, 6])
```

このようにデータを保存した場合には，ラスパイレス指数の計算は次のようになる。

```
sum(price[1, :] * quantity[0, :]) / sum(price[0, :] * quantity[0, :])
```

```
0.7
```

さらに言えば，比較年を表す 1 という数字は新しいデータが来れば変わるので，これは変数に保存しておくだけの価値がある。次のコードと式 (2.2) がかなり近い表現になっていることを確認してほしい。

```
t = 1
sum(price[t, :] * quantity[t, :]) / sum(price[0, :] * quantity[t, :])
```

```
0.8026315789473685
```

データの保存形式を工夫することで，次のようなご利益がある。

- 数式とコードの対応関係をわかりやすくできるので，他の人に説明しやすくなる。
- 結果的にコードの保守が容易になる。

1ヶ月後の自分の理解力を決して過信してはいけない。読みやすいコードを書くように心がけよう。

連鎖指数の計算

連鎖指数が意味を持つには 3 年以上の期間が必要なので，さきほどのデータを次のように拡張する。

[16] 多次元配列の場合は，第 1 次元のインデックス，第 2 次元のインデックス，第 3 次元のインデックス … とコンマでつないでいけばよい。なお，このような記法は **ndarray** でないただのリストのリストには使えないことに注意しよう。

価格	A 財	B 財
2000 年	40	80
2001 年	80	30
2002 年	70	40
2003 年	60	55

数量	A 財	B 財
2000 年	3	6
2001 年	5	7
2002 年	6	8
2003 年	8	10

ラスパイレス式の連鎖指数（参照年 = 2000 年）を計算するには次のようにすればよい。

```
price = np.array([[40, 80],
                  [80, 30],
                  [70, 40],
                  [60, 55]])
quantity = np.array([[3, 6],
                     [5, 7],
                     [6, 8],
                     [8, 10]])

laspeyres = np.empty(price.shape[0])   # 価格指数の保存場所を確保

laspeyres[0] = 1.
t = 1
laspeyres[t] = laspeyres[t-1] * (sum(price[t, :] * quantity[t-1, :])
                                 / sum(price[t-1, :] * quantity[t-1, :]))
t = 2
laspeyres[t] = laspeyres[t-1] * (sum(price[t, :] * quantity[t-1, :])
                                 / sum(price[t-1, :] * quantity[t-1, :]))
t = 3
laspeyres[t] = laspeyres[t-1] * (sum(price[t, :] * quantity[t-1, :])
                                 / sum(price[t-1, :] * quantity[t-1, :]))
laspeyres
```

```
array([1.        , 0.7       , 0.72295082, 0.78156845])
```

上のコードには

```
laspeyres[t] = laspeyres[t-1] * (sum(price[t, :] * quantity[t-1, :])
                                 / sum(price[t-1, :] * quantity[t-1, :]))
```

という同じコードが繰り返されていることに注目しよう。このような冗長なコードは本来は避けるべきだが，連鎖指数が同じ計算の繰り返しで計算されている感じを

十分に理解していただきたい。さらに，1 つ前のステップの計算結果を使って次の計算が実行されるような構造になっている点も重要である。これは,「漸化式」とか「再帰方程式」と呼ばれるものの一種である。動学モデルでは基本的なパターンなので，何度も練習して自力で書けるようになってほしい。おおよそ次のようなステップに分解される。

1. 結果を保存する配列を用意する。
2. 初期値を設定する。
3. 再帰方程式に基づいて順番に値を埋めていく。

繰り返し処理のための `for` ループは次章で学ぶ。

問題 2.8. 上の表に対して，パーシェ式の連鎖指数とフィッシャーの連鎖指数を計算しなさい。（ヒント: 平方根は `x ** 0.5` または `np.sqrt(x)` で計算できる。）

3. GDP の成長と格差

3.1. 概要

あなたは次のような記述をどのように理解するだろうか。

> 「GDP が伸び悩む中で，X 産業の付加価値生産額は年率 10% で成長している。国際競争力を高めつつある X 産業に重点的に補助金を配分することで，GDP 低迷を脱却することができるだろう」

日本経済の救世主とみなすべきか，あるいは業界団体による陳腐なポジショントークと見るか[1]。おそらく唯一の正しい理解は,「情報が不足しているので判断できない」というものだ。この章で学ぶのはそういう話だ。

マクロ経済学は一国や地域の経済を総体として扱う経済分析である。分析対象の性質上，分析の中心的なターゲットは「合計」や「平均」といった集計量になる。マクロ経済学が分析対象とする集計量の中で，現在のところもっとも重要な概念は GDP であろう。GDP（国内総生産）は企業や政府によって生産された財・サービスを重複なく合算したものである。GDP は経済全体の生産能力を表すとともに居住者の総所得を表しており，居住者の平均的な厚生（暮らし向き）と強い相関を持っている[2]。

GDP は生産された価値の合計であるから，合計する前の GDP を構成する細部のデータ，例えば産業ごとに計算した付加価値生産額など，も利用することができるということに注意しておこう。年率 10% で成長中の前述の X 産業は，GDP に対するシェアが 0.2% であるとしよう。GDP を 500 兆円とすれば，この産業は付加価値生産額 1 兆円の小さな産業である。補助金の投入によって産業の成長率を 10% → 11% に加速することができたとして，GDP 全体の成長に対する貢献がどの程度になるかを計算してみよう。補助金を投入しなければ 1.1 兆円を次年度に生産

[1]「ポジショントーク」という言葉は否定的に響いてしまうかもしれないが，批判する意図はまったくない。嘘をついてはいけないが，自らのポジションが有利になるような事実を強調することは誰もが日常的に行っている。結局は，情報の受け手のリテラシーが重要だ。

[2]生産と所得の関係については，ひとまずのところは，生産された財・サービスを販売することで得た売上から給料が支払われるから，と考えておくとよい。

していたところ，補助金の投入によって 1.11 兆円に生産額が増える。増分は 0.01 兆円であるから，現在の GDP 全体に占める増分の比率は

$$0.01 \text{ 兆円 } / 500 \text{ 兆円 } = 0.002\%$$

となる。この数字を大きいと見るか，小さいと見るかは意見の分かれるところだろう。しかし，短期的に（例えば，1 年とか 2 年で）GDP を拡大することに大きく貢献してくれるかといえば，答えはノーである。この産業は有望な成長産業であるかもしれないが，今はまだ小さすぎる。

とは言え，もし私が X 産業に従事していて，政府の補助金を獲得するための資料を作るように命じられたら，産業の成長力を強調する前述のようなレポートを書くだろう。ただし，それだけでは経済産業省の審査官や諮問委員は納得させられないだろうとも思う。短期的な GDP への貢献だけを見れば，全体から見て小さすぎるからだ。長期的なスパンで見たときに GDP の構成をどのように塗り替えていくかとか，あるいは関連産業に視野を広げて当該産業の影響の大きさをアピールするといった踏み込んだ分析をしなければ，魅力的な申請書にはならない。

私たちの多くは特定の産業を代弁する立場にもないし，申請書を審査する立場にもないが，このような話を理解しておく必要がある。政府の資源は有限であるし，財源は税金である。明らかに有効でない政策を実施する政府に対しては，投票行動やデモなどを通じて反対意見を表明しなければならない。しかし，そのためにはまず数字を読めるようになることが大切だ。

理論パートの後半では，近年注目を集めている格差の指標について簡単に紹介する。GDP の成長は国民の平均的な暮らし向きを向上させるが，増えた所得がすべての国民に均等に行き渡る訳ではない。全体の拡大とともに個々の暮らし向きの変化を捉える指標を押さえておこう。

この講義では以下のことを学ぶ。

- 理論
  - 名目 GDP と実質 GDP，GDP デフレータ
  - 寄与度と寄与率
  - 格差の指標: 相対的貧困率，ジニ係数，トップ 1% の所得シェア
- プログラミング
  - 繰り返し計算（`for` ループ，`while` ループ）
  - 条件分岐（`if-elif-else`）

3.2. 理論: GDP

3.2.1. 名目 GDP

GDP（Gross Domestic Product, 国内総生産）は，ある期間において（通常は，1年または四半期），各生産者が生産した付加価値をすべて合計したものである。付加価値とは，

付加価値 = 生産物の価値 – 投入中間財の価値

によって定義される。中間財というのは他の企業から購入した財・サービスであって，固定資本（資産）に分類されないものである。原料や材料のことと考えておけばよい。

生産された付加価値がどのように使われたかによって，

- 消費（Consumption）= 民間消費
- 投資（Investment）= 民間投資
- 政府購入（Government Purchases）
- 純輸出（Net Export）

に分類する。これはマクロ経済学や理論分析上の慣例であって，国民経済計算では通常，

- 消費 = 民間消費 + 政府消費
- 投資 = 民間投資 + 政府投資
- 純輸出

のように切り分けている。政府消費と政府投資を合算したものが政府購入である。数字を見るときには定義に注意しよう。

消費というのは，各期の経済厚生を表す概念である。家計部門の消費（民間消費）がその経済の構成員の厚生をもっともよく表すと考える場合には政府消費を切り離して民間消費を消費とみなす。政府から受けたサービス（役所が提供する無料または非常に低廉なサービス）も国民が享受していると考えれば，政府消費を含めて消費と扱う方が望ましい場合もある [3]。GDP 上は民間現実最終消費として記録されている。

投資というのは，生産設備を拡大する経済活動である。民間企業の生産設備を拡充したり，家計部門が持ち家（住宅サービスを供給する）を新築したりすることが

[3]例えば，とあるサービスに対して政府が 7 割の補助金を出したとしよう。これは，

含まれる。政府部門の投資には，道路や治水工事，庁舎の整備などが含まれる。いずれも，長期に渡って価値を生産する財・サービスの購入である。経済の生産キャパシティの拡大という意味では，政府投資を投資に含める方が理にかなっている。一方，民間部門の投資は経済の不沈を示す重要な指標であり，民間投資の変化は独立した重要性を持っている。

純輸出は

$$\text{純輸出} = \text{輸出（Export）} - \text{輸入（Import）}$$

によって定義される。自国で生産した財やサービスを輸出した場合には，輸出額は GDP に含まれている。生産活動に輸入した価値が投入されている場合は，これを控除しなければならない。自国で生産された価値ではないものが含まれないようにするためだ。

マクロ経済学の標準的な教科書では,「生産 = 民間消費 + 民間投資 + 政府購入 + 純輸出」という分解が用いられている。これは政府部門が民間部門と異なる役割を担うからである。不況時には民間消費・民間投資は減少するが，政府は政府購入を拡大することで景気の大幅な後退を避けようとする。逆に好況で民間消費・民間投資が拡大する際には，政府購入を維持するか減少させることで，不況時にできた負債の解消を目指す。民間部門の経済活動と政府部門の経済活動は，通常，反対方向に動くのでこれらを単純に合算してしまうと分析上都合が悪い。

Y を GDP，C を（民間）消費，I を（民間）投資，G を政府購入，NX を純輸出とすると，次の関係式が成り立っている。

$$Y = C + I + G + NX$$

輸出を EX，輸入を IM と書くと，$NX = EX - IM$ であり，

$$Y = C + I + G + EX - IM$$

が成り立つ。これらは教科書的なマクロ経済分析においてもっともよく使われる GDP の加法分解であり，もっとも荒い分解である。

- 消費 C は一国の経済活動において通常もっとも大きな項目である。消費は個人の厚生（幸福度）を決定する重要な要因であるので，マクロ経済理論においても最重要の項目である。
- 投資 I（設備投資，在庫投資，住宅投資）は一国経済における生産のキャパシティを左右する項目である。

- 政府購入 G は政府の規模を表す項目である。政府購入の調整によって短期的な経済変動をコントロールすることができると考えられており，学部標準レベルのマクロ経済学では重要な項目になる。
- 純輸出 NX は外国経済との関わりを表す部分である。輸出 EX から輸入 IM を控除したものである。

経済の生産規模を測る GDP ではあるが，米 800 万トンと自動車 70 万台を単純に足すことはできないので，貨幣の単位で測った価値を用いて生産規模を計算する。

$$\text{価値} = \text{価格} \times \text{数量}$$

なので，これをすべての最終財（国内生産物であって，民間・政府に消費または投資されるか，外国に輸出される財のこと）について合計し，輸入品の価値を控除したものが GDP になっている。価格として当期の価格を用いる GDP を**名目 GDP**（nominal GDP）という。

3.2.2. 実質 GDP

名目 GDP の計算の基礎となる価値の式に，時間のインデックスを付記しておこう。

$$\begin{aligned} \text{価値}_t &= \text{価格}_t \times \text{数量}_t \\ \text{価値}_{t+1} &= \text{価格}_{t+1} \times \text{数量}_{t+1} \end{aligned}$$

特定の財について，生産物の価値の上昇は 2 つの原因によって起こる。つまり，価格が上昇するか，生産数量が拡大するかだ。

$$\begin{aligned} (\text{価値の粗変化率}) &= \frac{\text{価値}_{t+1}}{\text{価値}_t} = \frac{\text{価格}_{t+1}}{\text{価格}_t} \times \frac{\text{数量}_{t+1}}{\text{数量}_t} \\ &= (\text{価格の粗変化率}) \times (\text{数量の粗変化率}) \end{aligned}$$

私たちの豊かさ（より正確には，物質的な豊かさ material wealth）は，財やサービスをどれだけ消費・利用できるかによって決まる。その財をいくらで買ったかとは関係ないはずである [4]。

[4] 個別の財や特別な消費行動を説明する文脈では，価格が上がることで消費の効用が高まるような財（ヴェブレン財）が存在することは否定しない。しかし，そのような財は多くはないだろうし，経済全体で見れば小さな市場規模だろう。マクロ経済学はあくまでも総体的・平均的な行動を分析することが目標なので，ひとまず素朴な消費行動のみを考えておけばよい。

そこで，豊かさを測る指標として，成長率

$$\frac{\text{数量}_{t+1}}{\text{数量}_t} = \frac{\text{価値}_{t+1}/\ \text{価値}_t}{\text{価格}_{t+1}/\ \text{価格}_t}$$

を用いようというのが実質 GDP の考え方である。この数値が 1 より大きければ経済活動は拡大していて，1 より小さければ縮小している。1 であれば現状維持となる。多種多様な財・サービスが生産される経済において，数量変化率を計算するためには 第 2 章で扱ったような工夫が必要になる。

各最終財（$i = 1, \ldots, N$）について，t 期における生産量を並べたベクトルを $\boldsymbol{x}_t$，価格ベクトルを $\boldsymbol{p}_t$ とする。$\boldsymbol{x}_t$ の第 i 要素 $x_{t,i}$ は財 i の生産量，$\boldsymbol{p}_t$ の対応する要素 $p_{t,i}$ が財 i の当該期の平均価格を表している。t 期の名目 GDP, Y_t, は

$$Y_t = \text{名目 GDP}_t = \boldsymbol{p}_t \cdot \boldsymbol{x}_t$$

と書ける。$t-1$ 期から t 期にかけての経済成長率 g_t は

$$\text{（粗）経済成長率}_t = 1 + g_t = \frac{\boldsymbol{p}_{t-1} \cdot \boldsymbol{x}_t}{\boldsymbol{p}_{t-1} \cdot \boldsymbol{x}_{t-1}}$$

と定義する。ベクトルを数値化するベクトルとして過去の情報 $\boldsymbol{p}_{t-1}$ を用いているので，ラスパイレス指数（あるいは**ラスパイレス数量指数**）である。

参照年を 0 期とする。0 期においては

$$\text{実質 GDP}_0 = \text{名目 GDP}_0 \qquad \text{あるいは} \qquad y_0 = Y_0$$

が成り立つとする。比較年 t 期の**実質 GDP**（連鎖方式）は次のように定義される。

$$\begin{aligned} y_t &= \prod_{i=1}^{t} (1 + g_i) \times y_0 \\ &= (1 + g_t)(1 + g_{t-1}) \times \cdots \times (1 + g_1) \times y_0 \\ &= \frac{\boldsymbol{p}_{t-1} \cdot \boldsymbol{x}_t}{\boldsymbol{p}_{t-1} \cdot \boldsymbol{x}_{t-1}} \times \frac{\boldsymbol{p}_{t-2} \cdot \boldsymbol{x}_{t-1}}{\boldsymbol{p}_{t-2} \cdot \boldsymbol{x}_{t-2}} \times \cdots \times \frac{\boldsymbol{p}_0 \cdot \boldsymbol{x}_1}{\boldsymbol{p}_0 \cdot \boldsymbol{x}_0} \times y_0 \end{aligned} \tag{3.1}$$

すなわち，連続する 2 期の経済成長率をラスパイレス数量指数で計算し，参照年から累積的に積算したものが実質 GDP である。

式 (3.1) において，0 期の実質 GDP が $y_0 = \boldsymbol{p}_0 \cdot \boldsymbol{x}_0$ であることに注意すれば，異

なる表現が得られる。

$$\begin{aligned} y_t &= \frac{\boldsymbol{p}_{t-1}\cdot\boldsymbol{x}_t}{\boldsymbol{p}_{t-1}\cdot\boldsymbol{x}_{t-1}} \times \frac{\boldsymbol{p}_{t-2}\cdot\boldsymbol{x}_{t-1}}{\boldsymbol{p}_{t-2}\cdot\boldsymbol{x}_{t-2}} \times \cdots \times \frac{\boldsymbol{p}_0\cdot\boldsymbol{x}_1}{\boldsymbol{p}_0\cdot\boldsymbol{x}_0} \times \boldsymbol{p}_0\cdot\boldsymbol{x}_0 \\ &= \boldsymbol{p}_t\cdot\boldsymbol{x}_t \times \frac{\boldsymbol{p}_{t-1}\cdot\boldsymbol{x}_t}{\boldsymbol{p}_t\cdot\boldsymbol{x}_t} \times \frac{\boldsymbol{p}_{t-2}\cdot\boldsymbol{x}_{t-1}}{\boldsymbol{p}_{t-1}\cdot\boldsymbol{x}_{t-1}} \times \cdots \times \frac{\boldsymbol{p}_0\cdot\boldsymbol{x}_1}{\boldsymbol{p}_1\cdot\boldsymbol{x}_1} \end{aligned}$$

右辺の最初の項に余分な項 $\boldsymbol{p}_t\cdot\boldsymbol{x}_t/\boldsymbol{p}_t\cdot\boldsymbol{x}_t$ を追加して，$\boldsymbol{p}_0\cdot\boldsymbol{x}_0/\boldsymbol{p}_0\cdot\boldsymbol{x}_0$ を消去した。分母を 1 つずつずらしていることにも注意しよう。これは，さらに

$$y_t = \frac{\boldsymbol{p}_t\cdot\boldsymbol{x}_t}{\frac{\boldsymbol{p}_t\cdot\boldsymbol{x}_t}{\boldsymbol{p}_{t-1}\cdot\boldsymbol{x}_t} \times \frac{\boldsymbol{p}_{t-1}\cdot\boldsymbol{x}_{t-1}}{\boldsymbol{p}_{t-2}\cdot\boldsymbol{x}_{t-1}} \times \cdots \times \frac{\boldsymbol{p}_1\cdot\boldsymbol{x}_1}{\boldsymbol{p}_0\cdot\boldsymbol{x}_1}}$$

と書き直すことができる。分母がパーシェ式連鎖指数となっていることに注意する。すなわち，

$$\begin{aligned} \mathrm{DFL}_0 &= 1 \\ \mathrm{DFL}_i &= \frac{\boldsymbol{p}_i\cdot\boldsymbol{x}_i}{\boldsymbol{p}_{i-1}\cdot\boldsymbol{x}_i} \times \mathrm{DFL}_{i-1}, \quad i = 1, 2, \ldots, t \end{aligned} \tag{3.2}$$

と定義すれば，

$$y_t = \frac{\boldsymbol{p}_t\cdot\boldsymbol{x}_t}{\mathrm{DFL}_t}$$

とできる。もちろん，DFL_t は当期の価格指数として用いることができる。このように定義される価格指数 DFL が連鎖方式の **GDP デフレータ**である。次のおなじみの公式が成り立っている [5]。

$$\text{実質 GDP} = \frac{\text{名目 GDP}}{\text{GDP デフレータ}}$$

問題 3.1. 名目 GDP の変化率が 2%，実質 GDP の変化率（経済成長率）が 1.5% であったとする。このとき，GDP デフレータで測った物価変化率（インフレ率）はいくらか。計算しなさい。

[5] 理論上は，(3.2) のようにデフレーターを計算してから実質 GDP を計算しても，(3.1) で実質 GDP を計算してからデフレーターを計算することもできる。実務上は，実質 GDP を先に計算しているようだ。

$$\text{デフレーター} = \frac{\text{名目値}}{\text{実質値}}$$

のように計測されるデフレーターを「インプリシット・デフレーター」という。

3.2.3. 寄与度と寄与率

政府発表やそれに基づいた新聞記事では，経済成長が加速したり減速したりする現象の要因を説明するための記述として,「個人消費が順調だった」とか「設備投資が伸び悩んだ」といった記述をよく見かける。この手の分析では**寄与度**（contribution）や寄与率という概念が用いられている。

成長率 t 期の GDP を Y_t，その次の期（$t+1$ 期）の GDP を Y_{t+1} とする [6]。経済成長率は GDP の変化率

$$\frac{\Delta Y_{t+1}}{Y_t} = \frac{Y_{t+1} - Y_t}{Y_t} = \frac{Y_{t+1}}{Y_t} - 1$$

によって定義される。Y_t, Y_{t+1} の分解を次のように書こう。

$$Y_t = C_t + I_t + G_t + NX_t$$
$$Y_{t+1} = C_{t+1} + I_{t+1} + G_{t+1} + NX_{t+1}$$

GDP が成長するためには C, I, G, NX のうち少なくとも 1 つは 大きくなっているはずだ。GDP 成長率を項目ごとに分解するのが寄与度である。

寄与度

項目ごとの寄与に分解するためには，GDP 成長率の定義に加法分解の公式を代入すればよい。

[6] ここでは慣例に従って大文字を利用しているが，実質変数（名目変数をデフレータで除した変数）であると考えてよい。

$$\begin{aligned}\frac{\Delta Y_{t+1}}{Y_t} &= \frac{Y_{t+1}-Y_t}{Y_t}\\ &= \frac{(C_{t+1}+I_{t+1}+G_{t+1}+NX_{t+1})-(C_t+I_t+G_t+NX_t)}{Y_t}\\ &= \frac{(C_{t+1}-C_t)+(I_{t+1}-I_t)+(G_{t+1}-G_t)+(NX_{t+1}-NX_t)}{Y_t}\\ &= \frac{\Delta C_{t+1}+\Delta I_{t+1}+\Delta G_{t+1}+\Delta NX_{t+1}}{Y_t}\\ &= \frac{\Delta C_{t+1}}{Y_t}+\frac{\Delta I_{t+1}}{Y_t}+\frac{\Delta G_{t+1}}{Y_t}+\frac{\Delta NX_{t+1}}{Y_t}\end{aligned}$$

ここで，例えば $\Delta I_{t+1}/Y_t$ が，GDP 成長における投資の寄与度である。通常，100 倍して % 表記する。

GDP に限らず加法的に分解できる指標であれば同様の公式で寄与度を計算出来る。例えば，

$$X_t = X_{t,1} + X_{t,2} + \cdots + X_{t,N-1} + X_{t,N} = \sum_{n=1}^{N} X_{t,n}$$

とすれば，

$$\frac{\Delta X_{t+1}}{X_t} = \sum_{n=1}^{N} \frac{\Delta X_{t+1,n}}{X_t} \Longrightarrow \text{項目 } n \text{ の寄与度} = \frac{\Delta X_{t+1,n}}{X_t}$$

問題 3.2. $X_t, t = 2000, 2001,$ の加法分解 $X_t = X_{t,1} + X_{t,2} + X_{t,3}$ が下の表で与えられている。2000 年から 2001 年にかける X の成長について，項目 1, 2, 3 の寄与度を計算しなさい。

	$X_{t,1}$	$X_{t,2}$	$X_{t,2}$
2000 年	100	300	150
2001 年	120	330	140

加法分解に控除項目が入る場合には，控除項目を負数として扱えばよい。

$$\begin{aligned} Y_t &= C_t + I_t + G_t + EX_t - IM_t \\ &= C_t + I_t + G_t + EX_t + (-IM_t) \end{aligned}$$

$$\frac{\Delta Y_{t+1}}{Y_t} = \frac{\Delta C_{t+1}}{Y_t} + \frac{\Delta I_{t+1}}{Y_t} + \frac{\Delta G_{t+1}}{Y_t} + \frac{\Delta EX_{t+1}}{Y_t} + \frac{\Delta(-IM_{t+1})}{Y_t}.$$

問題 3.3. 2017 年度と 2018 年度の実質 GDP（支出側）の構成は下表の通りであった。単位は兆円。このとき，経済成長率と，各項目の寄与度を計算しなさい。

	C	I	G	EX	IM
2017 年度	298.88	101.56	132.33	91.43	92.62
2018 年度	299.05	102.28	139.39	92.87	94.62

寄与率

GDP 成長率を 1（100%）に基準化したときの寄与度の大きさが寄与率である。例えば，GDP の支出面の分解であれば，寄与度の計算式

$$\frac{\Delta Y_{t+1}}{Y_t} = \frac{\Delta C_{t+1}}{Y_t} + \frac{\Delta I_{t+1}}{Y_t} + \frac{\Delta G_{t+1}}{Y_t} + \frac{\Delta NX_{t+1}}{Y_t}$$

の両辺を全体の成長率 $\Delta Y_{t+1}/Y_t$ で割ってやればよい。

$$
\begin{aligned}
1 &= \frac{(\Delta C_{t+1}/Y_t) + (\Delta I_{t+1}/Y_t) + (\Delta G_{t+1}/Y_t) + (\Delta NX_{t+1}/Y_t)}{(\Delta Y_{t+1}/Y_t)} \\
&= \frac{\Delta C_{t+1}}{\Delta Y_{t+1}} + \frac{\Delta I_{t+1}}{\Delta Y_{t+1}} + \frac{\Delta G_{t+1}}{\Delta Y_{t+1}} + \frac{\Delta NX_{t+1}}{\Delta Y_{t+1}}
\end{aligned}
$$

問題 3.4. 問題 3.2, 3.3 について，各項目の寄与率を計算しなさい。

3.3. 理論: 格差の指標

前節までは GDP の拡大と，その評価方法について説明してきた。「GDP の拡大は善である」という立場を暗黙のうちに取っていた。GDP は一国経済の居住者の平均的な暮らし向きを表す重要な指標であるので，多くの経済分析ではこの前提を置いている。しかし近年では，環境への影響や所得不平等の広がりを背景として，GDP の拡大だけを目標とする政策に対する批判が強くなってきている。ここでは，所得の不平等や格差を測る指標について，その一端を紹介する。

3.3.1. 順序統計量

概念の理解に必要な順序統計量について説明する。

順序に意味のない n 個の数値データ

$$x_1, x_2, x_3, \ldots, x_n$$

を小さい順に並べ替える。新しい記号を導入したくないので，下付き添字に括弧を

付けて順位を表すものとする。すなわち,

$$\begin{aligned} x_{(1)} &= 1 \text{ 番目に小さいデータ} \\ x_{(2)} &= 2 \text{ 番目に小さいデータ} \\ &\vdots \\ x_{(n)} &= n \text{ 番目に小さいデータ} \end{aligned}$$

x_i と書いたときの i はデータに付けた単なる ID で特に意味のないものである。$x_{(j)}$ と書いたときの j はデータを小さい順に並べて j 番目のデータである。$x_{(j)}$, $j = 1, \ldots, n$, の作り方から，次の不等式が成り立つ。

$$x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(n-1)} \leq x_{(n)}$$

最小値・最大値

データ $\{x_1, \ldots, x_n\}$ に対して，$x_{(1)}$ を**最小値**，$x_{(n)}$ を**最大値**という。

中央値

データ $\{x_1, \ldots, x_n\}$ に対して，**中央値**（**median**） を次のように定義する。

$$\mathrm{med}(\{x_1, \ldots, x_n\}) = \begin{cases} x_{(\frac{n+1}{2})} & n \text{ が奇数のとき} \\ \dfrac{x_{(\frac{n}{2})} + x_{(\frac{n}{2}+1)}}{2} & n \text{ が偶数のとき} \end{cases}$$

中央値は昇順で並べたデータを半々に分ける数のことである。

分位数

$q \in [0, 1]$ に対して [7]，q**-分位数**と呼ばれる統計量 Q_q は次のように定義される。

$$\frac{x_{(i)} \leq Q_q \text{ を満たす } i \text{ の数}}{n} = q$$

中央値は $Q_{1/2}$ である。厳密な定義は難しいので省略する。詳細は 久保川・国友 (2016) を参照。

[7] $a \in [0, 1]$ は $0 \leq a \leq 1$, $a \in (0, 1)$ は $0 < a < 1$ という意味である。丸括弧と角括弧を組み合わせて $a \in [0, 1) \Leftrightarrow 0 \leq a < 1$ などとも書く。

3.3.2. 相対的貧困率

相対的貧困率（relative poverty rate）の計算には可処分所得（所得から税・社会保険料等を引いた額）を用いる。所得を得ていない個人もいるので，世帯人員数も考慮した**等価可処分所得**を以下のように計算する。

$$\text{個人の等価可処分所得} = \frac{\text{属する世帯の可処分所得の総額}}{\sqrt{\text{属する世帯の人員数}}}$$

属する世帯の人員数の平方根を用いるのは，世帯人員数が多いほうが 1 人あたりの生活コストが下がることを考慮したものである。例えば，3 人世帯の家賃は 1 人暮らしの人の家賃の 3 倍にはならない。例えば下表左のような可処分所得の分布であれば，等価可処分所得は下表右のように計算できる。

個人	世帯	可処分所得
1	A	400
2	A	300
3	A	0
4	B	1,000
5	B	0
6	C	250
7	D	150

$\Longrightarrow$

個人	世帯人員数	等価可処分所得
1	3	404.1452
2	3	404.1452
3	3	404.1452
4	2	707.17
5	2	707.17
6	1	250
7	1	150

等価可処分所得の中央値（中位点所得）を半分にした数値を**相対的貧困線**と定義する。上の例には 7 人の個人がいるので，大きい順にならべて 4 番目にあたる所得が中央値である。

$$\text{上の例の中央値} = 404.1452$$

さらにこれを半分にしたものが相対的貧困線である。

$$\text{相対的貧困線 RPL} = \frac{404.1452}{2} = 202.0726$$

相対的貧困とは，この相対的貧困線未満の所得しかない状態である。上の例で，相対的貧困の状態にある個人について True，そうでない個人について False を付したものが下の表である。

個人	世帯人員数	等価可処分所得	相対的貧困
1	3	404.1452 $\not<$ RPL	False
2	3	404.1452 $\not<$ RPL	False
3	3	404.1452 $\not<$ RPL	False
4	2	707.17 $\not<$ RPL	False
5	2	707.17 $\not<$ RPL	False
6	1	250 $\not<$ RPL	False
7	1	150 $<$ RPL	True

True となる個人の割合が相対的貧困率である。

$$\text{相対的貧困率} = \frac{\text{等価可処分所得が中位点所得の半分未満である人の数}}{\text{総人口数}}$$

上の例では，$1/7 \fallingdotseq 14\%$ となる。

問題 3.5. 高所得グループと低所得グループとで二極化している場合には，相対的貧困率が高くなる傾向がある。このことを適当な数値例を構築して確認しなさい。

相対的貧困の状態にあると分類される人たち全員が日々の衣食にも困るような日常的な意味での「貧困」状態にあるとは限らない。しかし，彼らの生活は，多くの世帯が享受できている標準的な生活水準に達していない可能性が高いし，教育を受けることを諦めてしまう可能性も高く，次の世代に貧困を引き継いでしまうおそれがある。したがって，政府は目を配る必要があるし，多くの団体がサポートしようとしている。

相対的貧困の定義上，相対的貧困率は 50% を超えることはない。したがって，国民の多くが「貧困状態」にある経済の分析をする場合には相対的貧困を使うことが

できない。生活必需品を買うことすらままならないような極度の貧困状態を測るためには**絶対的貧困**という指標を用いる。発展途上国の支援の文脈では絶対的貧困が使われる。

問題 3.6. 絶対的貧困の定義を調べて，調べたことを簡潔にまとめなさい。

3.3.3. ジニ係数

所得分布の不平等を測る指標の 1 つとして**ジニ係数**（Gini coefficient）がよく使われている。データ $\{y_1, y_2, \ldots, y_n\}$ を個人の所得（等価可処分所得）であるとしよう。各個人の所得を小さい順に並び替えた

$$y_{(1)}, y_{(2)}, \cdots, y_{(n-1)}, y_{(n)}$$

をもとに次のようなデータを構築する。所得順位が下から i 番目の個人までの所得の合計を

$$\hat{y}_i = y_{(1)} + y_{(2)} + \cdots + y_{(i)}$$

と定義する。便宜上，$\hat{y}_0 = 0$ とする。国民の総所得は

$$\hat{y} = \hat{y}_n = y_{(1)} + y_{(2)} + \cdots + y_{(n)}$$

第 i 番目の個人までの所得 $\hat{y}_i$ が総所得 $\hat{y}$ にしめる割合 $Y_i = \hat{y}_i / \hat{y}$ のベクトル

$$\begin{aligned}
\boldsymbol{Y} &= (Y_0, Y_1, \ldots, Y_{n-1}, Y_n) \\
&= \left(\frac{\hat{y}_0}{\hat{y}}, \frac{\hat{y}_1}{\hat{y}}, \ldots, \frac{\hat{y}_{n-1}}{\hat{y}}, \frac{\hat{y}_n}{\hat{y}} \right) \\
&= \left(0, \frac{y_{(1)}}{y_{(1)} + \cdots + y_{(n)}}, \ldots, \frac{y_{(1)} + \cdots + y_{(n-1)}}{y_{(1)} + \cdots + y_{(n)}}, 1 \right)
\end{aligned}$$

を作る。さらに，i 番目までの個人の全体の総人口 n に占める割合 i/n を並べたベクトルも構築する。

$$\boldsymbol{X} = \left(0, \frac{1}{n}, \frac{2}{n}, \frac{3}{n}, \ldots, \frac{n-1}{n}, 1\right).$$

このようにして構築した $\boldsymbol{X}, \boldsymbol{Y}$ をプロットした点を結んだ線を**ローレンツ曲線**という。

問題 3.7. すべての個人が同じ所得を受け取っているとき，すなわち完全な平等が達成されているとき，ローレンツ曲線は $Y = X$ で示される直線になる。この事実を示しなさい。

完全な平等が達成されているときのローレンツ曲線を**完全平等線**と呼ぶ。図 3.1 の実線がローレンツ曲線の一例である。完全平等線と実際のローレンツ曲線に囲まれる領域（図 3.1 の影をつ付けた部分）の面積を 2 倍にした値がジニ係数である [8]。

$$G = \text{ジニ係数} = 2 \times (\text{ローレンツ曲線と完全平等線を囲む領域の面積})$$

定義より次の関係が成り立つ。

$$0 \leq G < 1$$

$G = 0$ はすべての個人が同じ所得を得ているときに限って成り立つ。$G = 1$ にもっとも近づくのは一人の個人が全所得を独占している場合である。完全な不平等を保った状態で総人口について $n \to \infty$ の極限を取れば，ジニ係数は 1 に収束する。この「完全な不平等」のケースについて次の問題を考えてみよう。

問題 3.8. $y_{(n)}$ を稼ぐ個人を除いて所得がゼロであるとする。すなわち，

$$0 = y_{(1)} = y_{(2)} = \cdots = y_{(n-1)} < y_{(n)}$$

[8] 完全平等線について対称にローレンツ曲線をもう 1 つ描くと，ジニ係数はこれら 2 つのローレンツ曲線で囲まれる部分の面積になる。

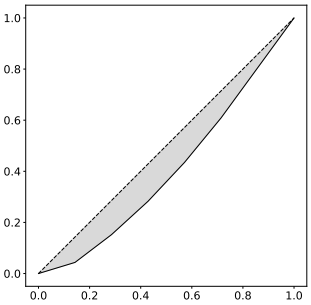


図 3.1.: ローレンツ曲線

が成り立つとする。この場合のローレンツ曲線を描きなさい。また，$n \to \infty$ の極限でジニ係数が 1 となることを示しなさい。

ジニ係数は 0 に近いほど不平等が小さく，1 に近ければ近いほど不平等が大きいものとみなせる。ジニ係数の国際比較や経年変化を見ることによって，各国の不平等を評価することができる。

注意 3.1. 個人の所得データからではなく，集計データ（例えば，所得階級ごと相対度数）からジニ係数を計算することも多い。詳細は 久保川・国友 (2016) を参照。

注意 3.2. 政府には，税や社会保障を通じて，所得の格差を縮小する役割がある。これを所得再分配という。再分配前のジニ係数と再分配後のジニ係数を比較することで，所得再分配政策の効果を測ることができる。

便宜上 $y_{(0)} = 0$ とする。n 個の台形の面積を足し上げればよいので，

$$\text{ローレンツ曲線の下側の面積} = \sum_{i=1}^{n} \left[\sum_{j=1}^{i} \frac{Y_{j-1} + Y_j}{2n} \right]$$

となる。したがって，ジニ係数は

$$\begin{aligned} G &= 1 - \sum_{i=1}^{n} \sum_{j=1}^{i} \frac{Y_{j-1} + Y_j}{n} \\ &= 1 - \frac{\sum_{i=1}^{n} Y_{j-1} + \sum_{j=1} Y_j}{n} \\ &= 1 - \frac{2\left(\sum_{j=1}^{n} Y_j\right) - Y_n}{n} \\ &= 1 - \frac{2\left(\sum_{j=1}^{n} Y_j\right) - 1}{n} \end{aligned} \tag{3.3}$$

とできる。総和記号 $\sum$ の部分は

$$\sum_{j=1}^{n} Y_j = \sum_{j=1}^{n} \left(\frac{y_{(1)} + y_{(2)} + \cdots + y_{(j)}}{y_{(1)} + y_{(2)} + \cdots + y_{(j)} + y_{(j+1)} + \cdots + y_{(n)}} \right)$$

とできることを思い出そう。データを小さい順に並べ替えて，累積和の和を計算している。

3.3.4. トップ 1% の所得シェア

超富裕層に富が集中している傾向について，近年大きな社会問題になっている。（ピケティ『21 世紀の資本』を参照する）

所得上位 1% の人々の所得が全所得に占める割合（top percentile income share）が，近年大幅に上昇しているというのだ。

上位 1% は下位 99% と同じなので，0.99-分位点以上の所得を合計すればよい。すなわち，

$$\text{(Top percentile income share)} = \frac{\sum_{i:y_i \geq Q_{0.99}} y_i}{\sum_i y_i}$$

問題 3.9. それぞれの定義に基づいて，ジニ係数とトップ 1% 所得シェアの類似性と相違点を説明しなさい。

3.4. プログラミング: 基礎

テキストファイルに書き込んだプログラムは基本的には上から順に実行される。例えば，次のコードは (1) 関数 `f()` を呼び出す，(2) 関数 `g()` を呼び出す，(3) 関数 `h()` を呼び出す，という順に実行される。

```
f()
g()
h()
```

注意 3.3. `f()`, `g()`, `h()` を定義していないので，このコードは実行できない。実行できないコードは理解しにくいという方は，次のように打ち込んでおくとよい。

```
def f(): print("f is called.")
def g(): print("g is called.")
def h(): print("h is called.")
```

□

改めて言うまでもない当たり前のことと感じるかもしれないが，これは非常に重要なことなので忘れないでほしい[9]。

線形の処理の流れを変更するための構文を**制御フロー構文**という。この節では，

- ループ
- 条件分岐

[9]特に，Jupyter Notebook などの環境でコードを書いているときには，コードの見かけ上の前後関係と実行順が異なる場合があるので混乱が生じやすい。気をつけよう。

について説明する。

Python ではループの使用はできる限り避けるべきであるとされる。**NumPy** を用いてベクトル化することでループを回避する方法，さらにはベクトル化によって避けることのできないループについても説明する。

3.4.1. ループ

ループは同じような処理を繰り返し実行するための構文である。`for` ループと `while` ループがある。

次のコードは同じコマンド `f()` を 3 回実行するためのコードである。

Code 3.1 実行したい処理

```
f()    # 0 回目
f()    # 1 回目
f()    # 2 回目
g()
```

3 回くらいであればこのままでもいいかもしれないが，100 回，1000 回，あるいは 100 万回繰り返す必要がある場合はどうすればよいだろう。

for ループ

`for` ループは繰り返しの実行回数が事前に分かっているときに使う構文である。次のように書く（ただし，これはまだ最善ではない）。

Code 3.2 for ループ

```
for i in [0, 1, 2]:
    f()

g()
```

図 3.2 に `for` ループの書き方を解説しているので確認してほしい。いちばん重要なポイントは，**実行される処理の本体にあたるコードは半角スペース 4 つ分のインデントで明示する**，ということである。実際には，コード全体で統一された数でありさえすれば 4 つでなくてもよいが，4 つ使うのが慣例である。したがって，Python でプログラミングをするときは Tab キーを押したときにスペース 4 つが入力されるように設定できるテキストエディタを使うべきである。

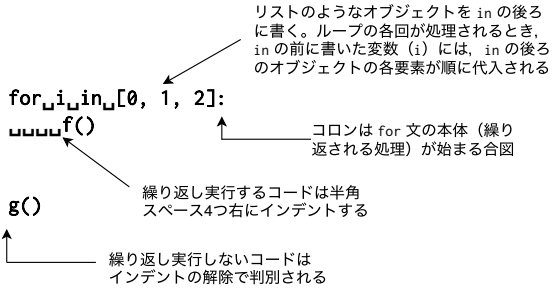


図 3.2.: for ループの文法

上の Code 3.2 は for ループ を使わない次のコードと同等である。

```
i = 0
f()

i = 1
f()

i = 2
f()

g()
```

使用されないカウンター i に代入する処理が増えたせいで，元のコード（Code 3.1 ）よりも長くなってしまった。しかし，for ループとはこういうものだ。実際には一時的に代入されるオブジェクトも使用できる [10]。

Code 3.3 カウンターを使う for ループ

```
for i in [0, 1, 2]:
    print(i)
```

```
0
1
2
```

[10] print() 関数はオブジェクトの情報を画面に表示する関数である。IPython や Jupyter を使っている場合に print() を使用する必要はないが，いずれ必要になる場面が出てくるので覚えておこう。

問題 3.10. Code 3.3 を for ループを使わないコードに書き換えなさい。

range()

上のコードには問題がある。[0, 1, 2] と書くのが面倒くさいということだ [11]。3 回繰り返しではなく 1000 回繰り返しだったらどうするのか。この問題は，range() 関数を使って解決できる。繰り返し回数が n であれば， in の後ろに range(n) と書けばよい。

```
for i in range(3):
    print(i)
```

```
0
1
2
```

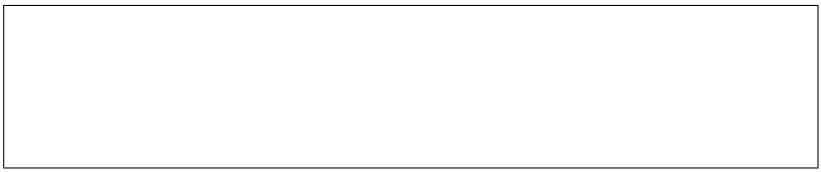
range(n) は [0, 1, ..., n-1] と同じような意味である 。[0, 1, ..., n-1] と書くと整数 n 個分のメモリが消費されるが，range(n) にはそのようなムダがないようにできている [12]。

問題 3.11. 整数 $i = 0, 1, \ldots, 9, 10$ について i^3 を表示するコードを書きなさい。

[11] 面倒なことを繰り返しやらされると人間は必ず間違える。そういう仕事はプログラミングを使ってなくさなければならない。

[12] list(range(n)) とすれば range オブジェクトが リストに変換される。for ループで使うときはリストに変換してはいけない。

問題 3.12. $m < n$ なる整数 m, n に対して，`range(m,n)` はどのような数列を表しているか。



3.4.2. 真偽値と while ループ

`while` ループは 繰り返しの実行回数が事前に分かっていなくても使える構文である [13]。`while` ループを理解するためには真偽値を理解する必要がある。

真偽値

Python では `True`, `False` という名前に特別な意味がある。次のコードを IPython で実行してみよう。

入力した結果がそのまま表示されるだけだが，それでよい。これらの名前と値が事前に定義されている証拠だ。

問題 3.13. `true`, `false`, `TRUE`, `FALSE` と入力するとどのような結果になるか。また，それは何故か。

[13] `for` と `while` に関するこのような切り分けは プログラミング言語一般に成り立つものではない。Python の `for` ループは range-based for loop と呼ばれる処理である。

True は「条件が成り立つ（真である）」ことを意味する値， False は「条件が成り立たない（偽である）」ことを意味する値である。and（かつ）,or（または）, not （否定）の演算がある。

```
[True and True, True and False, False and True, False and False]
```

```
[True, False, False, False]
```

```
[True or True, True or False, False or True, False or False]
```

```
[True, True, True, False]
```

```
[not True, not False]
```

```
[False, True]
```

条件式

結果が真偽値となる式を条件式という。数値比較のための以下の演算子が基本的である。

表 3.1.: 数値比較の演算子

x == y	x と y は等しい
x < y	x は y より小さい
x <= y	x は y より小さいか等しい
x > y	x は y より大きい
x >= y	x は y より大きいか等しい

```
[1 < 2, 1 <= 2, 1 == 2, 1 > 2, 1 >= 2]
```

```
[True, True, False, False, False]
```

問題 3.14. 次の各条件を表す Python の条件式を書きなさい。

1. $x \geq y$ または $x \geq z$
2. $x < y < z$

真偽値の足し算

真偽値に対して四則演算を実行しようとすると，

- `True` $\longrightarrow 1$
- `False` $\longrightarrow 0$

に勝手に変換される。この性質を使うと，リストや配列の中の `True` の数を数えるのに役立つ。

```
True + True + False
```

```
2
```

浮動小数点数に関する注意

コンピュータで実数を扱う場合，**浮動小数点数**と呼ばれる近似を用いる。あくまで近似なので，厳密な等式で比較することは避けた方がよい。私たちが有限小数と認識している普通の小数であっても，コンピュータは正確に表現できないものがある。次の 2 つのコードを比較してみよう。

```
0.1 + 0.3 + 0.6 == 1.0
```

```
True
```

```
0.3 + 0.6 + 0.1 == 1.0
```

```
False
```

浮動小数点数が計算に現れる場合，数値が近似的に等しいことを np.allclose() といった関数を用いて評価する。浮動小数点数の厳密な等号比較が必要になるような処理は不可能である。この話は行列の標準化を扱う際に重要になる。

```
import numpy as np
np.allclose(0.1 + 0.3 + 0.6, 1.0, rtol=1e-15)
```

```
True
```

rtol=1e-15 とあるのは，この基準よりも小さい誤差はゼロとみなすということ。1e − 15 = 1×10^{-15} という意味である。

while ループ

繰り返し処理の継続条件が条件式で表されているときには while ループを使う。基本は次のような形式になる。処理の本体にあたる部分をスペース 4 つ分のインデントで区別する点などは for ループと同じである。

```
while condition:
    f()
```

ここで condition の部分に条件式が入る。

- condition が True であればループの本体を実行し，次のループに進む。
- condition が False であれば ループを抜ける。

もちろん，while ループが正しく終了するためには，各回の処理のたびに条件式が変わっていく必要がある。例えば次のコードは，Code 3.2 を while ループを使って書き直したものだ。ただし，i += 1 は i = i + 1 と同じ意味である。

```
i = 0
while i < 3:
    f()
    i += 1

g()
```

問題 3.15. 次のコードを IPython で実行すると何が起こるか。現象と理由を説明しなさい。（実行を終了するときはキーボードで Ctrl + C を押す）

```
while True:
    print(i)
    i += 1
```

while ループは非常に複雑なループ処理もできるが，本章では使わないので，紹介はこれくらいにしておく。もっと実用的な while ループについては必要になったときに説明する。

3.4.3. 条件分岐

if 文を使うと，指定した条件が成り立つときにだけ実行される処理を指定することができる。条件が成り立たない場合の処理は else ブロックで指定する。実行条件が 2 つ以上ある場合には elif ブロックを用いる。elif ブロックは複数書くことができる。処理の本体にあたる部分をスペース 4 つ分のインデントで区別する点などは for，while の構文と同じである。

```
if condition:
    f()
```

```
if condition:
    f()
else:
    g()
```

```
if condition1:
    f()
elif condition2:
    g()
```

```
else:
    h()
```

ループと組み合わせると面白いことができる。

例 3.1. 1 から 100 の間に含まれる奇数の合計は次のように計算できる。

Code 3.4 奇数の合計

```
total = 0
for i in range(1, 101):
    if i % 2 == 1:
        total += i

total
```

```
2500
```

if の本体ブロック（total += i）の先頭に 8 個分のスペースが挿入されている。for のために 4 つ，if のために 4 つである。 □

問題 3.16. 1 から 1000 の間に含まれる偶数の合計を計算するコードを書きなさい。

3.4.4. ジェネレータ内包表記

Code 3.4 は実際には次のように書かれることが多い。ここでは詳しい説明は省くが，ジェネレータ内包表記と呼ばれる書き方である。英文と思って読めば，なんとなく使い方が分かるはずだ。

```
sum(i for i in range(1, 101) if i % 2 == 1)
```

```
2500
```

もちろん，sum() という関数（np.sum() ではない！）がこのような書き方を許しているというだけであって，あらゆる関数に使える訳ではない。

3.4.5. 真偽値の配列

NumPy 配列を比較演算子で比較した結果は真偽値の **NumPy** 配列になる。

```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
x < y
```

```
array([ True, False, False])
```

要素取得の角括弧の中に真偽値の配列を入れる。結果は True の位置にあたる要素である。

```
x[x < y]
```

```
array([1])
```

True が 1，False が 0 であることを思い出そう。0 と 1 からなるデータの総和を取ると 1 の個数，平均を取ると 1 の割合が計算できる。したがって，次のコードは 0.05 に近い値になると期待される。

```
z = np.random.random(1000)
np.mean(z > 0.95)
```

```
0.07
```

3.5. プログラミング: 実践

前置きが長くなった。数値例を使って理論パートで学んだ計算方法を確認しよう。

3.5.1. 連鎖方式の実質 GDP と GDP デフレータ

前章で用いた簡単な数値例を用いよう。A，B，C の財グループがあるとして，2000 年を基準に価格指数を計算したものが下表左であり，生産数量が下表右であるとする。

表 3.2.: 実質 GDP の計算例

価格	A	B	C
2000 年	100	100	100
2001 年	101	99	103
2002 年	100	98	104
2003 年	99	99	106

数量	A	B	C
2000 年	1000	2000	500
2001 年	980	1980	510
2002 年	1010	1990	520
2003 年	1005	2005	530

実際の価格ではなく価格指数を使うことで一般性が失われるということはない。実質 GDP にせよ価格指数にせよ，価格上昇率のみに意味があり，水準には意味がないからだ。理由が分からない場合は理論パートの式を丁寧に検証し直すとよい。

名目 GDP，GDP デフレータ，実質 GDP の順に計算しよう。3 群 × 4 年分の人工的なデータを用いたが，適切な形式でデータを格納していれば 3 とか 4 とかいう数字はコードには出てこない。したがって，ここで紹介するコードは，何千もの商品群を考慮している実際の計算でも同じように使えるはずだ。心してかかろう。

まずは表を配列形式で格納する。データの格納形式とその後のコードには密接に関連している。行が伸びる方向に時間が進んでいくものとしよう。配列全体が実数のデータであることを Python に知らせるためには，少なくとも 1 つの数字に小数点をつければよい [14]。

```
price = np.array([[100, 100, 100],
                  [101, 99, 103],
                  [100, 98, 104],
                  [99, 97, 106.]])
quantity = np.array([[1000, 2000, 500],
                     [980, 1980, 510],
                     [1010, 1990, 520],
                     [1005, 2005, 530.]])
```

名目 GDP を計算するためには，この表を成分ごとにかけてから（演算子は `*` だ），結果の配列の各行について，全列の和を取る。これは，`np.sum(x, axis=1)` でできる。

```
nominal_gdp = np.sum(price * quantity, axis=1)
```

[14] NumPy の配列には同じ種類のデータが並ぶことを思い出そう。1 つが浮動小数点数なら全部が浮動小数点数になる。

```
nominal_gdp
```

```
array([350000., 347530., 350100., 350160.])
```

式 (3.2) に基づいてデフレータを計算するのが次のコードだ。前章で学んだコード（連鎖方式のラスパイレス指数を計算したことを思い出そう）とこの章で学んだ for の構文を組み合わせればよい。

```
dfl = np.empty_like(nominal_gdp)
dfl[0] = 1.
for t in range(1, len(dfl)):
    dfl[t] = dfl[t-1] * (np.sum(price[t, :] * quantity[t, :])
                         / np.sum(price[t-1, :] * quantity[t, :]))

dfl * 100
```

```
array([100.        , 100.15273775,  99.44827695,
       98.89752821])
```

実質 GDP は名目 GDP をデフレータで除すことで計算できる。

```
real_gdp = nominal_gdp / dfl
real_gdp
```

```
array([350000.        , 347000.        , 352042.2985066 ,
       354063.44966341])
```

3.5.2. 寄与率と寄与度

複数年にわたる計算

表 3.2 の例を使って計算しよう。ここでは，名目価値の成長要因を分析する。実質変数の場合も，各項目の実質値の推計が手に入れば同じように実行できる。使うデータは price * value を計算した次のデータである。

```
value = price * quantity
value
```

```
array([[100000., 200000.,  50000.],
       [ 98980., 196020.,  52530.],
       [101000., 195020.,  54080.],
```

```
       [ 99495., 194485.,  56180.]])
```

価格	A	B	C	GDP	成長率
2000 年					
2001 年					
2002 年					
2003 年					

各行の総和はすでに計算した nominal_gdp に一致する。これは計算の方法から当たり前である。

```
np.allclose(nominal_gdp, np.sum(value, axis=1), rtol=1e-15)
```

```
True
```

名目 GDP の成長率は次のように計算できる。np.diff() は隣接する要素の差を計算する関数である。デフォルトでは元の変数よりも 1 つ長さが短いベクトルが出力される。この振る舞いを prepend=np.nan を付けることで変えることができる [15]。これは，出力の先頭要素に「存在しない値」を意味する NaN (Not a Number) を追加する設定である [16]。NaN を付加すれば，名目 GDP の成長率は次のように簡単に計算できる。

```
np.diff(nominal_gdp, prepend=np.nan) / np.roll(nominal_gdp, shift=1)
```

```
array([         nan, -0.00705714,  0.00739505,  0.00017138])
```

寄与度を計算するためには，まず各項目の増分を計算しないといけない。これにはデータ行列 value に対して np.diff() を取ればよい。ただし，axis=0 を付けて行が伸びる方向に差分を取ることを指定する。まとめると，np.diff(value, prepend=np.nan, axis=0) が各項目の差分時系列を表している [17]。これを名目 GDP

[15] **NumPy** のバージョンが v1.16.0 よりも古い場合，prepend=np.nan が使えない。この場合は **NumPy** のアップデートをお勧めする。その権限がない場合には，np.r_[[np.nan], np.diff(nominal_gdp)] / nominal_gdp などとする。

[16] 本来，NaN は 0.0/0.0 のように答えが定められない計算の結果を示す値である。Python には「欠損」（Not Available, missing）を表すための専用の値がなく，**NumPy** や **math** ライブラリがサポートする NaN を欠損に転用している。しかし，R や Julia のように言語レベルで欠損をサポートするプログラミング言語と違って，欠損値として一貫性のある振る舞いは期待できない。Python 用のデータ分析ライブラリである **pandas** には pandas.NA という欠損を表すための値が最近になって追加されたので，この状況は少しずつ改善されるだろう。

[17] **NumPy** のバージョンが v1.16.0 よりも古い場合，prepend=np.nan が使えない。この場合は **NumPy** のアッ

で除したものが寄与度である。単純に nominal_gdp で割るとサイズが合わないというエラーになるので，4×1 配列であることを明示するために nominal_gdp.reshape(4, 1) で割るとよさそうだ。しかし，これでは $\Delta X_{t+1}/Y_{t+1}$ を計算してしまうので，np.roll(nominal_gdp, shift=1) を先に

```
contribution = (np.diff(value, prepend=np.nan, axis=0)
                / np.roll(nominal_gdp, shift=1).reshape(4, 1)) * 100
contribution
```

```
array([[         nan,          nan,          nan],
       [-0.29142857, -1.13714286,  0.72285714],
       [ 0.58124478, -0.28774494,  0.44600466],
       [-0.42987718, -0.15281348,  0.59982862]])
```

最後に，寄与度の合計を計算すると成長率になることを確認しておこう。

```
contribution.sum(axis=1)
```

```
array([         nan, -0.70571429,  0.7395045 ,  0.01713796])
```

3.5.3. 不平等指数

相対的貧困率

相対的貧困線の計算に必要な中央値は np.median() を用いる。

```
x = np.array([3, 6, 10, 14, 3, 2, 1, 1])
np.median(x)
```

```
3.0
```

相対的貧困に該当するデータを発見するには次のようにする。

```
x < np.median(x) / 2
```

```
array([False, False, False, False, False, False,  True,
       True])
```

このデータのうち True の割合を計算したものが相対的貧困率である。

アデートを改めてお勧めする。どうしても認めてもらえない場合には，np.diff(..., prepend=np.nan) の部分を np.r_[[[np.nan] * value.shape[1]], np.diff(value, axis=0)] などと置き換えればよい。

```
np.mean(x < np.median(x) / 2)
```

```
0.25
```

問題 3.17. 節 3.3.2 の例に対して相対的貧困率を計算するプログラムを書きなさい。

ジニ係数

次の人工的なデータに対してジニ係数を計算しよう。

```
x = np.random.choice(np.arange(30.), 10)
x
```

```
array([19., 29., 17., 19.,  5., 18., 10., 26., 13., 12.])
```

ジニ係数を計算するためには，まずデータを小さい順に並べ替える必要がある [18]。

```
y = np.sort(x)
y
```

```
array([ 5., 10., 12., 13., 17., 18., 19., 19., 26., 29.])
```

式 (3.3) に基づいて計算する。Y_i は累積和を総和で割ったものだから，次のように計算すればよい。データの先頭にゼロを付加していることに注意。

```
Y = np.r_[0, y.cumsum()] / y.sum()
Y
```

```
array([0.        , 0.0297619 , 0.08928571, 0.16071429,
       0.23809524, 0.33928571, 0.44642857, 0.55952381,
       0.67261905, 0.82738095, 1.        ])
```

[18] 中央値の計算のためにも並べ替えは必要だが `np.median()` がその工程を隠蔽している

最後に式 (3.3) の公式に当てはめればよい。

```
gini = 1 - (2 * np.sum(Y) - 1) / y.size
gini
```

```
0.22738095238095235
```

トップ 1% 所得シェア

Q_q 統計量は `np.quantile(a, q)` で求められる。ここで，`a` が配列形式のデータで，`q` は下側 $100q$ % を表す $[0,1]$ の数字である。トップ 1% の所得水準以上の所得というのは，

$$\text{所得} \geq Q_{0.99}$$

と書けるので，次のように書けばトップ 1% のデータを判別できる。

```
x[x >= np.quantile(x, 0.99)]
```

```
array([29.])
```

この総和を計算して，全体に占める割合を計算しよう。

```
x[x >= np.quantile(x, 0.99)].sum() / x.sum()
```

```
0.17261904761904762
```

4. 時系列データのモデリングと行列計算

4.1. 概要

この章では時系列データ分析の入門的な話題を紹介するとともに Python で行列計算を行う方法を解説する。

最初にお断りをしておくが，この講義では，洗練された統計学を使った高度な統計分析を行わない。データを可視化したり，データの背後にある傾向を探す作業を体験してもらうのが目的だ。本格的な分析を始める前にデータを眺めるプロセスだと考えてほしい。

本章の理論パートでは，経済理論から離れて，データの見方やデータの背後にある構造を知るためのツールを学ぶ。非常に簡単なモデルだけ紹介するので，より進んだ学習をしたい読者は「時系列分析」をキーワードに参考資料を探すとよい。高校までで学んだ数学や大学初年次の線形代数がどのように役立つかを紹介したい。すぐに何をやっているか分からなくてもいいので，一通り目を通してほしい。

プログラミング・パートでは，時系列モデルのシミュレーションについて説明する。連鎖方式の指数計算で用いたテクニックが使われていることを意識しながら手を動かしてほしい。最後に，データからパラメータを推測する方法を簡単に紹介する。

4.2. 理論

4.2.1. 簡単な時系列モデル: 高校数学の復習

読者が日本で高校を卒業していれば，文理問わず時系列モデルの原型を学んでいるはずだ。それは「漸化式」と呼ばれる次のような式だ。

$$a_{n+1} = pa_n + q \tag{4.1}$$

(4.1) をどのように解いたかを思い出してみよう。まず「特性方程式」という次

の方程式を立てる。

$$a = pa + q \tag{4.2}$$

そして a について解く。さしあたり $p \neq 1$ として，

$$a = \frac{q}{1-p}$$

である。(4.1) と (4.2) の差を取ると，

$$a_{n+1} - a = p\,(a_n - a)$$

これは，$a_n - a$ という形が繰り返し現れることに注意すると，

$$\begin{aligned} a_n - a &= p\,(a_{n-1} - a) \\ &= p^2\,(a_{n-2} - a) \\ &= p^3\,(a_{n-3} - a) \\ &= \cdots \\ &= p^n\,(a_0 - a) \end{aligned}$$

a_0 が既知の情報（初期条件）とすれば，すべての $n = 1, 2, \ldots$ について

$$a_n = a + p^n\,(a_0 - a) \tag{4.3}$$

という公式が得られる。この作業を「漸化式を解く」と学んだ。

ここで，$n \to \infty$ についての極限挙動について確認しておこう。上で計算した a は

$$a = \cdots = a_{n+1} = a_n = a_{n-1} = \cdots$$

が成り立つ数字であった。初期条件がたまたま $a_0 = a$ を満たしていれば，漸化式の解 a_n はずっと a であり続ける。もし $a_0 \neq a$ であっても，$|p| < 1$ であれば $\lim_{n\to\infty} p^n = 0$ すなわち $\lim_{n\to\infty} a_n = a$ が成り立つ。a は漸化式の長期的な挙動を代表する重要な値であり，**定常状態**とか**平衡点**と呼ばれる。$|p| > 1$ であれば，例外的な $a_0 = a$ のケースを除いて，$\lim_{n\to\infty} |a_n| = \infty$ となる。

(4.3) をもう少し変形しておこう。

$$\begin{aligned} a_n &= a + p^n (a_0 - a) \\ &= p^n a_0 + \left(1 - p^n\right) a \\ &= p^n a_0 + \left(1 - p^n\right) \frac{q}{1-p} \\ &= p^n a_0 + \left(1 + p + p^2 + \cdots + p^{n-1}\right) q \end{aligned} \tag{4.4}$$

a_n の決定には 2 つの要因が絡んでいる。

1. 1 つは初期値 a_0 の影響である。p^n は「n 期の過去の情報が現在に与える影響」を取り出す効果があるので，$p^n a_0$ は「(n 期から見て n 期前であるところの) 0 期の情報 a_0 が a_n に与える影響」となっている。
2. もう 1 つは，毎期毎期新たに加算される q の影響である。こちらは 0 期前 ($p^0 = 1$)，1 期前 ($p^1 = p$)，...，$(n-1)$ 期前の q がすべての a_n に効いている。

なお，具体的な a_n の値を求めたいだけなら漸化式を解く必要はない。式 (4.1) を順番に計算していけばよいからだ。しかし,「漸化式を解く」ことで数値計算によらずとも長期的な振る舞いを理解できるようになる。収束・発散の境界は等比数列の拡大率 p の絶対値が 1 のところにある。

4.2.2. AR(1) 過程

時系列分析の文脈では,

$$y_t = a_0 + a_1 y_{t-1}$$

という方程式をベースとしたモデルがよく用いられる。なんらかの変数 (y) の時点 t における観測値 y_t が前期の観測値 y_{t-1} に依存することを表している。a_0, a_1 は定数，$y_1, y_2, \ldots$ が分析対象である。形式的に (4.1) と同じであることが分かるだろう。上の方程式に，事前には予測できない不確実性 (ε_t) を加えたものが時系列分析の基礎モデルである。

$$y_t = a_0 + a_1 y_{t-1} + \varepsilon_t \tag{4.5}$$

方程式 (4.5) で表現される y は「**AR(1) 過程**に従う」と言われる。単に y は AR(1) 過程である，と言ってもよい。AR は autoregressive の略で「自己回帰」と訳す。

過去の自分自身のデータ（y_{t-1}）を説明変数，今の自分（y_t）を被説明変数とした回帰方程式になっている。

上で導入した ε_t について,「時点 $t-1$ にいる分析者は何の情報も持っていない」ことを前提とする。もし情報を持っているならそれはモデルを改善するために使っているはずだからだ。この「時点 $t-1$ にいる分析者は何の情報も持っていない」ということを，数学的には

$$\mathbb{E}_{t-1}[\varepsilon_t] = 0$$

と表現する。$\mathbb{E}_{t-1}[\cdot]$ は条件付き期待値を表す記号である。確率論にアレルギーを持っている人もいるかも知れないが，あまり難しく考えなくてもよい。モデル外の情報（ε_t）は y_t を増やすのか減らすのかも分からないはずだから，ゼロにならなければならないというだけのことだ。通常 ε_t は，ε_t と ε_k $(t \neq k)$ が独立で，平均ゼロ，$\mathrm{Var}(\varepsilon_t)$ = 一定 が仮定される（ホワイト・ノイズの仮定）。次のことに注意する。

$$\mathbb{E}_{t-1}[y_{t-1}] = y_{t-1}, \quad \mathbb{E}_{t-1}[a_0] = a_0$$

$t-1$ 時点にいる観測者は y_{t-1} をすでに観測しているし，定数 a_0 についてはずっと前から知っているのだ。(4.5) の全体に $\mathbb{E}_{t-1}[\cdot]$ を付けると [1]，

$$\begin{aligned}\mathbb{E}_{t-1}[y_t] &= \mathbb{E}_{t-1}[a_0 + a_1 y_{t-1} + \varepsilon_t] \\ &= \mathbb{E}_{t-1}[a_0] + a_1 \mathbb{E}_{t-1}[y_{t-1}] + \mathbb{E}_{t-1}[\varepsilon_t] \\ &= a_0 + a_1 y_{t-1}\end{aligned}$$

となる。つまり，AR(1) モデルは，今期 $(t-1)$ の情報を用いて次期 (t) に観察されそうな値（期待値）を決定するモデルである。

4.2.3. AR(2) 過程

AR(1) があれば AR(2) も AR(3) もある。一般に AR(p) モデルと呼ばれる時系列モデルは

$$y_t = a_0 + a_1 y_{t-1} + \cdots + a_p y_{t-p} + \varepsilon_t \tag{4.6}$$

と書ける。以下では，AR(2) 過程と漸化式の関係，行列形式の表現を見ていこう。

[1] 期待値の線形性 $\mathbb{E}_{t-1}[\alpha x + \beta y] = \alpha \mathbb{E}_{t-1}[x] + \beta \mathbb{E}_{t-1}[y]$ となることは認めよう。心配な読者は適当な確率論のテキストを参照してほしい。

高校時代の解きかた（線形代数を使わない）

次の形式の漸化式を解く方法を覚えているだろうか。もう忘れてしまった人もいるかもしれないが，これも高校数学で履修済みのはずだ。

$$a_{n+2} = pa_{n+1} + qa_n + r$$

定数項 r が鬱陶しいので，最終的に定常状態に収束したらどうなるかな，と考えて次の方程式を解く。

$$a = pa + qa + r \Longrightarrow a = \frac{r}{1-p-q}$$

である。$b_n = a_n - a, n = 0, 1, 2, 3, \ldots$ として，

$$b_{n+2} = pb_{n+1} + qb_n$$

という定数項のない漸化式になる。これを解くときに，誰かが次のような形に変形することを閃いたらしい。

$$b_{n+2} - \alpha b_{n+1} = \beta(b_{n+1} - \alpha b_n)$$

これを満たす α, β を見つけ出そうという訳である。少し変形すると，

$$b_{n+2} - (\alpha + \beta)b_{n+1} + \alpha\beta b_n = 0$$

これは

$$p = \alpha + \beta, \quad q = \alpha\beta$$

なのだから，2 次方程式

$$x^2 - px + q = 0$$

の解によって α, β を決定できる（俗に「解と係数の関係」と呼ばれる関係）[2]。したがって，最初の 2 項 b_0, b_1 が既知であれば，

$$b_{n+1} = \alpha b_n + \beta^n(b_1 - \alpha b_0)$$

のようにして，一般の b_n あるいは a_n を決定できる。$\lim_{n\to\infty} |b_n|$ が $|\beta|$ に関係していることは明白だろう。しかし，右辺 1 項目 $b_{n+1} = \alpha b_n + \cdots$ に注目すると，

[2] ここでは α, β が実数で $\alpha \neq \beta$ を満たすことを想定している。

$|\alpha|$ も無関係ではなさそうだ。実際，$|\alpha| < 1$ かつ $|\beta| < 1$ が成り立っているときに，$\lim_{n\to\infty} |b_n| = 0$ が成り立つ。これは線形代数学の知識を使うと明快に示すことができる。

線形代数を使う

上の解法は多次元に一般化することは難しい。より実用的な方法を理解するには線形代数学の知識が必要になる。

$$a_{n+2} = pa_{n+1} + qa_n + r$$

次の行列型の漸化式に変形する。

$$\begin{bmatrix} a_{n+2} \\ a_{n+1} \end{bmatrix} = \begin{bmatrix} p & q \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a_{n+1} \\ a_n \end{bmatrix} + \begin{bmatrix} r \\ 0 \end{bmatrix},$$

ここの行列漸化式は次の連立方程式と同等である。

$$\begin{cases} a_{n+2} = pa_{n+1} + qa_n + r \\ a_{n+1} = a_{n+1} \end{cases}$$

さらに表現を簡略化するために次のような記法を導入する。

$$\boldsymbol{a}_n = \begin{bmatrix} a_{n+1} \\ a_n \end{bmatrix}, \boldsymbol{A} = \begin{bmatrix} p & q \\ 1 & 0 \end{bmatrix}, \quad \boldsymbol{c} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

すると，

$$\boldsymbol{a}_{n+1} = \boldsymbol{A}\boldsymbol{a}_n + \boldsymbol{c} \tag{4.7}$$

と書ける。自明な等式を導入することで表現をシンプルにすることができた。式 (4.4) と同じスピリットで一般の $\boldsymbol{a}_n$ を表現してみよう。

$$\begin{aligned}\boldsymbol{a}_n &= \boldsymbol{A}\boldsymbol{a}_{n-1} + \boldsymbol{c} \\ &= \boldsymbol{A}\left(\boldsymbol{A}\boldsymbol{a}_{n-2} + \boldsymbol{c}\right) + \boldsymbol{c} \\ &= \boldsymbol{A}^2\boldsymbol{a}_{n-2} + \left(\boldsymbol{A} + \boldsymbol{I}\right)\boldsymbol{c} \\ &= \cdots \\ &= \boldsymbol{A}^n\boldsymbol{a}_0 + \left(\boldsymbol{A}^{n-1} + \boldsymbol{A}^{n-2} + \cdots + \boldsymbol{A} + \boldsymbol{I}\right)\boldsymbol{c} \end{aligned} \tag{4.8}$$

この表現は一般の $\boldsymbol{A}, \boldsymbol{c}$ に対して正しい公式になっているが，極限挙動 $\lim_{n\to\infty} \boldsymbol{a}_n$ については何も教えてくれない。$\boldsymbol{A}^n$ の極限での振る舞いについて調べる必要があるのだが，これには行列の対角化，ジョルダン標準化という考え方を使う。詳細は省略するが，

$$\begin{aligned}\boldsymbol{A} &= \boldsymbol{V}\boldsymbol{D}\boldsymbol{V}^{-1} \\ \boldsymbol{D} &= \text{対角成分に } \boldsymbol{A} \text{ の固有値が並ぶ上三角行列} \\ &= \begin{bmatrix} \alpha & * \\ 0 & \beta \end{bmatrix}\end{aligned}$$

という形式で表現できて，$\boldsymbol{D}^n$ の成分は比較的簡単に計算できる。α, β は前節のものと同じと考えてもよいが，実数に限らず複素固有値も許容される。もし $\boldsymbol{A}$ のすべての固有値の絶対値が 1 未満であれば，$\lim_{n\to\infty} \boldsymbol{D}^n = \boldsymbol{0}$。さらにそこから $\lim_{n\to\infty} \boldsymbol{A}^n = \boldsymbol{0}$ を導くことができる。固有値に絶対値が 1 より大きいものがあれば，$\boldsymbol{A}^n$ は発散する。固有値に絶対値が 1 と等しいものがある場合には少し難しいのだが，発散せずに一定の大きさを保ち続けるケースと発散するケースがある。極限での振る舞いに基づいて，固有値を分類しておこう。

- 絶対値が 1 未満の固有値を**安定固有値**という。
- 絶対値が 1 より大きい固有値を**不安定固有値**という。
- 絶対値が 1 に等しい固有値を**中心固有値**という。

高校数学で $b_{n+2} - \alpha b_{n+1} = \beta\left(b_{n+1} - \alpha b_n\right)$ を考えたのは，固有値に対応する固有空間という概念に対応している。$-1 < \beta < 1$ のときには $n \to \infty$ の極限で (b_n, b_{n+1}) は $y = \alpha x$ を満たす直線に漸近する。この直線が固有空間の 1 つである（固有値 α

に対応する固有空間)。行列表現した上記の $\boldsymbol{V}$ に関連しているのだが，きちんとした説明にはそれなりの準備が必要なので，関心のある読者は適当な線形代数学や力学系の教科書で勉強してほしい。

重要な性質を命題の形でまとめておこう。

命題 4.1. 行列 $\boldsymbol{A}$ が安定固有値のみをもつとき，

$$\lim_{n\to\infty} \boldsymbol{A}^n = \boldsymbol{0}$$

が成り立つ。 □

なお，行列の極限は成分ごとに考えればよい。

安定固有値のみを持つ行列については，無限等比級数の公式（$|\rho| < 1$)

$$\sum_{n=0}^{\infty} \rho^n = 1 + \rho + \rho^2 + \cdots = \frac{1}{1-\rho}$$

と同等の公式が成り立つ。

命題 4.2. 行列 $\boldsymbol{A}$ が安定固有値のみをもつ場合，行列 $(\boldsymbol{I} - \boldsymbol{A})$ は逆行列を持ち，次の式が成り立つ。

$$\sum_{n=0}^{\infty} \boldsymbol{A}^n = \lim_{n\to\infty} \left(\boldsymbol{A}^{n-1} + \boldsymbol{A}^{n-2} + \cdots + \boldsymbol{A} + \boldsymbol{I} \right) = (\boldsymbol{I} - \boldsymbol{A})^{-1}$$

証明. 証明するには

$$(\boldsymbol{I} - \boldsymbol{A}) \left[\lim_{n\to\infty} \left(\boldsymbol{A}^{n-1} + \boldsymbol{A}^{n-2} + \cdots + \boldsymbol{A} + \boldsymbol{I} \right) \right] = \boldsymbol{I}$$
$$\left[\lim_{n\to\infty} \left(\boldsymbol{A}^{n-1} + \boldsymbol{A}^{n-2} + \cdots + \boldsymbol{A} + \boldsymbol{I} \right) \right] (\boldsymbol{I} - \boldsymbol{A}) = \boldsymbol{I}$$

を確かめればよい。 □

命題 4.3. 行列 $\boldsymbol{A}$ が安定固有値のみをもつ場合，*(4.7)* は定常状態に収束する。

証明. 上の命題 4.1, 4.2, 式 (4.8) を用いると，

$$\boldsymbol{a}_n = \boldsymbol{A}^n \boldsymbol{a}_0 + \left(\boldsymbol{A}^{n-1} + \boldsymbol{A}^{n-2} + \cdots + \boldsymbol{A} + \boldsymbol{I} \right) \boldsymbol{c}$$
$$\xrightarrow[n\to\infty]{} (\boldsymbol{I} - \boldsymbol{A})^{-1} \boldsymbol{c}$$

が分かる。 □

式 (4.7) のようにかけるモデルを線形差分方程式と呼ぶ。線形差分方程式が極限でよい振る舞いをするかどうかは，係数行列の固有値を調べればよい。これは非常に重要な性質なので記憶しておこう。

問題 4.1. 3 項間漸化式を 2 次ベクトルの漸化式に書き換えることができたのだから，一般の $(p+1)$ 項間漸化式を p 次ベクトルの漸化式に書き換えることができるはずだ。やってみよう。

$$b_n = \alpha_0 + \alpha_1 b_{n-1} + \alpha_2 b_{n-2} + \cdots + \alpha_{n-p} b_{n-p}$$

線形代数を使うと，次数の大きさに関わらずに同じ表現とアルゴリズムを使うことができるという大きなメリットがある。線形代数学を学ぼう。

AR(2) 過程

次の形式の時系列モデルを AR(2) 過程という。

$$y_t = a_0 + a_1 y_{t-1} + a_2 y_{t-2} + \varepsilon_t \tag{4.9}$$

y_0, y_1 が観測されていれば,

$$\begin{aligned} y_2 &= a_0 + a_1 y_1 + a_2 y_0 + \varepsilon_2 \\ y_3 &= a_0 + a_1 y_2 + a_2 y_1 + \varepsilon_3 \\ &\vdots \\ &\vdots \end{aligned}$$

を順に計算できる。もちろん，ε_t は $\mathbb{E}_{t-1}[\varepsilon_t] = 0$ （$\mathbb{E}[\varepsilon_t] = 0$），$\mathrm{Var}(\varepsilon_t) \equiv \sigma^2$ を満たすホワイトノイズで観察のたびにランダムに決まる。

次の行列表現と同じであることを確かめよう。

$$\begin{aligned} \boldsymbol{x}_t &= \boldsymbol{A}\boldsymbol{x}_{t-1} + \boldsymbol{B}\tilde{\varepsilon}_t \\ y_t &= \boldsymbol{C}\boldsymbol{x}_t, \end{aligned}$$

ただし，

$$\boldsymbol{x}_t = \begin{bmatrix} y_t \\ y_{t-1} \end{bmatrix}, \quad \boldsymbol{A} = \begin{bmatrix} a_1 & a_2 \\ 1 & 0 \end{bmatrix}, \quad \boldsymbol{B} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \boldsymbol{C} = \begin{bmatrix} 1 & 0 \end{bmatrix}, \quad \tilde{\varepsilon}_t = \varepsilon_t + a_0.$$

上で用いた行列表現とは違って，中間変数 $\boldsymbol{x}_t$ を導入した。このようなモデル表現を「状態空間表現」というが，今はあまり気にしなくてよい。係数行列の固有値は確率的な変動が入る場合にも役に立つ。行列 $\boldsymbol{A}$ が安定固有値のみを持つことは，時系列モデルの**弱定常性**と呼ばれる性質に関連している。弱定常性を持つ過程は

- すべての時点の変数 y_t が同じ期待値と分散を持つ。すなわち，ある定数の周りをおおよそ同じ振れ幅で振動するような挙動を示す。
- 過去のデータが現在のデータに与える影響は，現在と過去の時間差のみに依存する。例えば，3 日前のデータ今日のデータに与える影響の強さは，2 日前のデータが明日のデータに与える影響の強さと同じである。

AR 過程が 1 つ目の性質を持つことは次のようにして分かる。まず，$\boldsymbol{x}_t$ が下のように表現できることを確認する。

$$\boldsymbol{x}_t = \boldsymbol{A}^n \boldsymbol{x}_{t-n} + \sum_{k=0}^{n-1} \boldsymbol{A}^k \boldsymbol{B}\tilde{\varepsilon}_{t-k}, \quad n = 1, 2, \ldots$$

であることに注意する。両辺の期待値を取ると，（$\mathbb{E}[\tilde{\varepsilon}_t] = a_0$ に注意）

$$\begin{aligned} \mathbb{E}[\boldsymbol{x}_t] &= \boldsymbol{A}^n \mathbb{E}[\boldsymbol{x}_{t-n}] + \sum_{k=0}^{n-1} \boldsymbol{A}^k \boldsymbol{B}\mathbb{E}[\tilde{\varepsilon}_{t-k}] \\ &\xrightarrow[n\to\infty]{} \sum_{k=0}^{\infty} \boldsymbol{A}^k \boldsymbol{B} a_0 \\ &= (\boldsymbol{I} - \boldsymbol{A})^{-1} \boldsymbol{B} a_0 \end{aligned}$$

したがって，

$$\mathbb{E}[y_t] = \boldsymbol{C}(\boldsymbol{I} - \boldsymbol{A})^{-1}\boldsymbol{B}a_0 \equiv \mu$$

となり，すべての時点で 平均が同じになる。

問題 4.2. AR(2) 過程 (4.9) に対して $\boldsymbol{\mu}$ を計算しなさい。

すべての t について分散が同一であることは以下のように示せる。この計算では $a_0 = 0$（したがって，$\mu = 0$）と置くが，これによって一般性を失うことはない。

$$\begin{aligned}
\mathrm{Var}(y_t) &= \boldsymbol{C}\mathbb{E}\left[\boldsymbol{x}_t\boldsymbol{x}_t^\top\right]\boldsymbol{C}^\top \\
\mathbb{E}\left[\boldsymbol{x}_t\boldsymbol{x}_t^\top\right] &= \mathbb{E}\left[\boldsymbol{A}\boldsymbol{x}_{t-1}\boldsymbol{x}_{t-1}^\top\boldsymbol{A}^\top + (\text{独立性によりゼロになる項}) + \boldsymbol{B}\boldsymbol{B}^\top\varepsilon_t^2\right] \\
&= \boldsymbol{A}\mathbb{E}\left[\boldsymbol{x}_{t-1}\boldsymbol{x}_{t-1}^\top\right]\boldsymbol{A}^\top + \boldsymbol{B}\boldsymbol{B}^\top\sigma^2 \\
&= (\boldsymbol{A})^2\,\mathbb{E}\left[\boldsymbol{x}_{t-2}\boldsymbol{x}_{t-2}^\top\right]\left(\boldsymbol{A}^\top\right)^2 + \left(\boldsymbol{A}\boldsymbol{B}\boldsymbol{B}^\top\boldsymbol{A}^\top + \boldsymbol{B}\boldsymbol{B}^\top\right)\sigma^2 \\
&= \cdots \\
&= (\boldsymbol{A})^n\,\mathbb{E}\left[\boldsymbol{x}_{t-n}\boldsymbol{x}_{t-n}^\top\right]\left(\boldsymbol{A}^\top\right)^n + \left[\boldsymbol{A}^n\boldsymbol{B}\boldsymbol{B}^\top\left(\boldsymbol{A}^\top\right)^n + \cdots + \boldsymbol{A}\boldsymbol{B}\boldsymbol{B}^\top\boldsymbol{A}^\top + \boldsymbol{B}\boldsymbol{B}^\top\right]\sigma^2
\end{aligned}$$

$\boldsymbol{A}$ の安定性より，$n \to \infty$ で収束して，

$$\begin{aligned}
\mathbb{E}\left[\boldsymbol{x}_t\boldsymbol{x}_t^\top\right] &= \sum_{n=0}^{\infty}\boldsymbol{A}^n\boldsymbol{B}\boldsymbol{B}^\top\left(\boldsymbol{A}^\top\right)^n\sigma^2 \equiv \boldsymbol{\Sigma}, \\
\mathrm{Var}(y_t) &\equiv \boldsymbol{C}\boldsymbol{\Sigma}\boldsymbol{C}^\top
\end{aligned}$$

$\boldsymbol{\Sigma}$ の具体的な値を求めるには，行列方程式

$$\boldsymbol{\Sigma} = \boldsymbol{A}\boldsymbol{\Sigma}\boldsymbol{A}^\top + \boldsymbol{B}\boldsymbol{B}^\top\sigma^2 \tag{4.10}$$

を $\mathbf{\Sigma}$ について解けばよい。式 (4.10) のような方程式は**離散時間リアプノフ方程式**と呼ばれている。

問題 4.3. 行列方程式 (4.10) の未知行列 $\mathbf{\Sigma}$ を

$$\mathbf{\Sigma} = \begin{bmatrix} \alpha & \beta \\ \beta & \gamma \end{bmatrix}$$

と成分表示する。方程式

$$\begin{bmatrix} \alpha & \beta \\ \beta & \gamma \end{bmatrix} = \begin{bmatrix} a_1 & a_2 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha & \beta \\ \beta & \gamma \end{bmatrix} \begin{bmatrix} a_1 & 1 \\ a_2 & 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \end{bmatrix} \sigma^2$$

を α, β, γ について解き,

$$\mathrm{Var}(y_t) = \frac{(1 - a_1^2)\sigma^2}{1 - a_2 - (a_1^2 + a_2^2) - a_2(a_1^2 - a_2^2)}$$

であることを示しなさい。

異なる時点の変数間の共分散（自己共分散）を計算しよう。これも $a_0 = 0$ とし

て一般性を失うことはない。

$$\begin{aligned}\mathbb{E}\left[\boldsymbol{x}_t \boldsymbol{x}_{t-k}^\top\right] &= \mathbb{E}\left[\left(\boldsymbol{A}^k \boldsymbol{x}_{t-k} + \sum_{j=0}^{k-1} \boldsymbol{A}^j \boldsymbol{B} \varepsilon_{t-j}\right) \boldsymbol{x}_{t-k}^\top\right] \\ &= \boldsymbol{A}^k \boldsymbol{\Sigma} + \sum_{j=0}^{k-1} \boldsymbol{A}^j \mathbb{E}\left[\boldsymbol{B} \underbrace{\varepsilon_{t-j}}_{t-j>t-k} \boldsymbol{x}_{t-k}^\top\right] \\ &= \boldsymbol{A}^k \boldsymbol{\Sigma}\end{aligned}$$

となる。最後の等式は，ホワイトノイズの仮定から得られる「過去の $\boldsymbol{x}$ と未来の ε が独立である」という性質によって成り立つ。y については

$$\mathrm{Cov}(y_t, y_{t-k}) = \boldsymbol{C}\boldsymbol{A}^k \boldsymbol{\Sigma} \boldsymbol{C}^\top$$

である。一般に，$\mathrm{Cov}(y_t, y_{t-k})$ は t, k に依存するが，弱定常 AR 過程の自己共分散は t にはよらず時間差 k のみで決定できる。自己共分散を k の関数と見て，

$$\phi(k) = \mathrm{Cov}(y_t, y_{t-k}) = \boldsymbol{C}\boldsymbol{A}^k \boldsymbol{\Sigma} \boldsymbol{C}^\top$$

を定義する。$\phi(k)$ を**自己共分散関数**という。

自己共分散が時間差のみに依存するという性質は応用上極めて重要である。$\mathrm{Cov}(y_t, y_{t-k})$ の定義に含まれる期待値は，本来，時系列の生成を何度も繰り返して計算した結果の平均（アンサンブル平均）によって推定されるべきものである。しかし，1 回だけ生成した長時間時系列の中には同じ時間差を持つデータがたくさん入っているので（y_t と y_{t-k}, y_{t+1} と y_{t-k+1}, y_{t+2} と y_{t-k+2} など），これらの実データの平均値を使うことで共分散の推定が可能になる。アンサンブル平均が長時間平均と一致するという性質は**エルゴード性**として知られている。実データを用いる時系列分析では，1 回だけ生成された時系列しか使えないことがほとんどなので，エルゴード性を持つモデルは大変都合がよいのだ。

自己相関のグラフ

$\boldsymbol{A}$ が安定固有値のみをもつ場合，自己共分散関数

$$ACF_k = \boldsymbol{C}\boldsymbol{A}^k\boldsymbol{\Sigma}\boldsymbol{C}^\top, \quad k = 0, 1, 2, \ldots$$

は $k \to \infty$ に伴って減衰（ゼロに漸近）する挙動を示す。これは弱定常 AR 過程の重要な特徴である。

さらに，偏自己相関関数（partial autocorrelation function, PACF) という重要な概念があるので，ここで紹介しておこう。AR (2) のケースでは，

$$\begin{aligned} y_t &= a_1 y_{t-1} + a_2 y_{t-2} + (*) \\ &= a_1(a_1 y_{t-2} + (*)) + a_2 y_{t-2} + (*) \end{aligned}$$

なので（議論に関係ない部分は $(*)$ としている），$y_{t-2} \longrightarrow y_t$ の影響は，y_{t-1} を経由した間接的な影響（$a_1^2 y_{t-2}$）と，y_1 を介さない直接的な影響（$a_2 y_{t-2}$）の 2 つの部分に分解できる。間接的な影響を取り除いて直接的な影響のみを取り出そうとするものが PACF である。AR 過程の場合は係数が PACF に相当するので，AR(2) では $k = 2$ のことろでするどいカットオフが生じる。

$$PACF_1 = a_1, \quad PACF_2 = a_2, \quad PACF_3 = PACF_4 = \cdots = 0$$

与えられたデータから ACF や PACF を推定 [3]，可視化することができるので，ここで説明したような定性的な性質を理解しておけば，手元の時系列データが AR 過程で表現できそうか，また，次数がどれくらいかを見積もることができる。詳細は 沖本 (2010) を参照。

4.3. プログラミング：シミュレーション

データからモデルやモデルのパラメータを決定することを**推定**という。逆に，1 つのモデルパラメータからデータを生成することを**シミュレーション**という。この節ではシミュレーションについて説明し，次節で推定について簡単に紹介する。実データを使った分析例は次節で紹介する。（予定）

```
import numpy as np
```

[3] 標本 ACF，標本 PACF は線形回帰モデルを用いて推定する。

```
import matplotlib.pyplot as plt
```

4.3.1. 時系列データの表現

2 章では，ベクトル時系列データを行列形式で表現した。つまり，T 個の d 次元ベクトル

$$\boldsymbol{x}_t = \begin{bmatrix} x_{t,1} \\ x_{t,2} \\ \vdots \\ x_{t,d} \end{bmatrix}, \qquad t = 0, 1, 2, \ldots, T-1$$

からなる時系列データを $T \times d$ 行列（2 次元配列）を用いて，

$$\begin{bmatrix} x_{0,1} & x_{0,2} & \cdots & x_{0,d} \\ x_{1,1} & x_{1,2} & \cdots & x_{1,d} \\ \vdots & \vdots & \ddots & \vdots \\ x_{t,1} & x_{t,2} & \cdots & x_{t,d} \\ \vdots & \vdots & \ddots & \vdots \\ x_{T-1,1} & x_{T-1,2} & \cdots & x_{T-1,d} \end{bmatrix}$$

のように表現した。このような表現が最も基本的であるのは間違いないのだが，**NumPy** で「t 期のデータ（行）を取り出す」という操作をするときに，取り出したベクトルには縦横の情報が設定されていない状況（1 次元配列）になってしまう。行列とベクトルの計算するときに間違いが起こりやすく，エラーの温床になってしまう。そこで，少し方針を変えて次のような愚直な理解をしよう。

ベクトル時系列 = T 個の $d \times 1$ 行列を並べたもの

つまり，シェイプが $(T, d, 1)$ であるような **NumPy** 配列（3 次元配列）を用いて T 期間 d 変数のデータを表現する。このように作った配列から t 期の情報を取り出すと，シェイプ　$(d, 1)$ の配列（2 次元配列）となるので，いつでも「列ベクトル」になっている。

実際のコードを見てみよう。配列 x は 3 期間，2 変数のデータである。ゼロで初期化している。

表 4.1.: ベクトル時系列の表現

	2 章の表現	この章の表現
全体のシェイプ	(T, d)	$(T, d, 1)$
t 期のベクトルのシェイプ	$(d,)$	$(d, 1)$

```
T, d = 3, 2
x = np.zeros((T, d, 1))
x
```

```
array([[[0.],
        [0.]],

       [[0.],
        [0.]],

       [[0.],
        [0.]]])
```

初期値を代入しよう。

```
x[0] = np.array([[1],
                 [2]])
x
```

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

       [[0.],
        [0.]],

       [[0.],
        [0.]]])
```

$t-1$ 期のデータから t 期のデータに変換する行列 A を次のように設定する。

```
A = np.array([[0.5, -0.2],
              [1.0, 0.0]])
A
```

```
array([[ 0.5, -0.2],
```

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

ここでは

$$x_t = Ax_{t-1}, \quad t = 1, 2$$

という単純な差分方程式の解を逐次的に計算してみよう。「行列 × ベクトル」の演算は Python ではアットマーク@ を用いる。

```
for t in range(1, T):
    x[t] = A @ x[t-1]

x
```

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

       [[ 0.1 ],
        [ 1.  ]],

       [[-0.15],
        [ 0.1 ]]])
```

計算が終わった後には，表形式でデータを持っておきたいと考えるかもしれない。これには シェイプを上書きするだけでよい。

```
x.shape = (T, d)
x
```

```
array([[ 1.  ,  2.  ],
       [ 0.1 ,  1.  ],
       [-0.15,  0.1 ]])
```

問題 4.4. 最後のコードで，シェイプが (T, d) である 2 次元配列を得た。これをシェイプが (T, d, 1) である 3 次元配列に戻すにはどのようにすればよいか？[4]

問題 4.5. この節のコードを 3 次元配列を使わないコードで再現しなさい。

4.3.2. 行列の安定性

安定行列に関する命題 4.1, 4.2 を確認しておこう。

[4]解答: x.shape = *x.shape, 1 とするのがクリーンだろう。右辺は * でタプルを一旦ばらして，1 を追加したタプルを作っている。こういうものだと考えておけばよい。

安定性を判別するため数値線形代数用のライブラリをインポートしておく [5]。

```
import scipy.linalg as LA
import matplotlib.pyplot as plt
```

次の行列 $\boldsymbol{A}$ の安定性を確認しよう。

```
A1 = np.array([[0.8, 0.1],
               [0.5, 1.2]])
```

E, V = LA.eig(A) というコードを実行すると，E に固有値，V に固有ベクトルを並べた行列が格納される（E, V という名前は変えてもよい）。今知りたいのは固有値の絶対値なので，np.abs(E) をチェックする。

```
E, V = LA.eig(A1)
np.abs(E)
```

```
array([0.7, 1.3])
```

1 を超えるものがあるので，A^n は発散するはずだ。確認するために各要素 $A = [a_{ij}]$ の絶対値の和 $\sum_{i,j} |a_{ij}|$ の振る舞いを可視化してみよう。（図 4.1）

```
N = 50
x = np.zeros(N)

An = np.eye(A1.shape[0])
x[0] = np.sum(np.abs(An))       # Σ|a_n|
for n in range(1, N):
    An = An @  A1
    x[n] = np.sum(np.abs(An))   # Σ|a_n|

plt.plot(x)
```

次の行列は安定固有値のみを持つようだ。

```
A2 = np.array([[0.7, -0.1],
               [0.3, 1.05]])

E, V = LA.eig(A2)
np.abs(E)
```

[5] numpy.linalg を使ってもよい。

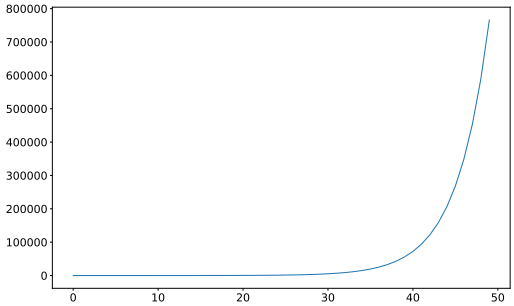


図 4.1.: 不安定固有値を持つ行列のべき

```
array([0.85, 0.9 ])
```

可視化の結果は図 4.2 のようになる。コードは省略する。

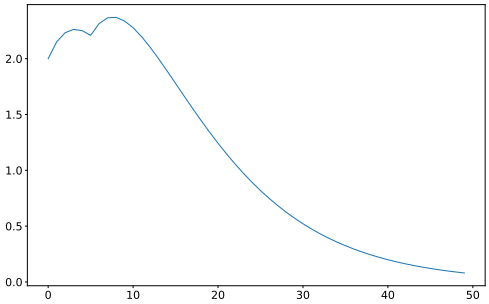


図 4.2.: 安定固有値のみを持つ行列のべき

上で定義した `A2` は命題 4.2 の公式

$$(\boldsymbol{I} - \boldsymbol{A})^{-1} = \sum_{n=0}^{\infty} \boldsymbol{A}^n$$

を満たすはずである。極限の計算はできないので，十分大きい N に対して，

$$(\boldsymbol{I} - \boldsymbol{A})\left(\sum_{n=0}^{N} \boldsymbol{A}^n\right) \approx \boldsymbol{I}$$

が成り立つことを確認しよう。総和の計算には，

$$\sum_{n=0}^{N} \boldsymbol{A}^n = \boldsymbol{I} + \boldsymbol{A}\left(\sum_{n=0}^{N-1} \boldsymbol{A}^n\right)$$

という関係を用いると便利である。

```
I = np.eye(A2.shape[0])
S = I
for n in range(1, 200):
    S = I + A2 @ S

np.allclose((I - A2) @ S, I)
```

```
True
```

4.3.3. AR 過程

AR(2) 過程

$$y_t = 0.6y_{t-1} + 0.3y_{t-2} + \varepsilon_t \tag{4.11}$$

をシミュレーションしてみよう。シミュレーションというのは，適当な初期値 y_0, y_1 と，適当に発生させたランダムな ε_t を使って y_t を計算してみることである。状態空間表現を用いるとクリーンなコードを書けるので，この表現を用いよう。

$$\begin{aligned}\boldsymbol{x}_t &= \boldsymbol{A}\boldsymbol{x}_{t-1} + \boldsymbol{B}\varepsilon_t \\ y_t &= \boldsymbol{C}\boldsymbol{x}_t,\end{aligned}$$

ただし，

$$\boldsymbol{x}_t = \begin{bmatrix} y_t \\ y_{t-1} \end{bmatrix}, \quad \boldsymbol{A} = \begin{bmatrix} 0.6 & 0.3 \\ 1 & 0 \end{bmatrix}, \quad \boldsymbol{B} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \boldsymbol{C} = \begin{bmatrix} 1 & 0 \end{bmatrix},$$

$$\mathbb{E}[\varepsilon_t] = 0.0, \quad \mathrm{Var}(\varepsilon_t) = 1.0$$

とする。初期値はゼロとしよう。係数行列と初期条件を適当に設定しさえすれば，一般の AR (p) 過程をほぼ同じコードで記述できる。

```
A = np.array([[0.6, 0.3],
              [1.0, 0.0]])
```

固有値を確認する。

```
E, V = LA.eig(A)
np.abs(E)
```

```
array([0.9244998, 0.3244998])
```

問題なさそうなので，他の行列も定義しよう。

```
B = np.array([[1.0],
              [0.0]])
C = np.array([[1.0, 0.0]])
x0 = np.array([[0.0],
               [0.0]])
```

ランダムな外乱 ε_t は正規乱数を用いて生成しよう。平均 0，分散 1 の正規乱数は次のように生成する。loc, scale, size はそれぞれ平均，標準偏差，大きさを表すパラメータである。size にタプルを渡すと，この章の時系列表現と同じような形式で乱数を生成できる。

```
rng = np.random.default_rng(1234)    # 124 は再現性のため
rng.normal(loc=0, scale=1.0, size=(3, 1, 1))
```

```
array([[[-1.60383681]],

       [[ 0.06409991]],

       [[ 0.7408913 ]]])
```

これで準備が整った。300 期間分のシミュレーションをして結果をプロットしてみよう。

```
T, d = 300, 2
x = np.empty((T, d, 1))
y = np.empty((T, 1, 1))
eps = rng.normal(loc=0, scale=1.0, size=(T, 1, 1))

x[0] = x0
y[0] = C @ x[0]

for t in range(1, T):
    x[t] = A @ x[t-1] + B @ eps[t]
    y[t] = C @ x[t]

y.shape = (T, )
y[:5]
```

```
array([0.        , 0.86374389, 3.43134556, 0.83910714,
       2.47834093])
```

作図のために 1 次元配列に変換していることに注意しよう。

```
plt.plot(y)
```

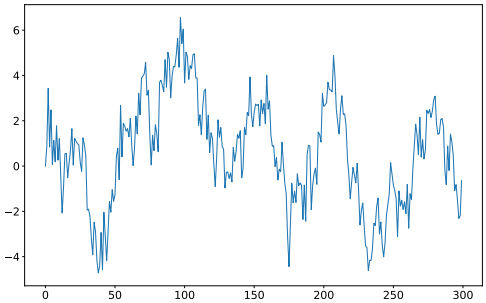


図 4.3.: AR(2) のシミュレーション

4.4. プログラミング：推定

この節では，Python を使ったデータの取得とモデルパラメータの推定を簡単に紹介する。Python による推定は **statsmodels** というライブラリを用いる。時系列分析（time series analysis）用のツール群を使うためには次のインポート文を実行する。

```
import statsmodels.tsa.api as tsa
```

データとして先程作った y を用いよう。

次のように標本自己相関を可視化できる（図 4.4）。標本自己相関は，間接的な影響も含めて，過去から現在への影響の強さを見るもので，図の右の方に進むほど「より過去」のデータが現在に与える影響を表している。図 4.4 は標本自己相関が徐々に減衰するという AR 過程の特徴を表現している。

```
tsa.graphics.plot_acf(y)
plt.show()
```

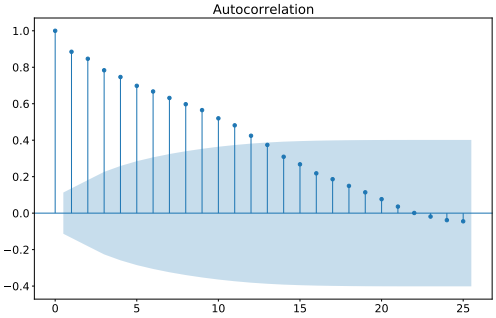


図 4.4.: 標本自己相関

標本偏自己相関は次のようにして可視化する（図 4.5）。標本偏自己相関は，間接的な影響を除いて，過去から現在への影響の強さを見るものである。図の右の方に進むほど「より過去」のデータが現在に与える影響を表しているという点では標本自己相関と同じである。図 4.5 は標本偏自己相関がするどくカットオフするという AR 過程の特徴を表現している。

```
tsa.graphics.plot_pacf(y)
plt.show()
```

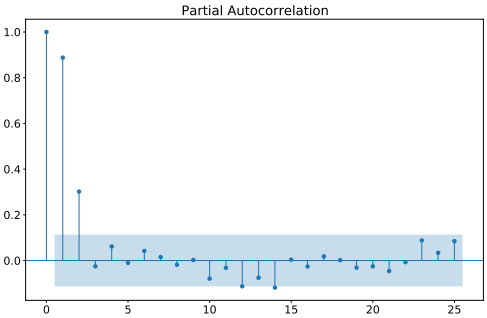


図 4.5.: 標本自己相関

推定には SARIMAX というメソッドを用いる。これは AR モデルを特殊ケースとして含む SARIMAX モデルの係数を推定するメソッドである。ちなみに，SARIMAX というのは，以下のようなコンポーネントを持つ時系列モデルである。詳しくは時系列分析の専門書を読んでほしい。

- S: Seasonal, 季節変動成分を許す
- AR: 自己回帰
- I: Integral, 和分過程
- MA: Moving Average, 移動平均
- X: eXogenous, 外生変数

データと `order=(2, 0, 0)` だけを指定すると，季節成分も移動平均成分も外生変数を持たない和分過程でない自己回帰モデル，すなわち普通の AR(2) 過程を推定する。

推定は

- モデルの設定
- モデルのフィット

という 2 段階で行われる。

```
mod = tsa.SARIMAX(y, order=(2, 0, 0))
result = mod.fit()
```

結果は summary メソッドを使って確認する。ここでは紙面の都合で params を見ることにする。

```
result.params
```

```
array([-0.00000000e+00, -0.00000000e+00,  2.60641186e+13])
```

得られた結果とデータを生成するときに使ったパラメータと見比べて，数字の意味を確認しておいてほしい。

問題 4.6. 次のコードを実行して，表示された内容を確認しなさい。パラメータはどこに書かれているか。他の数字をどのように解釈するか。

```
result.summary()
```

問題 4.7. 適当に与えたパラメータに対して AR 過程のシミュレーションを実行しなさい。シミュレーション結果に対して推定のコードを実行し，パラメータを復元できているかどうかを確認しなさい。

5. 成長と変動

5.1. 概要

この講義では以下のことを学ぶ。

- 片対数グラフ
- マクロ経済学における短期と長期
- 完全雇用，自然失業率，潜在 GDP
- 利用可能なマクロ・データ
- Pandas を使って取得する方法

理論パートの内容は，短期モデルと長期モデルを切り分けるときの考え方について説明する。次章以降に学ぶマクロ経済理論を学ぶための準備である。この章のプログラミングパートの後半部分は他の章と比べて陳腐化するのが早い。次のような理由からだ。

- Pandas のような進化の早いライブラリは使用法が比較的早く変化するため，ここで紹介したコードが標準的な方法ではなくなる，あるいは，まったく実行できなくなる可能性がある。
- データ取得は外部の Web API に依存している。API のバージョンアップによって古いコードが動かなくなる可能性がある。
- 新たなデータ提供者，外部 API のの出現により，より洗練されたデータ取得方法が生み出される。

このような問題を事前に解決することはできないので，必要な修正と知識の更新は読み手に委ねることにする。気をつけて読んでください。

5.2. 理論：数学

5.2.1. 片対数グラフ

経済時系列を図示するときには，片対数グラフがよく使われるので簡単に説明しておこう。

ある変数 $y > 0$ の変化を次のように表現する。

$$y_t = (1 + g_t) y_{t-1}$$

ここで g は y の変化率（$1 + g_t > 0$）であり，時間変化する可能性がある。対数値を取ると，

$$\begin{aligned}
\log y_t &= \log(1 + g_t) + \log y_{t-1} \\
&= \log(1 + g_t) + \log(1 + g_{t-1}) + \log y_{t-2} \\
&= \cdots \\
&= \sum_{k=1}^{t} \log(1 + g_k) + \log y_0
\end{aligned}$$

と書ける。ここで，$g_t \equiv g$ と定数であるとすれば，

$$\log y_t = t \log(1 + g) + \log y_0$$

横軸を t，縦軸を $\log y_t$ としてグラフを書くと，傾きが $\log(1 + g) \approx g$ の直線になる。実際にグラフを見てみよう。コードを短くするため，$y_0 = 1, g_t \equiv g = 0.02$ として，$y_t = (1 + 0.02)^t$ と解いてしまっている。図 5.1 のようになる。

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

t = np.arange(100)
y = (1 + 0.02) ** t
plt.plot(t, y)
plt.show()
```

対数変換したグラフは図 5.2 のようになる。

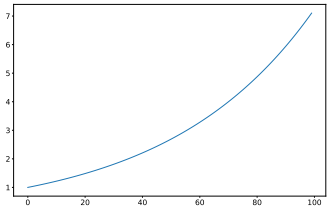


図 5.1.: y_t のグラフ

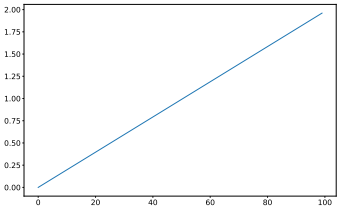


図 5.2.: $\log y_t$ のグラフ

```
plt.plot(t, np.log(y))
plt.show()
```

g が上昇すると傾きがきつくなる。では，g_t が一定ではなく上昇傾向にある場合はどうなるだろうか？変化の様子をみるために，$t < 50$ に対して $g_t = 0.02, t \geq 50$ に対して $g_t = 0.04$ であるとして作図してみよう（図 5.3）。$t = 50$ のあたりで変化率が変わっていることが分かるだろうか。

```
y2 = np.empty_like(y)
y2[:50] = (1 + 0.02) ** t[:50]
y2[50:] = y2[49] * (1 + 0.04) ** (t[50:] - 49)
plt.plot(t, y2)
plt.show()
```

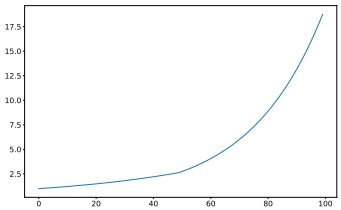


図 5.3.: y_t のグラフ（$t \geq 50$ で成長率が 0.04 に上昇）

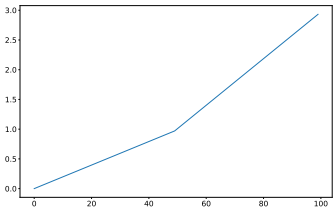


図 5.4.: $\log y_t$ のグラフ（$t \geq 50$ で成長率が 0.04 に上昇）

対数変換したグラフ図 5.4 を見ると，変化率の変化は明白である。

```
plt.plot(t, np.log(y2))
plt.show()
```

上のコードでは，対数計算を明示的に行ったが，Matplotlib でプロットするときには明示的に対数計算をする代わりにメモリ（スケール）を変更するのが普通である。次のようにする（図 5.5）。縦軸のメモリに注目せよ。

```
plt.plot(t, y2)
plt.yscale('log')
plt.show()
```

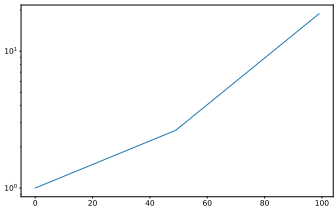


図 5.5.: y_t の片対数グラフ（$t \geq 50$ で成長率が 0.04 に上昇）

片方の軸だけが対数メモリになっているので，このようなグラフを片対数グラフという。10 を底とする対数（常用対数）を用いることが多い。その場合，縦軸メモリを 1 つ上に上るとデータが 10 倍になるようなメモリになっている。

問題 5.1. 文房具店や大学生協に行き，片対数メモリのグラフ用紙を購入し，使ってみよう。

問題 5.2. 上のコードは途中で成長率が上昇するケースを可視化した。逆に，途中で成長率が低下する場合はどのようになるか。結果を予想し，コードを書き，グラフを確認せよ。

5.2.2. 移動平均

非常に変動が激しいデータ x が手元にあるとしよう。このとき，隣り合う時点間の変化

$$\Delta x_{t+1} = x_{t+1} - x_t$$

は一時的かつ外的な要因に起因する変動まで拾ってしまって，変化の傾向を見失ってしまう可能性がある。例えば，$\bar{x}_t$ を観測できない本質的な変動要因であるとし

て（例：企業の生産性が上がるから株価が上がる），観測可能なデータ x_t はあまり本質的でない要因 ε_t （例：政治スキャンダルで株価が下がる）によって撹乱されてしまうとする。

$$x_t = \bar{x}_t + \varepsilon_t$$

ε_t と異なる時点の s の ε_s には独立性が仮定される。$\mathrm{Var}(\varepsilon_t) \equiv \sigma^2$ と仮定しよう。本来，

$$\bar{x}_{t+1} - \bar{x}_t$$

を測りたいのだけども，観測できないものは測れない。しぶしぶ，$x_{t+1} - x_t$ を使うのだが，

$$x_{t+1} - x_t = (\bar{x}_{t+1} - \bar{x}_t) + (\varepsilon_{t+1} - \varepsilon_t)$$

$$\Longrightarrow \mathrm{Var}\left((x_{t+1} - x_t) - (\bar{x}_{t+1} - \bar{x}_t)\right) = 2\sigma^2$$

となり，誤差の分散が拡大してしまう。

撹乱項の影響を除去して，時系列の変化の傾向（**トレンド**）を見るために使うのが**移動平均**である。例えば，2 期間前方移動平均は次のように定義する [1]。

$$x_t^{(2)} = \frac{x_{t-1} + x_t}{2}$$

分散にどのように影響するかを見てみよう。

$$\begin{aligned} x_{t+1}^{(2)} - x_t^{(2)} &= \frac{x_t + x_{t+1}}{2} - \frac{x_{t-1} + x_t}{2} \\ &= \frac{x_{t+1} - x_{t-1}}{2} \\ &= \frac{\bar{x}_{t+1} - \bar{x}_{t-1}}{2} + \frac{\varepsilon_{t+1} - \varepsilon_{t-1}}{2} \end{aligned}$$

$$\Longrightarrow \mathrm{Var}\left(\left(x_{t+1}^{(2)} - x_t^{(2)}\right) - \left(\frac{\bar{x}_{t+1} - \bar{x}_{t-1}}{2}\right)\right) = \frac{\sigma^2}{2}$$

移動平均を計算することで，本質的な変動要因 $\bar{x}$ の推定に関する不確実性を逓減できる。平均を取る期間の長さ（ウインドウ）を長くすれば，変動要因の影響をさ

[1]「前方」の意味，前方移動平均以外のオプションは以下のように整理できる。

- t 期の移動平均の値を t 期以前のデータのみで計算するとき，前方移動平均。
- t 期の移動平均の値を t 期以後のデータのみで計算するとき，後方移動平均。
- t 期の移動平均の値を t 期を中心としたデータで計算するとき，中心移動平均。

らに小さくすることができる。ただし，1 つ注意が必要である。移動平均が推定しているのはあくまでも $\frac{\bar{x}_{t+1}-\bar{x}_{t-1}}{2}$ であり，$x_{t+1}-x_t$ でない。トレンドが線形である場合にはこれらは一致するが，トレンドにも循環的な傾向がある場合にはその循環傾向さえも打ち消してしまう。

この性質をうまく利用すると，季節的な変動を除去することができる。例えば，消費の四半期データはボーナス月を含むか含まないかが実測値に影響しそうである。知りたいことが長期的な傾向である場合には，季節的な要因を除去するために 4 期の移動平均を計算するとよい。

問題 5.3. デパートの売上のデータを毎日取っているとする。どのような周期の循環成分が見られるだろうか。除去するためにはどのような移動平均を取ればよいか。

5.3. 理論: 成長と変動

マクロ経済学で最も重視される経済変数は実質 GDP である。実質 GDP は，経済で生産された総付加価値額から物価上昇を除去したものとして定義される。人口規模で標準化した「一人あたり実質 GDP」は，その国の個人が享受する物質的豊かさの基本指標として特に注目される。実質 GDP や一人あたり実質 GDP にはもちろん様々な原因で変化が起こるのだが，マクロ経済学では時間変化を大まかに次の 2 つのタイプに分けて分析する。

1. 経済成長。資本の蓄積，教育や技術の進歩によって生産性が向上し，経済規模が拡大する。
2. 経済変動。予期しない出来事によって経済活動が落ち込んだり，逆に，活発になること。景気循環とも言う。

5.3.1. 長期と短期

経済指標の変化を「成長」と「変動」という観点で切り分けるときには，注目している時間のスパンに意識を向けるようにしよう。通常，成長は長い時間を掛けて起

こる現象なので，長期的な視野で経済を眺めることが有効である。一方，変動に注目するときには短い時間の中での変化に関心があることが多い。

短期は長期の一部だし，短期の積み重ねが長期ではないか，と考えた人もいるかもしれない。日常言語を基準に考えれば，まさにその通りなのだけど，多くの経済理論はそのようには作られていない。長期と短期は別々の枠組みを使って考えなければならない。マクロ経済モデルにはおおまかな分類として，

- 長期のモデル
- 短期のモデル

の 2 つのタイプのモデルが存在する[2]。この 2 つのタイプを切り分けるのは，時間の長さというよりも，市場の機能や性質に由来する違いである。

- 長期のモデルでは，市場が完全であり，需給調整が円滑に機能する。
- 短期のモデルでは，市場は不完全であり，需給調整の機能が制限されている。

特に次のような分け方をすることが多い。

- 長期のモデルでは，物価が伸縮的（需給に応じて速やかに変化）である。
- 短期のモデルでは，物価が硬直的（需給に応じて変化しない）である。

なぜこのような切り分けに対して，長期・短期という言葉を使うかというと，

- 長い期間を通して見れば市場はうまく機能しているように見える，
- 短い期間だけを見ると，市場にはいろいろな不備が生じている，

ということだろう。

長期では，財・サービス市場，要素市場，金融市場などすべての市場が需給一致の状態にある。したがって，働きたいと考えている労働者はすべて雇用されている状態にあるし，遊休状態にある資本も存在しない。実際の経済はこのような状態にはないが，ベンチマークとなる時間変化を示すときに使われる便利な想定である。

短期では，市場は物価の変化によって需給を一致させることができない。したがって，一時的な不均衡状態が生じる可能性が高い。失業の拡大によって一時的な厚生の悪化が生じるので，政府介入の余地が生まれる。望ましくない景気循環に対抗するための「反循環的経済政策」の理解が短期分析の 1 つの目標になっている。景気後退を引き起こす外的な撹乱要因（ショック）が起こったときに，市場が完全

[2] もちろん，このような分類は不完全なものだ。「成長」という長期の動向を扱う分析であっても，工業化以前の経済が工業化に至る飛躍的な成長を分析するモデルと，先進国の比較的安定した成長を記述するモデルは異なっている。

に機能していれば経済主体の間の取引によって解消できたであろう問題であっても，市場が不完全であるときには大きな厚生悪化につながる可能性がある。このようなショックが，経済全体にどのように波及するか，そして，その問題にどのように対処するかというのが短期分析のテーマになっている。

5.3.2. 完全雇用 GDP・潜在 GDP

さて，ここではシンプルに「短期には失業が存在する」という一点だけに注目しよう。失業が存在するということは，その経済では使える生産要素を使い切っていないということだ。したがって，経済が生産できる最大の量を生産していない。経済が生産できる最大の量を**完全雇用 GDP** と呼ぼう。なお，後で説明する潜在 GDP のことを完全雇用 GDP と呼ぶことも多いが，ここでは区別して使っている。長期のモデルでは，失業が存在しないという大胆な仮定をおいて分析することが多い。すなわち，完全雇用 GDP の時間変化・成長を見るのが長期分析の目的である。

短期モデルでは，失業が中心的な役割を果たす。実際の経済においても，労働市場の価格（賃金）には色々な原因で硬直性が存在すると考えられていて，それが失業の 1 つの原因になっている。例えば，物価が下落したときに名目賃金が十分下がらなければ，実質賃金は上昇する。実質賃金に反応する労働需要と労働供給は，超過供給の状態になる。これは働きたい人が働けない状態，つまり失業，が生じる。名目賃金の硬直性について，大抵のマクロ経済学の教科書には次の仮説を提示して説明している。

- インサイダー・アウトサイダー理論
- 効率賃金仮説

いずれも，賃金が市場を清算させる水準よりも高い水準に留まってしまう原因に関する仮説である。各自，標準的なマクロ経済学の教科書を読んで概要を調べておいてほしい。

また，市場が清算する水準に賃金が設定されていたとしても労働市場には失業が存在するという考え方もある。労働市場で取引されている財（労働力）は，食品や工業製品などのように均質ではないので，適材適所な組み合わせ（マッチング）を見つけるためにどうしても時間や費用がかかってしまうからだ。したがって，市場均衡価格がついているにも関わらず失業が存在する。企業が労働者に望む能力や技能と，求職者が持っている能力との差によって生じる失業を**構造的失業**，職探し

のためにかかる時間が原因で起こる失業を**摩擦的失業**と呼ぶ[3]。構造的失業と摩擦的失業は切り分けることが難しいので，摩擦的・構造的失業というようにセットで扱われることが多い。

問題 5.4. 失業者に対して支払われる失業給付金は摩擦的・構造的失業を意図せず高めてしまう可能性がある。理由を検討しなさい。

摩擦的・構造的失業だけが残っているときの失業率を**自然失業率**という。自然失業率を離職と就職が均衡している状態の失業率と解釈することもできて，長期的には自然失業率の水準で失業が推移すると考えられている。言うまでもなく，現実の経済の失業率は自然失業率より高いときもあれば低いときもある。この差，特に失業の上乗せ部分のことを**循環的失業**という。摩擦的・構造的失業は労働市場の制度，慣行や時代背景によって変わるものなので，短期的な政策目標とはなりにくい。一方，循環的失業は景気後退によって生じた失業なので，政策介入によって解消することができるかもしれない。

自然失業率を達成している経済で生産されている量は**潜在 GDP**（あるいは完全雇用 GDP）と呼ばれる。その定義から，経済のベンチマークとなる生産量としてふさわしいことが分かるだろう。もちろん，潜在 GDP を直接観測できる訳ではないので，各国の中央銀行や政府機関が推計している。図 5.6 はアメリカの四半期実質 GDP（Real GDP）と潜在 GDP（Potential GDP）の推計値をプロットしたものである。縦軸を対数スケールにしているので，経済成長率は徐々に低下しつつも比較的順調な成長経路を辿っていることが分かる。長期のモデルによって行う経済成長の分析は，このような経済の右肩上がりの拡大を対象としている[4]。

[3] 本書では学部の標準的なマクロ経済学の教科書として『マンキュー マクロ経済学 I, II』を推奨しているが，ここで用いている失業に関する用語法はマンキューと異なっていることに注意しておく。マンキューの教科書では「実質賃金の硬直性によって生じる需給の不一致が原因の失業」を構造的失業と定義し，マッチングの違いによって生じる失業を摩擦的失業と定義している。

[4] 完全雇用 GDP と潜在 GDP の違いを無視しているが，これらは非常に強く相関しているので大きな問題にはならないだろう。

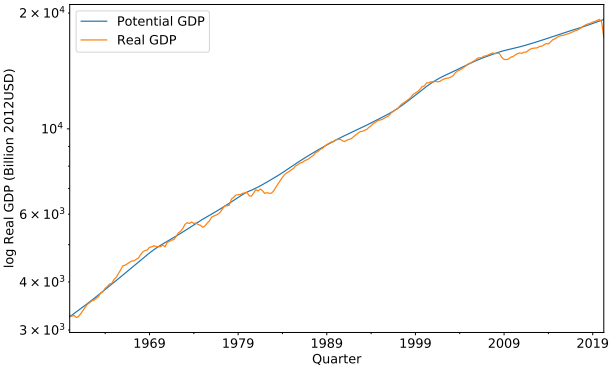


図 5.6.: アメリカの実質 GDP と潜在 GDP（Source: FRED）

一方，実質 GDP と潜在 GDP の差に注目すると，いくつかの時期で大きな差が表れている。差を明確にするための変数を導入しよう。実質 GDP と潜在 GDP の差を **GDP ギャップ**という。下の図 5.7 は GDP ギャップを潜在 GDP で除した値（GDP ギャップ率）

$$\text{GDP ギャップ率} = \frac{\text{実質 GDP} - \text{潜在 GDP}}{\text{潜在 GDP}}$$

をプロットしたグラフである。データは図 5.6 と同じものを使っている。正の値を取っている時期には経済のベンチマークとなるパフォーマンス（＝潜在 GDP）よりも大きな生産を行っているので，好景気（景気拡張期）を表している。逆に，負の値を取っている時期には経済は本来生産できる量よりも少ない量しか生産していない。したがって，この時期（色付き部分）には景気後退を経験している [5]。経済変動と経済変動に伴う厚生低下の是正に関する分析はマクロ経済の短期モデルの仕事である。

失業に関するデータも見ておこう。図 5.8 は同じ時期の失業率と自然失業率（natural rate of unemployment) の推計値をプロットしたものである。色付き部分は図 5.7 と同様，GDP ギャップが負になる領域を表している。失業率が自然失業率

[5] ここでは，GDP ギャップの符号のみで景況判断をしているが，景気後退の終了前には「GDP ギャップは負だが景気はよい」という状況があり得る。したがって，実際の景況判断は実質 GDP の山と谷を見極めるという仕事になる。アメリカの景気の転換点は NBER（全米経済研究所）が発表している。

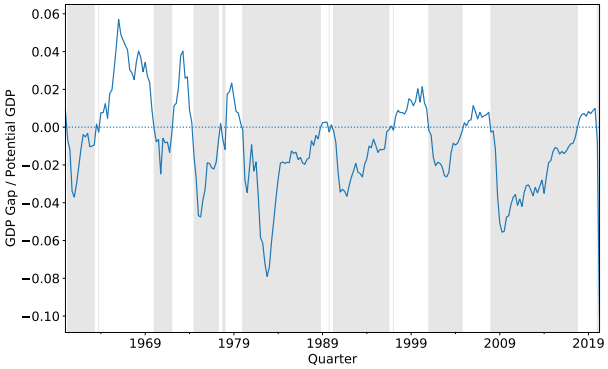


図 5.7.: GDP ギャップ（Source: FRED）

の水準を超えて高くなっている時期と，GDP ギャップが負になる時期がおおよそ同じであることに注意をしてほしい。もちろんこれは驚くようなことではなく，潜在 GDP の定義に由来する。

5.3.3. フィリップス曲線

次にインフレーションと失業の関係を見てみよう。図 5.9 は PCE デフレーター（個人消費デフレーター）から計算したインフレ率の 12 期中心移動平均をプロットしたものである。色付き部分は図 5.7 と同様，GDP ギャップが負になる領域を表している。**色付き部分がインフレ率が減少している時期におおむね重なっている**ことに注意してほしい。つまり，

- GDP ギャップが正になるときには，インフレ率は上昇する傾向にある。
- GDP ギャップが負になるときには，インフレ率は低下する傾向にある。

つまり，

- GDP ギャップとインフレ率との間には正の相関がある。

正の GDP ギャップを**インフレギャップ**，負の GDP ギャップを**デフレギャップ**と呼ぶ。GDP ギャップが正であるということは，その時期には基準となる水準より大きな生産活動が行われている。企業にとっては新しく雇用をして生産を拡大した

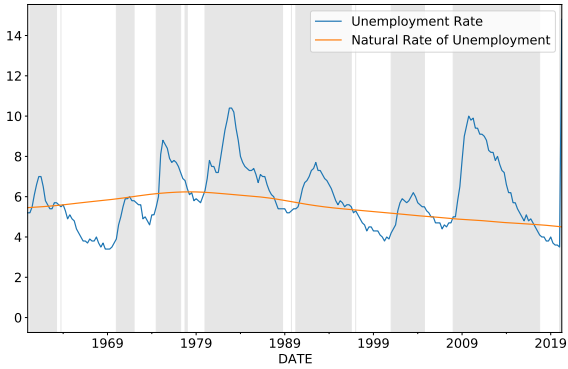


図 5.8.: GDP ギャップ（Source: FRED）

い局面ではあるが，労働市場は逼迫しているから，賃金に上昇圧力が加わるだろう。消費者の所得も増えているので，上昇した雇用コストを生産物価格に転嫁したい企業は容易にそのようにできる。結果的にインフレ率は当初の予想水準よりも高めになる。逆に，GDP ギャップが負であるときには，企業は生産活動を抑えている。労働需要が小さく，賃金には下降圧力が加わる。所得は小さいので，生産物価格を上げにくい環境になっているし，企業はその必要性も感じていない。結果として，インフレ率は当初の予想水準よりも低くなる。このインフレ率に関する「当初の予想水準」は，**予想インフレ率**という。

長期的には市場は理想的な機能を果たし，情報の非対称性も解消される。長期的な経済の動向は次のようになる。

- 長期的には自然失業率が達成される。
- 長期的には潜在 GDP が生産される。
- 長期的には予想インフレ率は実際のインフレ率と一致する。

t 期のインフレ率を π_t, $(t-1)$ 期に形成された t 期のインフレ率に関する予想を $\mathbb{E}_{t-1}\pi_t$，実質 GDP を Y_t, 潜在 GDP を $\bar{Y}_t$ とすると，前述の関係は次のように表現できる。

$$\pi_t - \mathbb{E}_{t-1}\pi_t = \alpha\left(Y_t - \bar{Y}_t\right), \qquad \alpha > 0$$

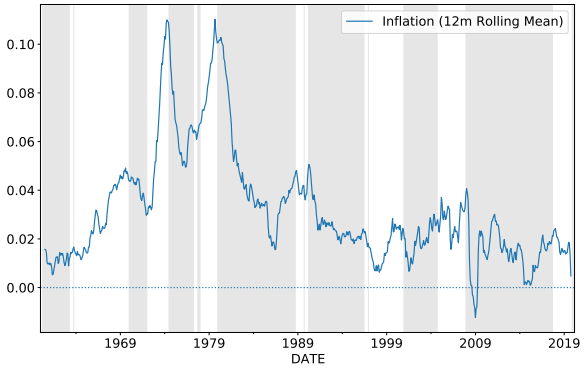


図 5.9.: GDP ギャップとインフレーション（Source: FRED）

この関係式は（修正）**フィリップス曲線**である。賃金上昇率と所得の関係を表す。マクロ経済の物価水準と供給水準の関係を決定する方程式で，AS 曲線（Aggregate Supply）とも呼ばれている。

5.4. プログラミング: 準備

5.4.1. Conda による追加ライブラリのインストール

この章のプログラムでは，**pandas-datareader** というライブラリを用いる。インスールされていない可能性があるので，Anaconda プロンプトで次のコマンドを実行してインスールしておく。

```
conda install pandas-datareader
```

IPython からこのコマンドを実行するには次のようにする。

```
!conda install pandas-datareader
```

5.4.2. Python の基本型

文字列

すでに何度も使っているが「**文字列型**」について確認しておこう。文字列は文字の並びである。シングルクォーテーションかダブルクオーテーションのペアで囲むことで作成する。

```
s1 = "Hello, world."
s1
```

```
'Hello, world.'
```

日本語の文字も登録できる。

```
s2 = 'こんにちは'
s2
```

```
'こんにちは'
```

リストと同じ記法で文字列の一部を取り出すことができる。

```
s1[0]
```

```
'H'
```

```
s1[1:3]
```

```
'el'
```

```
s1[-1]
```

```
'.'
```

`list()` 関数を使うと明示的にリストに変換できる。

```
list(s2)
```

```
['こ', 'ん', 'に', 'ち', 'は']
```

このような変換が必要なときに自動的に行われていると考えてよい。文字列の中に含まれる各文字を走査する `for` ループを書くこともできる。

```
for i, s in enumerate(s2):
```

```
    print(s * (i + 1))
```

```
こ
んん
にに
ちちち
ははははは
```

問題 5.5. `enumerate()` の使い方を調べて説明しなさい。

f-string という便利な記法がある。f-string を使えば文字列定義の中で式を使うことができる。次の使用例を見れば使い方は明らかだろう。クォーテーションマークの前の `f` と中の波括弧部分に注目する。

```
x, y = 10, 20
f"{x} + {y} = {x + y}"
```

```
'10 + 20 = 30'
```

文字列フォーマット用の記法を組み合わせると，小数点以下の表示桁数や，右寄せ左寄せなどのコントロールが可能になる [6]。

```
f"{x} / ({x} + {y}) ≒ {x / (x + y):.2f}"
```

```
'10 / (10 + 20) ≒ 0.33'
```

辞書

辞書（dictionary）というデータ構造が非常によく用いられるので覚えておこう。普通の辞書は「見出し語」と「定義」の多数の組合せから構成されている。Python の辞書も同じような構成になっている。「キー」と「値」のペアが辞書である。次

[6] 詳細は https://docs.python.org/3/library/string.html#format-string-syntax を参照。

の辞書 d には 3 つのキー, 1, 'a', 'x', が登録されている。対応する値はそれぞれ, 100, 200, [1, 2] である。

```
d = {1: 100, 'a': 200, 'x': [1, 2]}
d
```

```
{1: 100, 'a': 200, 'x': [1, 2]}
```

値はどのような型のオブジェクトでも割り当てることができる。一方，キーの方には制約があって,「不変（immutable）」なオブジェクトのみを割り当てることができる。例えば，整数，浮動小数点数，文字列やタプルは不変なオブジェクトである。リストは不変なオブジェクトではないので，キーにすることはできない。

問題 5.6. 「リスト」をキーとする辞書を定義しようとすると，どのようなエラーが発生するか。実験して結果を記録しなさい。

角括弧の中にキーを入れると値を読み出せる。キーは変数に格納していても問題ない。

```
d[1]
```

```
100
```

```
k = 'a'
d[k]
```

```
200
```

辞書をループするとすべての「キー」を走査する。この例では登録順に表示されているが，これは必ずしも保証されていない [7]。

```
for key in d:
```

[7] 順序が意味をもつ場合には OrderedDict 型を用いる。詳細は https://docs.python.org/3/library/collections.html#collections.OrderedDict

```
    print(key)
```

```
1
a
x
```

キーと値の組合せについてループしたいときには，items() メソッドを使うとよい。

```
for k, v in d.items():
    print(f"{k} : {v}")
```

```
1 : 100
a : 200
x : [1, 2]
```

5.4.3. Pandas 入門

Python で表形式のデータを扱うときは **pandas** というライブラリを用いるのが標準的である。**NumPy** の配列をベースに設計されている。配列との主要な違いは，

- 異なるデータ型を持つ列の混在が許される。
- 行・列に意味のあるラベル（インデックス）を付けることができる。
- その他データの加工・可視化を便利にする機能が追加されている。

4×3 行列をもとに **pandas** の DataFrame オブジェクトを作ってみよう。インデックスを付けずに作る場合は次のようになる。

```
import pandas as pd
x = np.arange(12.0).reshape(4, 3)
pd.DataFrame(x)
```

```
      0     1     2
0   0.0   1.0   2.0
1   3.0   4.0   5.0
2   6.0   7.0   8.0
3   9.0  10.0  11.0
```

多くの場合，次のようにインデックスが付けられている。

```
frame = pd.DataFrame(x, columns=['a', 'b', 'c'],
                     index=pd.PeriodIndex(range(2000, 2004), freq='A'))
```

```
frame
```

```
          a      b      c
2000   0.0    1.0    2.0
2001   3.0    4.0    5.0
2002   6.0    7.0    8.0
2003   9.0   10.0   11.0
```

DataFrame の定義にはリスト様のオブジェクトを値に持つ辞書を用いることもできる。

```
pd.DataFrame({'a': [0.0, 3.0, 6.0, 9.0],
              'b': [1.0, 4.0, 7.0, 10.0],
              'id': list('xyzw')},
              index=range(2000, 2004))
```

```
          a      b id
2000   0.0    1.0  x
2001   3.0    4.0  y
2002   6.0    7.0  z
2003   9.0   10.0  w
```

pandas の DataFrame オブジェクトには時系列データを扱うための便利なメソッドが定義されている。差分を計算する diff() や変化率を計算する pct_change() はその一例である。これらのメソッドを使うを 1 行目のデータが欠測値（NA）となるので，dropna() メソッドをつなげて削除することが多い。

```
frame.diff()
```

```
          a     b     c
2000   NaN   NaN   NaN
2001   3.0   3.0   3.0
2002   3.0   3.0   3.0
2003   3.0   3.0   3.0
```

```
frame.pct_change().dropna()
```

```
          a         b       c
2001   inf  3.000000   1.500
2002   1.0  0.750000   0.600
2003   0.5  0.428571   0.375
```

DataFrame オブジェクトの列を取得するのにはいくつかの方法があるが，列名を指定して呼び出す 2 つの方法を確実に覚えておこう。単一の列をけ呼び出す場合には表示の見た目が変わっていることに注意しよう。Series オブジェクトと呼ばれるオブジェクトになっている。要するに，DataFrame は Series を列方向に並べたものだ。これらの違いは，**NumPy** の 2 次元配列と 1 次元配列の違いと考えておけばよい。

```
frame.a
```

```
2000    0.0
2001    3.0
2002    6.0
2003    9.0
Freq: A-DEC, Name: a, dtype: float64
```

```
frame['b']
```

```
2000     1.0
2001     4.0
2002     7.0
2003    10.0
Freq: A-DEC, Name: b, dtype: float64
```

```
frame[['a', 'c']]
```

```
         a     c
2000  0.0   2.0
2001  3.0   5.0
2002  6.0   8.0
2003  9.0  11.0
```

行を取得するには次のようにインデックスのスライス記法を用いる。

```
frame.loc['2000':'2002', :]
```

```
         a    b    c
2000  0.0  1.0  2.0
2001  3.0  4.0  5.0
2002  6.0  7.0  8.0
```

pandas を使えば列を追加するのも簡単だ。

```
frame['d'] = frame.b * frame.c + 10
frame
```

```
          a      b      c       d
2000   0.0    1.0    2.0    12.0
2001   3.0    4.0    5.0    30.0
2002   6.0    7.0    8.0    66.0
2003   9.0   10.0   11.0   120.0
```

データを **pandas** の DataFrame にしておくと簡便な記法で可視化ができる。基本の plot() メソッドを使うと折れ線グラフが描ける（図 5.10）。

```
frame.plot(y = ['a', 'd'])
plt.show()
```

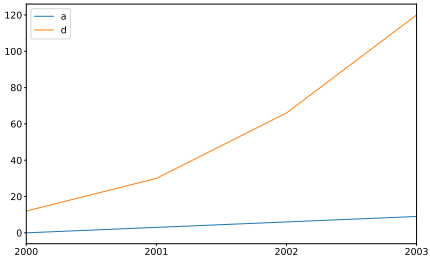


図 5.10.: **pandas** のプロット

散布図を描くには plot.scatter() を使う。図 5.11 はマーカーのサイズをデータ依存にさせる，いわゆるバブルチャートである。

```
frame.plot.scatter(x='a', y='d', s=10 * frame.b ** 2)
plt.show()
```

plot.bar(), plot.barh() を使うと棒グラフを描写できる。 結果はそれぞれ図 5.12 と図 5.13 のようになる。stacked=True とすれば積み上げ棒グラフになる。水平方向にグラフを伸ばす方法も紹介しておこう。

```
frame.plot.bar()
plt.show()
```

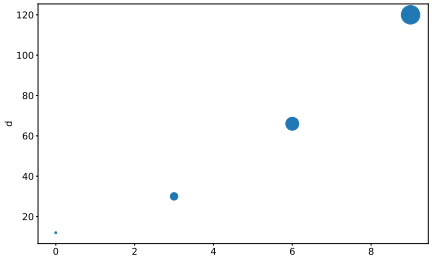


図 5.11.: **pandas** の散布図

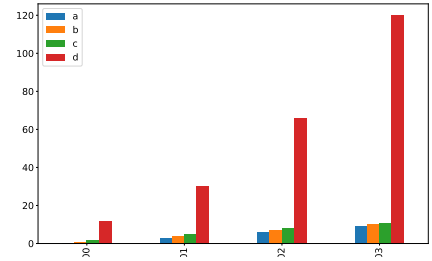


図 5.12.: **pandas** の棒グラフ

```
ax = frame.plot.barh(stacked=True)
ax.invert_yaxis()
plt.show()
```

5.5. プログラミング: データを眺める

さて，本章プログラミングパートの本題に入ろう。

「オープンデータ」という標語のもとに多くの公的組織・私的組織が広く利用可能なデータをインターネット上で公開している。ウェブサイトに Excel ファイルや CSV ファイルを置いているだけのものから，機械可読が容易な形式で配布されているもの，プログラム上で取得ができるような特別な配慮がなされたものまで様々である。

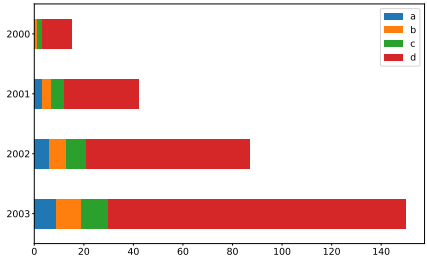


図 5.13.: **pandas** の積み上げ棒グラフプロット

5.5.1. API

一般にソフトウェア間で通信を行う方式のことを API (Application Programming Interface) と呼ぶ。データ提供者が API（Web API）を提供している場合，プログラマは API の規則に適合したコードを書くことで，プログラム的にデータを取得することができる。API には登録不要で利用できるものから，事前に利用者情報を登録した上で API キーや API トークンといった認証情報を取得しておく必要があるものもある。プログラマは必要な認証情報を取得した後，データ提供者のウェブサーバーに API リクエストを送信する。API リクエストは HTTP (HyperText Transfer Protocol) という方式で行われる。大雑把に言えば，ウェブブラウザのアドレスバーにアドレスを打ち込むようなイメージである。ただし，ブラウザは使わない（使ってもいいけど）。API リクエスト用の「アドレス」には必要なデータは何かを説明するためのテキスト（「クエリ」）を付加する。データ提供者のサーバーがデータ要求を受け取ると，必要なデータをデータベースサーバーから取得する。API リクエストのクエリを使用しているデータベースに対するクエリに変換し，出力結果を適切にフォーマットしてプログラマに送り返す。プログラマは XML や JSON, CSV といった形式でフォーマットされたテキスト情報としてデータを受け取る [8]。

すべてのデータ提供者が同じ形式でデータを提供していれば話は簡単なのだが，実際にはデータ提供者ごとに

[8] XML (eXtensible Markup Language) は HTML (HyperText Markup Language; ウェブページを書くときの記法）に似た記法でデータとメタデータを記録する方法。JSON (JavaScript Object Notation) は JavaScript (ウェブページの動的処理で用いられるプログラミング言語) のオブジェクト記法を用いたデータ記法。Python の辞書のような書き方をする。CSV (Comma Separated Values) は表形式のデータをコンマと改行で区切るだけのシンプルな表現である。

- クエリ記法が違う,
- 出力データのフォーマット方法が違う

という問題がある。個別の API のドキュメントを読んで使用方法を調べるというのがデータ利用の最初の難関である。かなり早い段階で難関がやってくるので，ここで諦めてしまう人も多いだろう。

5.5.2. pandas-datareader

データ提供者ごとに API の規則が異なっていることがデータ利用を難しくしているのであれば，API の差異を吸収して共通の記法でデータを取得できるようにすれば，データ活用の利便性は格段に向上するだろう。**pandas-datareader** というライブラリはこのような目的で作られている。プログラマが直接 API リクエストを発行する代わりに，**pandas-datareader** に対して，標準化された記法でリクエストを送る。**pandas-datareader** はデータ提供者ごとのクエリ記法に変換し，データの取得を行う。取得されたデータはデータ提供者が使用しているフォーマット方式から **pandas** のデータフレームに変換する。

pandas-datareader を使うとインターネット上で公開されているデータをダウンロードして **pandas** のデータフレーム形式に変換する作業が簡単になる。利用可能なデータは公式の安定ドキュメントで確認しておいてほしい。

- `https://pydata.github.io/pandas-datareader/remote_data.html`

インポートして実際に使ってみよう。

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import pandas_datareader as pdr
```

FRED (St. Louis FED)

FRED はセントルイス連邦準備銀行が提供する経済時系列データのデータベースである。アメリカのデータだけでなく世界各国の情報も手に入れることができる。ウェブサイト `https://fred.stlouisfed.org/` にもデータのダウンロードや可視化のツールが一通り揃っているので，ぜひ試してほしい。

さて，ウェブサイトの検索窓で「GDP」と検索すると，1 つ目の結果は「Real Gross Domestic Product」となっている。実質 GDP のことである。これを開くと図 5.14 のような見出しと，グラフが表示されるはずだ。Python で処理するために必要な情報は図の赤枠で囲った部分の小さなコード「GDPC1」である。同じようにして潜在 GDP「Potential GDP」も検索してコードを確認しておこう。

図 5.14.: FRED のウェブサイトでデータコードを確認する

次のコードが使用例である。pdr.get_data_fred() で FRED のデータを取得できる [9]。1 つ目の引数としてリスト ['GDPPOT', 'GDPC1'] を書いて，ダウンロード対象が GDPPOT（潜在 GDP）と GDPC1（実質 GDP）であることを指定する。結果に gdp という名前を付けた。これは **pandas** のデータフレームになっている。

```
gdp = pdr.get_data_fred(['GDPPOT', 'GDPC1'],
                        start='1960-01-01', end='2020-04-01')
gdp
```

```
                GDPPOT       GDPC1
DATE
1960-01-01   3247.590212   3275.757
1960-04-01   3280.322655   3258.088
1960-07-01   3313.239234   3274.029
1960-10-01   3345.865071   3232.009
1961-01-01   3378.938041   3253.826
...                  ...        ...
```

[9] 多くのデータソースに対して pdr.get_data_*() という関数が用意されている。

```
2019-04-01   18885.480000   19020.599
2019-07-01   18976.490000   19141.744
2019-10-01   19065.580000   19253.959
2020-01-01   19153.980000   19010.848
2020-04-01   19242.040000   17302.511

[242 rows x 2 columns]
```

データフレームに対して実行できる処理はすぐに使える。例えば，図を描くのも簡単だ。

```
gdp[['GDPPOT', 'GDPC1']].plot()
plt.show()
```

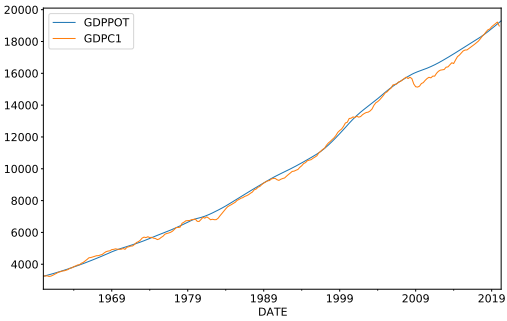


図 5.15.: FRED で取得したデータ

問題 5.7. GDP ギャップと GDP ギャップ率を計算して，gdp データフレームに新しい列 (gap, rgap) として追加しなさい。結果は次のようになる。

```
                GDPPOT       GDPC1          gap       rgap
DATE
1960-01-01   3247.590212   3275.757    28.166788  0.008673
1960-04-01   3280.322655   3258.088   -22.234655 -0.006778
1960-07-01   3313.239234   3274.029   -39.210234 -0.011834
1960-10-01   3345.865071   3232.009  -113.856071 -0.034029
1961-01-01   3378.938041   3253.826  -125.112041 -0.037027
...                  ...        ...          ...       ...
```

```
2019-04-01  18885.480000  19020.599   135.119000  0.007155
2019-07-01  18976.490000  19141.744   165.254000  0.008708
2019-10-01  19065.580000  19253.959   188.379000  0.009881
2020-01-01  19153.980000  19010.848  -143.132000 -0.007473
2020-04-01  19242.040000  17302.511 -1939.529000 -0.100796

[242 rows x 4 columns]
```

問題 5.8. GDP ギャップ率をプロットしなさい。

次に，インフレ率を見るために Consumer Price Index for All Urban Consumers: All Items in U.S. City Average (CPIAUCSL) を取得してみよう。これは消費者物価指数の月次データである。

```
price = pdr.get_data_fred('CPIAUCSL',
                          start='1960-01-01', end='2020-04-01')
price
```

```
            CPIAUCSL
DATE
1960-01-01    29.370
1960-02-01    29.410
1960-03-01    29.410
1960-04-01    29.540
1960-05-01    29.570
...               ...
2019-12-01   258.203
2020-01-01   258.687
2020-02-01   258.824
2020-03-01   257.989
2020-04-01   256.192

[724 rows x 1 columns]
```

インフレ率を見るために pct_change() メソッドを用いる。月次のデータなので 12 倍して年率換算しておこう。

```
price['inflation'] = price.CPIAUCSL.pct_change() * 12
price.inflation.plot()
plt.show()
```

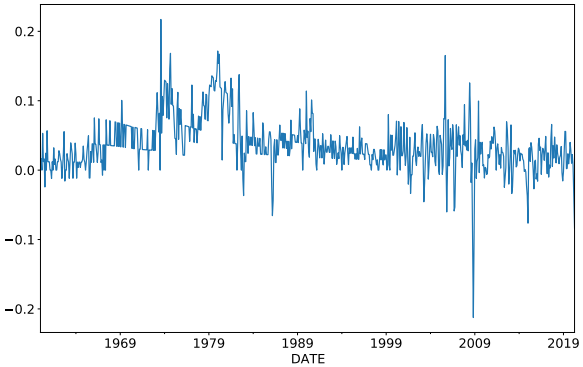


図 5.16.: CPIAUCSL (Source: FRED)

変動が激しいので移動平均を計算してみよう。rolling() メソッドでグループ化して，グループごとに平均を取るとよい。

```
price['ma3'] = price.inflation.rolling(3, center=True).mean()
price['ma9'] = price.inflation.rolling(9, center=True).mean()
price
```

```
            CPIAUCSL   inflation        ma3        ma9
DATE
1960-01-01    29.370         NaN        NaN        NaN
1960-02-01    29.410    0.016343        NaN        NaN
1960-03-01    29.410    0.000000   0.023129        NaN
1960-04-01    29.540    0.053043   0.021743        NaN
1960-05-01    29.570    0.012187   0.027154        NaN
...              ...         ...        ...        ...
2019-12-01   258.203    0.009954   0.020172   0.001446
2020-01-01   258.687    0.022494   0.012934        NaN
2020-02-01   258.824    0.006355  -0.003288        NaN
2020-03-01   257.989   -0.038714  -0.038648        NaN
2020-04-01   256.192   -0.083585        NaN        NaN

[724 rows x 4 columns]
```

最後にプロットしてみよう。移動平均を取る期間（ウインドウ）が長くなるほど，なめらかなグラフになっていることを確認してほしい。

```
ax = price.inflation.plot(alpha=0.2)
price[['ma3', 'ma9']].plot(ax = ax)
plt.show()
```

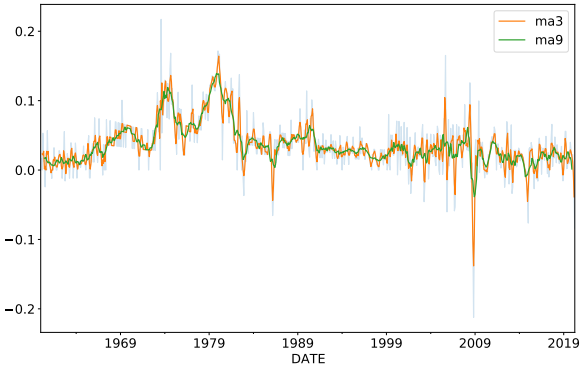


図 5.17.: CPIAUCSL の移動平均 (Source: FRED)

World Bank

国際比較をしたいときには，世界銀行のパネルデータを用いることができる。1960 年から 2010 年までの一人あたりの名目 GDP のデータ（米ドル換算）をダウンロードするコードは以下のように書ける。FRED とはかなり使い方が異なっているので注意しよう。`from pandas_datareader import wb` というコードで世界銀行のデータベースにアクセスするための特別なモジュール `wb` を使えるようにしている。

```
from pandas_datareader import wb
gdp = wb.download(indicator='NY.GDP.PCAP.CD', country='all',
                  start=1960, end=2010)
gdp
```

NY.GDP.PCAP.CD

```
country     year
Arab World  2010      5926.712963
            2009      5180.581491
            2008      6149.511489
            2007      4963.224710
            2006      4360.335052
...                           ...
Zimbabwe    1964       281.558440
            1963       277.479715
            1962       276.688781
            1961       280.828951
            1960       278.813699

[13464 rows x 1 columns]
```

いわゆる「縦持ち」のデータになっているので国ごとに列を分けておこう。**pandas** を使ったデータ加工の詳細は McKinney (2018) などを参考にしてほしい。あとで可視化をするので，結果が見やすくなるように対数変換もしている。

```
gdp_pivot = gdp.reset_index()
gdp_pivot['NY.GDP.PCAP.CD'] = np.log(gdp_pivot['NY.GDP.PCAP.CD'])
```

```
Error in py_call_impl(callable, dots$args, dots$keywords): KeyError: 'NY

Detailed traceback:
  File "<string>", line 1, in <module>
  File "/Users/kenjisato/.pyenv/versions/anaconda3-2020.11/lib/python3.8
    indexer = self.columns.get_loc(key)
  File "/Users/kenjisato/.pyenv/versions/anaconda3-2020.11/lib/python3.8
    raise KeyError(key) from err
```

```
gdp_pivot = gdp_pivot.pivot(index='year', columns='country',
                            values='NY.GDP.PCAP.CD')
```

```
Error in py_call_impl(callable, dots$args, dots$keywords): KeyError: 'ye

Detailed traceback:
  File "<string>", line 1, in <module>
```

```
  File "/Users/kenjisato/.pyenv/versions/anaconda3-2020.11/lib/py
    return pivot(self, index=index, columns=columns, values=value
  File "/Users/kenjisato/.pyenv/versions/anaconda3-2020.11/lib/py
    index = [data[idx] for idx in index]
  File "/Users/kenjisato/.pyenv/versions/anaconda3-2020.11/lib/py
    index = [data[idx] for idx in index]
  File "/Users/kenjisato/.pyenv/versions/anaconda3-2020.11/lib/py
    indexer = self.columns.get_loc(key)
  File "/Users/kenjisato/.pyenv/versions/anaconda3-2020.11/lib/py
    raise KeyError(key) from err
```

```
gdp_pivot.index = gdp_pivot.index.astype('uint64')
```

出来上がった表は非常に大きいので出力は省略する。手元のコンピュータで確認してほしい。

```
gdp_pivot
```

4 年分のデータの密度プロットを表示する。この図はヒストグラムをなめらかにしたもので，横軸は 1 人あたり名目 GDP の対数値，縦軸はその GDP 水準に所属する国の頻度を表している。山が高いほどその水準の GDP を獲得する国が多い。

```
years = [1960, 1980, 2000, 2010]
fig, ax = plt.subplots(1)
for year in years:
    gdp_pivot.loc[year].dropna().plot.density(ax=ax)
```

```
Error in py_call_impl(callable, dots$args, dots$keywords): KeyErr

Detailed traceback:
  File "<string>", line 2, in <module>
  File "/Users/kenjisato/.pyenv/versions/anaconda3-2020.11/lib/py
    return self._getitem_axis(maybe_callable, axis=axis)
  File "/Users/kenjisato/.pyenv/versions/anaconda3-2020.11/lib/py
    return self._get_label(key, axis=axis)
  File "/Users/kenjisato/.pyenv/versions/anaconda3-2020.11/lib/py
```

```
    return self.obj.xs(label, axis=axis)
  File "/Users/kenjisato/.pyenv/versions/anaconda3-2020.11/lib/python3.8
    loc = self.index.get_loc(key)
  File "/Users/kenjisato/.pyenv/versions/anaconda3-2020.11/lib/python3.8
    raise KeyError(key) from err
```

```
ax.legend(years)
ax.set_xlabel('Log GDP per capita')
plt.show()
```

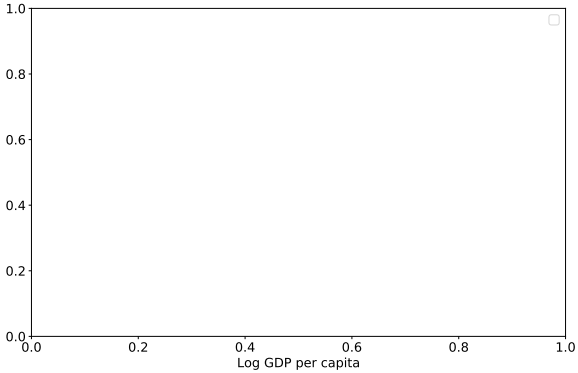


図 5.18.: CPIAUCSL の移動平均 (Source: FRED)

グラフから次の傾向が読み取れることを見てほしい。

- 密度のグラフが右方向に移動している。つまり，平均的に見れば時代の経過とともに，多くの 1 人あたり GDP を生産するようになる。これは経済成長を意味している。
- 密度のグラフが徐々に広がってきているように見える。これは，上位の国と下位の国の間での所得格差（国内の格差ではないことに注意）が広がっていることを意味している。ただし，この図だけでは格差についての正確な議論はできない。

問題 5.9. 1 人あたり名目 GDP をダウンロードするときに NY.GDP.PCAP.CD と

いうコードを使った。World Bank Open Data `https://data.worldbank.org/` のウェブサイトを閲覧して，このデータを検索しなさい。ページのどの場所にコードが書かれているか。検索ボックスから他の指標についても調べてコードを探し，**pandas-datareader** でダウンロードしてみなさい。

6. 成長会計とソロー・モデル

6.1. 概要

この講義では以下のことを学ぶ。

- 生産関数と成長会計
- ソロー・モデル

プログラミングパートでは関数定義について学び，ソロー・モデルのシミュレーションを行う。

6.2. 理論: 成長会計

6.2.1. 成長率公式

第 1 章において成長率（変化率）の定義と次の近似公式について説明した。

$$\frac{y_t - y_{t-1}}{y_{t-1}} \approx \log y_t - \log y_{t-1}$$

左辺で定義される成長率は取り扱いが難しいので，この節では右辺で定義される成長率を使う [1]。第 1 章においては「瞬時変化率」と読んで詳細な説明を省略したものだ。

瞬時変化率（以下，単に成長率あるいは変化率）には便利な公式がある。y の t 期の前期比成長率を g_t^y と表そう。x_t と y_t という 2 つの時系列があったとき，その積 $x_t y_t$ の成長率は

$$g_t^{xy} = \log x_t y_t - \log x_{t-1} y_{t-1} = \left(\log x_t - \log x_{t-1}\right) + \left(\log y_t - \log y_{t-1}\right) = g_t^x + g_t^y$$

と書ける。つまり,「積の成長率は成長率の和」になる。同様に差と商，べきについても公式を導くことができる。

[1] ソローモデルについて説明する次節ではもう一度，通常の離散時間の成長率を用いる。

問題 6.1. 次の事実を示しなさい。ただし，g^{xyz} や g^{x^α} はそれぞれ xyz の成長率，x^α などの成長率を意味する。すべての時系列変数（x, y, z, w）は正の値を取るものとする。ギリシャ文字（α, β, γ）は実数定数で正，負，またはゼロの値を取る。

1. $g^{x/y} = g^x - g^y$
2. $g^{x^\alpha} = \alpha g^x$
3. $g^{x^\alpha y^\beta z^\gamma} = \alpha g^x + \beta g^y + \gamma g^z$
4. $g^{x^\alpha / y^\beta} = \alpha g^x - \beta g^y$

6.2.2. コブ=ダグラス型生産関数

マクロ経済の総生産（実質 GDP） Y が，資本 K と労働 L という生産要素を投入することで生産されると考える。投入要素を利用・雇用するには費用がかかる。さらに，(K, L) から Y を作るという生産過程の効率性を「技術」という言葉で表現しよう。技術は公共的な性質を持つもの（例えば微積分の公式）もあれば，各企業が独自で持っている場合（いわゆる，企業秘密，コカ・コーラのレシピなど）も，特許などで守られている場合もあるだろう。理論的な分析を簡単にするために，これらはひとまず無償で利用可能な共有知識であるとして，すべてひっくるめて A で表す。生産における投入，産出の関係 $(K, L, A) \mapsto Y$ を総生産関数（aggregate production function） F を使って次のように書くことができる。

$$Y = F(K, L, A)$$

一般的な表現では分析を先にすすめるのが難しいので，パラメータ $0 < \alpha < 1$ を導入して，**コブ=ダグラス型**と呼ばれる次の形式で総生産関数を特定化する。

$$Y = AK^\alpha L^{1-\alpha}$$

上記の形式で生産関数に組み込まれた技術水準 A は**全要素生産性**（total factor productivity）と呼ばれる。

コブ=ダグラス型生産関数は次の性質を持っている。

規模に関して収穫一定である

投入要素を c 倍にすると，生産物も c 倍になる（$c > 0$）。$Y = AK^{\alpha}L^{1-\alpha}$ のとき，

$$\begin{aligned} Y' &= A\,(cK)^{\alpha}\,(cL)^{1-\alpha} \\ &= c^{\alpha+(1-\alpha)}AK^{\alpha}L^{1-\alpha} \\ &= cAK^{\alpha}L^{1-\alpha} \end{aligned}$$

が成り立つ。規模に関して収穫一定という性質は「1 次同次」とも呼ばれる。

限界生産性が平均生産性に比例する

資本の限界生産性（marginal product of capital, MPK) は次のように定義される。

$$MPK = \frac{\partial Y}{\partial K}$$

平均生産性は Y/K で定義される。コブ=ダグラス型生産関数を偏微分して，2 つの生産性の関係を見てみよう。

$$MPK = \frac{\partial Y}{\partial K} = \alpha AK^{\alpha-1}L^{1-\alpha} = \alpha\frac{AK^{\alpha}L^{1-\alpha}}{K} = \alpha\frac{Y}{K}$$

となる。

問題 6.2. 労働の限界生産性（marginal product of labor, MPL）と労働の平均生産性 Y/L の関係を調べなさい。

ゼロ利潤

企業が次の利潤最大化問題を解いて資本と労働の需要を決定するとしよう。

$$\max_{K,L} Y - rK - wL$$

ここで r は資本を市場から借りてくる場合に支払わなければならない実質レンタル率である [2]。通常，企業は資本を保有しているので会計的には費用とはならないので，機会費用として計算する。w は実質賃金率である。$rK + wL$ は生産活動のコストである。

利潤最大化のための 1 階条件は

$$MPK = r, \qquad MPL = w$$

である。つまり，

$$\alpha \frac{Y}{K} = r, \qquad (1-\alpha)\frac{Y}{L} = w$$

が成り立つ。もう少し整理すると，

$$rK = \alpha Y, \qquad wL = (1-\alpha)Y$$

となる。rK は資本に対して支払われる報酬額であり，wL は労働力に対して支払われる報酬額である。資本は総生産の α の割合を報酬として得る。$\alpha = rK/Y$ を**資本分配率**（capital share）と呼ぶ。同様に $1-\alpha = wL/Y$ を**労働分配率**（labor share）と呼ぶ。

これらの報酬を合算すると企業の費用になるのだが，企業の売上と一致することに注意しよう。

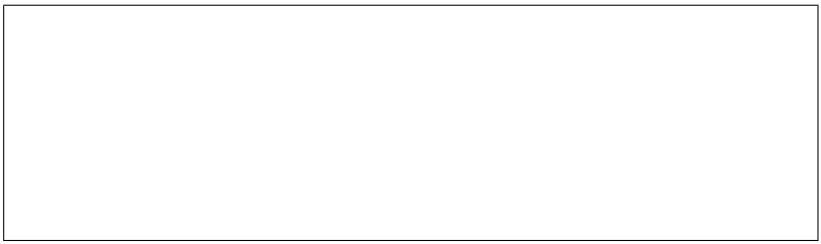
$$rK + wL = \alpha Y + (1-\alpha)Y = Y$$

実は，一般に規模に関する収穫一定を満たす生産関数のもとでは，企業利潤は高々ゼロになる。

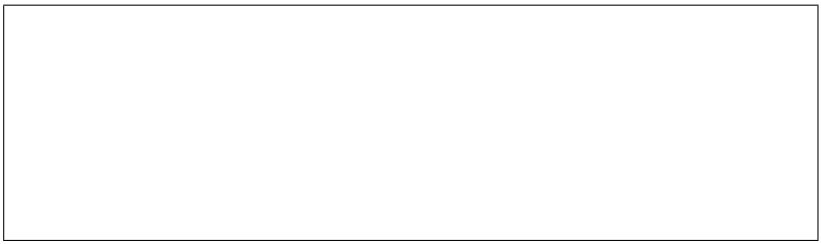
問題 6.3. 規模に関する収穫一定を仮定する。企業利潤が正になるような生産プランが仮にあったとすれば，その生産プランは最適にはならない。また，費用を一切かけずに操業停止ができるなら，負の利潤も最適になりえない。これらの事実を証

[2] ソロー・モデルの節で出てくる r と定義が少し異なっている。ここの r は資本減耗率 δ を含んでいるが，ソロー・モデルの節では δ を含んでいないと考えればよい。

明しなさい。



問題 6.4. 上記の数学的事実は，多くの実在する企業が正の利潤を稼いでいることと矛盾するだろうか。説明しなさい。



6.2.3. 成長会計

2 時点における，生産活動のデータ $(Y_{t-1}, K_{t-1}, L_{t-1})$, (Y_t, K_t, L_t) を手に入れたとしよう。

$$Y_t = A_t K_t^{\alpha} L_t^{1-\alpha}$$
$$Y_{t-1} = A_{t-1} K_{t-1}^{\alpha} L_{t-1}^{1-\alpha}$$

ここで，A_{t-1}, A_t に関する情報は得られない。

上式の対数差を取ると，

$$\log Y_t - \log Y_{t-1} = \left(\log A_t - \log A_{t-1}\right) + \alpha \left(\log K_t - \log K_{t-1}\right) + (1-\alpha)\left(\log L_t - \log L_{t-1}\right)$$

と書ける。対数差は成長率なので，節 6.2.1 の記法にならえば，

$$g_t^Y = g_t^A + \alpha g_t^K + (1-\alpha) g_t^L$$

となる。つまり，経済成長（Y の拡大）は，

- A の成長（技術進歩）
- K の成長（資本蓄積）
- L の成長（人口成長）

の 3 つの要因に分解される。**成長会計**（growth accounting）とは，このような要因分解を通して経済成長に貢献する要因を明らかにする分析のことである。ここで，g_t^Y, g_t^K, g_t^L の情報はデータから計算可能であるので，観測されない技術の成長率を次のように逆算できる。

$$g_t^A = g_t^Y - \alpha g_t^K - (1-\alpha) g_t^L$$

全要素生産性の成長率を「残差」によって求めるのである。これは**ソロー残差**と呼ばれている。

注意 6.1. 第 3 章で，GDP の支出面の恒等式 $Y = C + I + G + NX$ を使って寄与度を計算したことを思い出そう。寄与度は C や I の差分を Y で割って求めるものであって，C や I の成長率は使っていない。

6.2.4. JIP データベース

独立行政法人経済産業研究所では，日本の経済成長と産業構造変化を分析するための基礎資料として，日本産業生産性データベース（JIP データベース）を構築，公開している [3]。表 6.1 は JIP データベース 2018 として公表されている 2010 年までの成長会計の結果である。

JIP データベースでは産業部門ごとに成長会計を行っている。また，生産関数には資本や労働の質を考慮したものを使っているという違いがある。しかし，本章で説明した簡易的な成長会計を知っておけば詳細な分析マニュアルに取り組むことができるはずだ。

[3] https://www.rieti.go.jp/jp/database/jip.html

表 6.1.: 日本経済の成長会計（ソース：JIP データベース 2018）

マクロ（すべて）	**1995-2000**	**2000-2005**	**2005-2010**	**2000-2010**
GDP 成長率	1.35%	0.88%	−0.11%	0.89%
労働投入増加の寄与	−0.05%	−0.09%	−0.26%	0.07%
マンアワー増加	−0.37%	−0.42%	−0.55%	−0.04%
労働の質向上	0.33%	0.34%	0.29%	0.11%
資本投入増加の寄与	1.07%	0.39%	0.10%	0.03%
資本の量の増加	0.85%	0.28%	−0.05%	−0.03%
資本の質向上	0.21%	0.11%	0.15%	0.06%
TFP の寄与	0.33%	0.58%	0.04%	0.80%

6.3. ソローモデル

ソローモデルは，資本，労働，技術の成長によって経済成長を説明しようとするモデルである。以下のような特徴を持っている。

- 労働力の成長率は外生的に与えられた定数である。
- 技術の成長率は外生的に与えられた定数である。
- 資本蓄積は投資と資本減耗の関係によって定まる。
- 投資資金の供給源である貯蓄は所得の一定割合である。

労働 L と技術 A は以下の成長ルールに従う。

$$L_t = (1+n)L_{t-1}$$
$$A_t = (1+g)A_{t-1}$$

n は人口成長率，g は技術進歩率である。

資本の変化は投資と資本減耗によって引き起こされる。企業が行う粗投資を I_t と書く。生産のために利用された資本は一定の割合 $0 < \delta < 1$ だけ減耗すると仮定すると，次の方程式が得られる。

$$K_t - K_{t-1} = I_{t-1} - \delta K_{t-1} \tag{6.1}$$

この方程式を資本蓄積方程式と呼ぶ。ソローモデルの最重要方程式である。

所得 $Y_t = F(K_t, L_t, A_t)$ の一定割合 $0 < s < 1$ が貯蓄されるとする。貸付資金の市場が均衡しているとき貯蓄 sY_t と投資　I_t は一致するので,

$$I_t = sY_t = sF(K_t, L_t, A_t)$$

となる。

注意 6.2. 変数の初期値 K_0, L_0, A_0, パラメータ s, δ および生産関数 F を与えればシミュレーションは実行可能である。これは，下式右辺が t 期の変数とパラメータのみで表現されていることから分かる。

$$K_t = sF(K_{t-1}, L_{t-1}, A_{t-1}) + (1 - \delta)K_{t-1} \tag{6.2}$$

$$L_t = (1 + n)L_{t-1} \tag{6.3}$$

$$A_t = (1 + g)A_{t-1} \tag{6.4}$$

しかし，マクロ経済モデルとしての解釈性のために追加的な仮定を置く。節を変えて説明しよう。

6.3.1. 解析的な分析

総生産関数 F は次の形式を持つとする。

$$Y = F(K, AL)$$

つまり，A, L は生産に対して独立に影響するのではなく，AL という積の形で性質に作用する。AL を効率労働と呼ぶ。F は規模に関して収穫一定である。つまり,

$$F(cK, cAL) = cF(K, AL), \qquad c > 0.$$

収穫一定なので，次のように変形できる。

$$Y = F(K, AL) = AL \cdot F\left(\frac{K}{AL}, 1\right)$$

両辺を AL で割って,

$$\frac{Y}{AL} = F\left(\frac{K}{AL}, 1\right)$$

つまり，効率労働 1 単位当たりの生産 $y = Y/(AL)$ は効率労働 1 単位当たりの資本 $k = K/(AL)$ には，

$$y = F(k, 1) = f(k)$$

という関係がある。f は効率労働当たりの資本を効率労働あたりの生産に変換する生産関数である。

仮定 6.1. f には次の性質を仮定する。

- $f(0) = 0$,
- $f'(k) > 0$,
- $f''(k) < 0$,
- $f'(0+) > (\delta + g + n + gn)/s$,
- $f'(+\infty) < (\delta + g + n + gn)/s$.

問題 6.5. コブ=ダグラス型 $F(K, AL) = K^{\alpha}(AL)^{1-\alpha}$ のとき，上記 5 つの仮定がすべて満たされることを確認せよ。

$f''(k) < 0$ は資本の限界生産力が逓減することを表している。以下の命題を通して確認しておこう。

命題 6.1. $F = F(K, AL)$ は収穫一定，$k = K/(AL)$，$f(k) = F(k, 1)$ のとき，

$$f'(k) = \frac{\partial F}{\partial K}$$

が成り立つ。

証明. 定義に忠実に計算すればよい。

$$\begin{aligned}
\frac{\partial F}{\partial K} &= \lim_{\Delta K \to 0} \frac{F(K + \Delta K, AL) - F(K, AL)}{\Delta K} \\
&= \lim_{\Delta K \to 0} \frac{F\left(\frac{K}{AL} + \frac{\Delta K}{AL}, 1\right) - F\left(\frac{K}{AL}, 1\right)}{\frac{\Delta K}{AL}} \\
&= \lim_{\Delta K/(AL) \to 0} \frac{f\left(k + \frac{\Delta K}{AL}\right) - f(k)}{\frac{\Delta K}{AL}} \\
&= f'(k)
\end{aligned}$$

□

資本蓄積方程式 (6.1) を効率労働当たりの変数を用いて書き直そう。

$$K_{t+1} = sF(K_t, A_t L_t) + (1-\delta)K_t$$

両辺を $A_t L_t$ で割ると,

$$\frac{K_{t+1}}{A_t L_t} = s\frac{F(K_t, A_t L_t)}{A_t L_t} + (1-\delta)\frac{K_t}{A_t L_t}$$

$$\frac{A_{t+1}L_{t+1}}{A_t L_t}\frac{K_{t+1}}{A_{t+1}L_{t+1}} = sf(k_t) + (1-\delta)k_t$$

k の変化を表現する次の差分方程式を得る。

$$k_{t+1} = \frac{sf(k_t) + (1-\delta)k_t}{(1+g)(1+n)}$$

命題 6.2. ソローモデルの効率労働 1 単位あたりの資本 k_t は差分方程式

$$k_t = \frac{sf(k_{t-1}) + (1-\delta)k_{t-1}}{(1+g)(1+n)}, \qquad t = 1, 2, \ldots \tag{6.5}$$

に従って変化する。ただし,

$$k_0 = \frac{K_0}{A_0 L_0}$$

は所与である。

位相図

$$G(k) = \frac{sf(k) + (1-\delta)k}{(1+g)(1+n)}$$

と定義しよう。ソローモデルの差分方程式 (6.5) は $k_t = G(k_{t-1})$ と書ける。G のグラフは単調増加になる。

$$G'(k) = \frac{sf'(k) + (1-\delta)}{(1+g)(1+n)} > 0$$

図 6.1 は $k_t = G(k_{t-1})$ を作図したものである。45° の直線は $k_t = k_{t-1}$ なる点を表している。このグラフを使って k_0, k_1, k_2 を順々に定められることを確認してほしい。

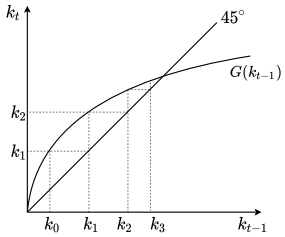


図 6.1.: $k_t = G(k_{t-1})$

図 6.1 では,

$$k_t = G(k_{t-1})$$
$$k_t = k_{t-1}$$

の交点（この点を $k_{t-1} = k_t = k^*$ としよう）に徐々に近づいていくことが分かる。このような収束性は f に課した仮定 6.1 から導かれるものである。詳細は省略するが G には次のような性質がある。

- G は単調増加である。
- G' は単調減少である。
- $k < k^*$ において，$G(k) > k$ である。
- $k > k^*$ において，$G(k) < k$ である。

ダイナミクス

k の時間変化を見るには，$k_{t+1} - k_t$ を調べるとよい。

$$\begin{aligned} k_{t+1} - k_t &= \frac{sf(k_t) + (1-\delta)k_t - (1+g)(1+n)k_t}{(1+g)(1+n)} \\ &= \frac{sf(k_t) - (\delta + g + n + gn)k_t}{(1+g)(1+n)} \end{aligned}$$

したがって,

$$k_{t+1} > k_t \iff sf(k_t) > (\delta + g + n + gn)k_t$$
$$k_{t+1} < k_t \iff sf(k_t) < (\delta + g + n + gn)k_t$$

$$k_{t+1} = k_t \iff sf(k_t) = (\delta + g + n + gn)k_t$$

が分かる。$sf(k_t)$ は効率労働あたりの貯蓄である。$(\delta + g + n + gn)k_t$ は効率労働あたりの資本を増やしも減らしもしない投資の水準になっていることが分かる。

定常状態

3 つ目の条件に注目しよう。$k_t = k^*$ が

$$sf(k^*) = (\delta + g + n + gn)k^*$$

となるとき，$k_t = k_{t+1} = k^*$ が成り立つ。この $k_t = k^*$ は時間変化しないので，**定常状態**（steady state）と呼ばれる。

$$[(1+g)(1+n) - (1-\delta)]\, k^* = sf(k^*)$$

$$(g + n + \delta + gn)\, k^* = sf(k^*)$$

定常状態 k^* はパラメータ δ, g, n, s に依存して変化する（f によっても変化する）。

$$k^* = k^*(\delta, g, n, s)$$

と書くとすれば，

$$\frac{\partial k^*}{\partial \delta} < 0, \quad \frac{\partial k^*}{\partial g} < 0, \quad \frac{\partial k^*}{\partial n} < 0, \quad \frac{\partial k^*}{\partial s} > 0$$

が成り立つ。

問題 6.6. 上の不等式を確認しなさい。

初期状態が $0 < k_0 < k^*$ である場合には，k_t は単調増加しながら k^* に収束する。$k_0 > k^*$ なら k_t は単調減少しながら k^* に収束する。いずれの場合でも

$\lim_{t\to\infty} k_t \to k^*$ が成り立つので，k^* は経済の長期の均衡状態であるとみなすことができる。

均整成長経路

長期均衡に収束した後の経済を考えよう。効率労働あたりの資本は k^* となっている。このとき，効率労働あたりの生産は $y^* = f(k^*)$ となり，一定値である。

しかし，私たちが本当に知りたいのは総所得 Y や一人あたりの総所得 Y/L といった変数であって，k^* や y^* ではない。しかし，Y や Y/L の情報は簡単に復元できる。

$$Y_t = A_t L_t y^* = A_t L_t f(k^*)$$
$$\frac{Y_t}{L_t} = A_t f(k^*)$$

この関係から Y や Y/L の成長率を計算することができる。

$$\frac{Y_t - Y_{t-1}}{Y_{t-1}} = \frac{A_t L_t f(k^*) - A_{t-1} L_{t-1} f(k^*)}{A_{t-1} L_{t-1} f(k^*)} = g + n + gn$$

$$\frac{(Y_t/L_t) - (Y_{t-1}/L_{t-1})}{(Y_{t-1}/L_{t-1})} = \frac{A_t f(k^*) - A_{t-1} f(k^*)}{A_{t-1} f(k^*)} = g$$

命題 6.3. ソローモデルの定常状態では，総生産 Y, 総消費 $C = (1-s)Y$, 総投資 $I = sY$, 資本ストック K の成長率は $g + n + gn \approx g + n$ となる。一人あたりの総生産 Y/L, 一人あたりの総消費 C/L, 一人あたりの総投資 I/L, 一人あたりの資本ストック K/L の成長率は g となる。

問題 6.7. 命題 6.3 を証明しなさい。

長期均衡ではすべての経済変数が同じ率で定率の成長を経験する。このような状況を均整成長（balanced growth）と呼ぶ。長期均衡ではソローモデルは「均整成長経路」（balanced growth path）に乗っている。

成長率に関する命題 6.3 を日常的な表現で書き換えておこう。この事実は非常に重要なので暗記しておくとよい。

命題 6.4. ソロー・モデルの均整成長経路を考える。一人あたりの実質 *GDP*，一人あたりの資本，一人あたりの消費の成長率はは技術進歩率と一致する。マクロ経済全体の実質 *GDP*，資本，消費の成長率は技術進歩率と人口成長率の和と一致する。

要素所得

マクロ経済には代表的な企業が一社あって，その企業が利潤最大化問題を解いていると考える。市場は完全競争的である。

$$\max_{K,L} F(K, AL) - (r + \delta) K - wL$$

資本コストについては実質利子率と資本減耗率を区別していることに注意する [4]。利潤最大化のための 1 階条件は

$$\frac{\partial F}{\partial K} = r + \delta, \qquad \frac{\partial F}{\partial L} = w$$

である。命題 6.1 で説明したとおり $\partial F/\partial K = f'(k)$ なので，均整成長経路上では

$$r = f'(k^*) - \delta$$

が成り立つ。収穫一定の下では利潤ゼロになるので（問題 6.3），労働所得について，

$$wL = Y - (r + \delta)K$$

[4] 資本を所有する家計は 1 単位の資本を貸し出すかわりに，市場で決まる実質利子率 r を要求する。さらに，資本の減耗分 δ も補填した上で 1 単位の資本を返さないといけないので，企業にとってのコストは $r + \delta$ となる。要するに δ は原状回復のためのコストである。ゼロ利潤条件は

$$Y = wL + rK + \delta K$$

となるが，GDP 統計において，分配面の GDP が労働所得，企業所得，資本減耗に大きく分類することを思い出そう。rK は企業所得を企業所有者に分配したものだと考えれば，理論とデータの対応関係が見えてくるはずだ。

が成り立つ。両辺を L で割ってやると，

$$\begin{aligned} w &= A\left[y - (r+\delta)k\right] \\ &= A\left[f(k) - f'(k)k\right] \end{aligned}$$

均整成長経路上では，

$$w = A\left[f(k^*) - f'(k^*)k^*\right]$$

となるので，w は A と同じ成長率で成長することが分かる。

命題 6.5. ソロー・モデルの均整成長経路上では実質利子率は一定である。賃金率は技術進歩率と同じ率で上昇する。

移行過程

ソローモデルに従う経済が均整成長経路にあるとする。次のような環境変化・政策変化が起こってモデルのパラメータが変化すると，定常状態からの乖離が生じるので新しい均衡経路に向かう移行が始まる。

- 積極的な移民政策によって n が上昇する。
- 子育て支援策を縮小して n が低下する。
- 技術開発を支援する政策によって g が上昇する。
- 資本ストックの保全技術が向上して δ が低下する。
- 大災害が発生して資本ストックの一部が破壊される。

ここでは，n の上昇に伴う均衡の移行を分析してみよう。経済は最初の定常状態 k_1^* にあるとする。n の上昇にともなって，定常状態の効率労働あたり資本ストックは減少する。

$$k_2^* < k_1^*$$

このとき，$k_0 = k_1^*$ として，方程式 (6.5) に従って，$t > 0$ の効率労働あたり資本ストックが単調に減少し，長期的には k_2^* に収束する。

6.3.2. 時間の流れと時間変数の測り方

動き続けるアナログ時計をイメージしてほしい。経済活動は連続的に流れる時間の中で行われている。しかし，モデル分析上の都合で離散時間的な変数 $Y_{t-1}, Y_t,$

K_{t-1}, K_t を使って分析している。データの測り方や変数の定義は恣意的なものだから，書き手と読み手の間で意識のすり合わせが必要だ。

連続的な時間を適当な，通常一定の，長さの期間に区切る（図 6.2）。各期間を「期」と呼ぶ。期の長さは 1ヶ月，3ヶ月，1 年という長さがよく使われる [5]。例えば，t 期の始まり（期首）は $t-1$ 期の終わり（期末）と一致している。

経済変数には，

- フロー変数
- ストック変数

という 2 種類の変数がある。

資本 K や労働力 L，技術 A は分析期間のある時点で計測される量のことである。このような変数をストック変数という。ストック変数は原理的には任意の時点で測ることのできる量であるが，通常は期末（あるいは同じことだが期首）に測る。ストック変数のことは「期末の残高」というイメージで捉えておけばよいだろう。図 6.2 の K_{t-1} が $t-1$ 期末の境界線近くに書かれているのはそのようなイメージを描いたものだ [6]。

生産 Y, 消費 C, 投資 I のような変数はある瞬間に計測されるものではなく，計測期間中に随時行われる経済活動の総量を計測する変数である。このように，ある期間中の経済活動を測る変数をフロー変数という。

図 6.2 にはソローモデルの変数間の関係を描いているので，図を見ながらソローモデルを再構築できるか腕試ししてみよう。

6.3.3. 技術進歩の分類

節 6.2 では $Y = AK^{\alpha}L^{1-\alpha}$ という形のコブ=ダグラス型生産関数を用いた。この節では，$Y = F(K, AL)$ という形の生産関数を用いている。コブ=ダグラス型に置き換えると，$Y = K^{\alpha}(AL)^{1-\alpha}$ と書ける。もちろん，A の意味合いは異なるのだが，コブ=ダグラス型を使う限り本質的な違いはない。前者の関数形を後者の関数形に変形してみよう。

[5] 現実のデータと，モデル分析に使われる理論的な変数との対応関係を真面目に考えようとすると大変に複雑である。例えば，月次データを分析するときには月の日数の長さとか，クリスマスを含むとか，ボーナス月を含むかなどの理論モデルには出てこないような要因で変動が生じる可能性がある。現実社会のすべての複雑性を理論モデルに取り込むことはできないので，データの方を理論にあわせる場合も多い（例：季節調整）。

[6] 変数の時間添字の選び方には自由度があることに注意してほしい。この章の説明では K_{t-1} を $t-1$ 期末の資本ストック（つまり，t 期期首の資本ストック）と定義したが，同じ量を表すのに K_t と書くこともできる。定義をきちんと読む必要がある。

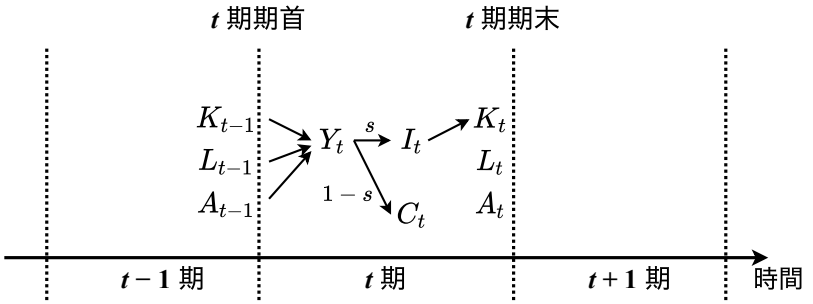


図 6.2.: ソローモデルの変数間の関係

$$
\begin{aligned}
Y &= AK^{\alpha}L^{1-\alpha} \\
&= K^{\alpha}\left(A^{\frac{1}{1-\alpha}}L\right)^{1-\alpha}
\end{aligned}
$$

$B = A^{\frac{1}{1-\alpha}}$ という変数変換をすれば,

$$Y = K^{\alpha}(BL)^{1-\alpha}$$

とできる。

1 つ注意をしておこう。$Y = AK^{\alpha}L^{1-\alpha}$ なるソロー・モデルでは，均整成長経路の 1 人当たり実質 GDP 成長率が A の成長率と一致しない。B の成長率と一致するので，$g^B \approx g^A/(1-\alpha)$ になる。節 6.2.4 の表 6.1 の 2000-2005 に注目してみると，$g^A = 0.58\%$ である。$\alpha = 1/3$（よく使われる資本分配率の近似値）とおくと

$$g^B = \frac{0.58}{1-(1/3)} \approx 0.87$$

であり，実際の数値と非常によく似た値になる。この時期の日本経済は定常状態の近くにあったのかもしれない（断定するだけの証拠はない）。

さて，生産関数に技術進歩を導入する方法には 3 パターンある。

1. $Y = AF(K,L)$ → ヒックス中立的技術進歩（要素節約的技術進歩）
2. $Y = F(AK,L)$ → ソロー中立的技術進歩（資本節約的技術進歩）

3. $Y = F(K, AL)$ → ハロッド中立的技術進歩（労働節約的技術進歩）

ソロー・モデルでは分析の都合上，ハロッド中立な技術進歩を念頭において考えている。

6.4. プログラミング：関数

6.4.1. 関数

「関数」（function）という言葉は数学でもおなじみのもので，

- 関数とは，数を変換する規則

のことだ。数に限定する必要はないので，抽象的な考え方が苦手でなければ，何らかの対象を別の対象に写す写像（mapping）と捉えておくとよい。これは数学でもプログラミングでも同じことだ。

プログラミングで言うところの関数は，もう少し広い概念である。例えば，一連のタスクをまとめて名前を付けたものも「関数」と呼ぶ。その関数が結果として意味のある値を返すかどうかは重要ではない[7]。値を返さない関数は，

- 画面上にメッセージを出力するとか，
- ファイルにデータを出力するとか，
- プログラムの他の所で定義されているオブジェクトを上書き変更する

などといった操作を行っている。プログラミングを実行しているコンピュータの状態に変更を与えるような効果を「副作用」（side effect）という。

関数はオブジェクトに変換操作を施して結果として別のオブジェクトを返すもの，基本的には副作用は避けるべきものだと考えておこう。つまり，

- 関数とは，オブジェクトを変換する規則

のことだ。副作用を避けるというのは結構難しいもので，Python では配列やリストを操作する関数を作るときには慎重に書かないと意図せず副作用を作りこんでしまうことがある。入力された配列を関数の中で書き換えてしまうという間違いがよく起こる。こればかりは慣れるしかないので，今は次のことだけ覚えておこう。

- 副作用は避ける。
- 副作用を使うときは意識的に行う。

[7] このような「関数」をプロシージャと言って区別することもあるが，Python ではそのような区別はない。

6.4.2. Python の関数定義

いつも通り次のコードを実行しておこう。

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Python における関数定義の基本は `def` キーワードを用いる次のようなコードだ。

```
def f(x):
    return x ** 3 - 10 * x
```

この関数は数学的な関数 $f(x) = x^3 - 10x$ を Python で表現したものだ。
`for` や `if` と同じように,

- コロンの後に改行,
- 改行後のブロックは空白 4 つでインデント

という形式で関数の本体ブロックを明示している。`return` の右側の値が関数呼び出しの結果として返される [8]。

```
def 関数名 (仮引数名):
    関数の本体（結果を計算するための長いコード）
    return 結果
```

関数名 (実引数) という形式で関数を呼び出すことができる。関数呼び出しで指定された実引数が，関数定義の仮引数に代入されて関数の本体ブロックのコードが実行される。

```
f(3)
```

```
-3
```

```
f(-4)
```

```
-24
```

実引数がどのような型であるかを関数定義の際に指定する必要がないので，本体ブロックの実行に支障がなければどんな型のオブジェクトも実引数に入れることが

[8] R や Julia とは異なり，Python では明示的な `return` 文が必要である。

できる。例えば，**NumPy** の配列はべきや四則演算を自由に行えるので，次のような関数呼び出しが可能である。

```
a = np.array([1, 2, 3])
f(a)
```

```
array([ -9, -12,  -3])
```

問題 6.8. 次のコードを実行するとどのような結果になるか。予想し，実行しなさい。予想通りの結果にならなかった場合は理由を考えなさい。

```
b = [1, 2, 3]
f(b)
```

数学的な関数を定義したら作図をしてみよう。結果は図 6.3 のようになる。

```
x = np.linspace(-4, 4, 200)
plt.plot(x, f(x))
plt.show()
```

複数の引数を取る関数を定義することもできる。次のコードは $\alpha = 0.3$ のコブ=ダグラス型関数を定義する。

```
def cd(x, y):
    alpha = 0.3
    return x**alpha * y**(1 - alpha)
```

```
cd(2, 3)
```

```
2.6564024798866686
```

なお，コロンの後の改行は必須ではなく，本体が 1 行だけのシンプルな関数は改行をせずに 1 行に書くこともできる。

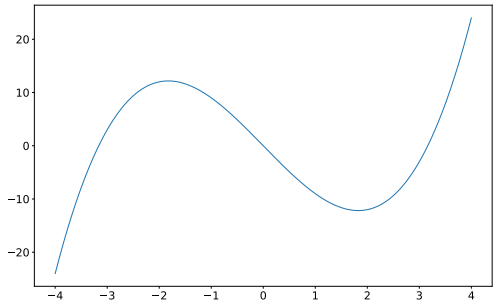


図 6.3.: $f(x) = x^3 - 10x$ のグラフ

```
def digits(x): return np.ceil(np.log10(x + 1))

digits(np.array([102, 2020, 155550]))
```

```
array([3., 4., 6.])
```

格段に読みやすくなったりしないし，行数をケチる必要はないので，いつでも改行を入れる方がよい。

ラムダ式

def による関数定義の他にも関数を定義する方法がある。**ラムダ式** (lambda expression) を用いるものだ。

数字の桁数を数える先程のコードをもう一度使おう。

```
digits2 = lambda x: np.ceil(np.log10(x + 1))
digits2(993)
```

```
3.0
```

ラムダ式は名前を持たない関数（無名関数）を使いたいときに便利な記法である [9]。

[9] def を使える局面でも lambda を使いたがる人もいる。好みの問題かもしれないけど，読みにくいのでやめたほうがいい。

Python ではどこでも def で関数を作れるので，ラムダ式を知らないと実現できないことはまったくないはずだ。しかし，2 つの理由で覚えておくとよい。

- 他の人が書いたコードを読むとき。ラムダ式は非常によく好まれる。
- lambda は Python のキーワードなので，変数名として使うことができない。数理モデルの分析では λ という変数名をよく使うのだけど，アルファベットの変数名としては lamda にする。スペルミスではなく衝突を避けるためだ。

ラムダ式を使うべき局面は，例えば **Pandas** のデータフレームや列の apply() メソッド，transform() メソッドを用いるときなど，1 回限りの無名関数を使いたいときである。

```
x = pd.DataFrame(np.arange(12).reshape(4, 3), columns=list("ABC"))
x.apply(lambda x: x ** 2)
```

```
     A    B    C
0    0    1    4
1    9   16   25
2   36   49   64
3   81  100  121
```

関数（メソッド）の引数として関数を渡している。これは高階関数という機能である。後ほど紹介する。

6.4.3. 引数のテクニック

落ち穂拾い的に Python の引数に関するルールを書き連ねておこう。今までですでに説明なく使ってきたものも含まれるので，思考の整理に役立ててほしい。

位置引数とキーワード引数

cd(2, 3) という関数呼び出しは実引数を書いた位置によって，どの仮引数に代入されるかが決まる。上で def cd(x, y): と定義したので，x=2, y=3 という代入操作が関数の本体ブロックの最初に実行される。このような書いた位置によって判定される引数を**位置引数**（positional argument）という。

関数と引数の関係についてはもう少し話しておくべきことがある。すでに実例では使用しているのだが，関数呼び出しの際に x=2, y=3 という明示的な仮引数指定ができる。

```
cd(x=2, y=3)
```

```
2.6564024798866686
```

このように指定した引数は**キーワード引数**（keyword argument）という。なにが嬉しいのか？キーワード引数は順序を入れ替えることができるのだ。

```
cd(y=3, x=2)
```

```
2.6564024798866686
```

一部だけをキーワード引数にすることもできる。

```
cd(2, y=3)
```

```
2.6564024798866686
```

位置引数より先にキーワード引数を書くことはできない。次のコードはエラーになるので，実行してエラーメッセージを確認しておこう。（必ず実行すること）

```
cd(x=2, 3)
```

デフォルト引数

引数の一部はデフォルトの値を設定することができる。

```
def production(k, l, a=1):
    alpha = 0.3
    return a * k**alpha * l**(1 - alpha)
```

引数の指定を省略するとデフォルト値が使われる。

```
production(2, 3)
```

```
2.6564024798866686
```

引数を指定するとデフォルトが上書きされて指定した値が使われる。

```
production(2, 3, 2)
```

```
5.312804959773337
```

* と **

関数を定義するとき，事前に引数の数が分からないことがある。「可変長引数」を作るには 1 つにはリストを入力に受け付けるようにする方法がある。任意個の要素を持つ 1 個のリストを引数として設定する代わりに，次のように書くこともできる。引数の数を数えるだけのシンプルなコードだ。

```
def number_of_args(*args):
    return len(args)

number_of_args("First", 2, [3, 2, 1], "End")
```

```
4
```

args という名前は重要ではない。その前に付記された * が重要である。* を付記された変を関数定義の仮引数リストに書くと，ゼロ個以上不定個の引数を受け付ける関数を書くことができる。（いくつあるかは事前には分からない）引数は，関数本体の中では args という名前のリストになっている。

位置引数だけでなくキーワード引数も可変長にできる。これには ** という記号を付記して仮引数リストを作ればよい。次の関数は，キーワード引数が格納された辞書をそのまま返している。関数本体で引数を使って何らかの操作をしたいときには，辞書として引数にアクセスできる。

```
def show_arguments(**kwargs):
    return kwargs

show_arguments(x=1, y=2, z=[1, 2])
```

```
{'x': 1, 'y': 2, 'z': [1, 2]}
```

位置引数はキーワード引数に先行しなければならないので，*args と **kwargs を両方書くときには順序に気をつけよう。

```
def function(*args, **kwargs):
    return (args, kwargs)

function("x", "y", a=1, b=2, c=3)
```

```
(('x', 'y'), {'a': 1, 'b': 2, 'c': 3})
```

さて，関数定義の可変長位置引数を示す *は関数本体ではリストを作り，可変長

キーワード引数を示す** は関数本体では辞書を作る。* や ** を関数呼び出しのときに使うとリストや辞書を使って引数を指定できる。

```
xy = (2, 3)
cd(*xy)
```

```
2.6564024798866686
```

```
xydict = {'x': 2, 'y': 3}
cd(**xydict)
```

```
2.6564024798866686
```

6.4.4. 副作用

副作用の代表例は画面上への出力だ。

```
def g(x):
    print(x)
    print("Doubling...")
    return 2 * x

g(3)
```

```
3
Doubling...
6
```

わざわざこのような説明しているのは，ときどき次のような誤りをする人がいるからだ。print() を使った画面上の出力は関数の出力ではないので，このコードはエラーになる。値を返さない関数を計算式の中で使うことはできない。

```
def h(x):
    print(2 * x)

4 * h(3)    # 4 * 2 * 3 のつもり
```

問題 6.9. 上のコードを実行するとどのような結果になるか。確認せよ。

上のコードのような誤りは比較的容易に問題点に気づけるので大きなミスにはつながらないだろう。しかし，次のコードはより深刻な問題を含んでいる。

```
def double_1st_elem(x):
    y = x
    y[0] = 2 * x[0]
    return y

x = [1, 2, 3]
double_1st_elem(x)
```

```
[2, 2, 3]
```

次のような意図で書かれたコードだ。

1. 引数 x には変更を加えたくないので，結果の配列 y を x のコピーとして作成する。
2. y の第 0 要素を 2 倍する。
3. y を返す。

なんとなく正しいコードのように感じただろうか。しかし，x を変更したくないという気持ちは裏切られることになる。

```
x
```

```
[2, 2, 3]
```

本書のレベルで「なぜこのような結果になるか」の説明をすることは難しい（その必要もないだろう）。リストや配列の「コピー」は慎重に行わなければならないということだけ肝に命じておけば十分だろう [10]。今の場合には次のようにすれば

[10] Python の変数はオブジェクト（メモリ上のデータ）へのポインタ（位置を示す目印）であり，リストはポインタを並べた配列である。y = x という代入文で x がリストだった場合，ポインタへのポインタが指し示す本体の数字をコピーすればオブジェクトの共有が起こらないのだけど，そこまで深追いしない。このような振る舞いについては Python Tutor の可視化を使って確認するとよい。x がリストの

問題を解決できる。「全要素」を表すコロンを使った下記のテクニックは多用されるので覚えておこう。

```
def double_1st(x):
    y = x[:]
    y[0] = 2 * x[0]
    return y

x = [1, 2, 3]
double_1st(x)
```

```
[2, 2, 3]
```

```
x
```

```
[1, 2, 3]
```

引数を変更するという厄介な副作用を取り除くことができた。

6.4.5. 関数のスコープ

次のコードを見てみよう。a という変数が関数の外側と内側で 2 回定義されている。(3) の関数呼び出しのときに (2) `a = 0` が実行される。関数呼び出しの結果は `a = 0` の影響を受けている。しかし，関数呼び出しが終わった後に a の値を表示しようとすると，関数呼び出しがなかったかのように元の値 3 が表示される。

```
a = 3               # (1) a==3

def fun(x):
    a = 0           # (2) a==0
    return x + a

print(fun(0))       # (3) a==0
```

```
0
```

```
print(a)            # (4) a==3
```

```
3
```

ケース（https://kjst.jp/453）と x が数のケース（https://kjst.jp/t0i）を較べてみよう。深くネストされたリストの完全なコピーを作るには copy.deepcopy() を使う

関数の内側 (2) で定義された a と外側 (1) で定義された a は同名だが別のオブジェクトである。プログラムが名前を探索する範囲のことをスコープと呼ぶ。関数は独自のスコープを持っていて，新しい変数を定義するコードが実行されるときに，外側の変数に影響を与えることがないように，新しい変数を作る。変数を定義するコードが実行されない限りは関数の外側の変数を自由に使うことができる。

```
b = 3
def fun2(x):
    y = b
    return x + y

fun2(3)
```

```
6
```

前節「副作用」では，リストが誤って書き換わってしまう挙動を紹介した。平仄が合わないと感じるかもしれないが，そういうものだと受け入れてほしい。

6.4.6. 高階関数

関数はオブジェクトを変換して，1 個のオブジェトを返す [11]。

Python では関数も普通のオブジェクトなので，

- 関数を返す関数
- 関数を引数にする関数（上で **pandas** データフレームの `apply()` メソッドを紹介した），
- 関数を引数にして関数を返す関数

を定義できる。やってみよう。

コブ=ダグラス型生産関数

ソロー・モデルで用いたコブ=ダグラス型生産関数

$$F(K, L, A) = K^{\alpha}(AL)^{1-\alpha}$$

[11] `return` を書かなかった，書き忘れた場合は `None` という特別なオブジェクトを返す。

は α でパラメータ付けられている。つまり，α を決めれば F が決まる，という関係にある。これが高階関数だ。

```
def cd_factory(a):
    def F(K, L, A):
        return K**a * (A * L)**(1 - a)
    return F

cobb_douglas = cd_factory(0.33)
cobb_douglas(2, 3, 1)
```

```
2.624285852902312
```

関数を返す関数を定義するには次のことをする。

- 関数定義の中で関数を定義する。
- 関数定義の中で定義した関数を返す。

シミュレーション

漸化式

$$y_t = ay_{t-1} + b, \qquad t = 1, 2, \ldots, T-1$$

で表現される時系列のシミュレーションも高階関数で書ける。

まず，次のように書き換えると，漸化式の右辺（更新ルール）が高階関数（関数を返す関数）であることが分かる。

$$y_t = G(y_{t-1}), \qquad G(y) = ay + b$$

すなわち，

$$(a, b) \mapsto G$$

これは先程と同様にできる。

```
def makeG(a, b):
    def G(y):
        return a * y + b
    return G
```

シミュレーションは更新ルール G と，初期値 y_0 と，シミュレーションの長さ T を与えて，$(y_0, y_1, \ldots, y_{T-1})$ を返す高階関数（関数を引数とする関数）として理

解できる。

$$(G, y_0, T) \mapsto (y_0, y_1, y_2, \ldots, y_{T-1})$$

以下の例では,

- `y0 = np.asarray(y0)`：入力された初期値 `y0` が NumPy の配列であることを保証する。
- `y = np.empty((T, *y0.shape))`：初期値 `y0` のシェイプが `(a,b)` なら，結果を格納する配列のシェイプは `(T,a,b)` である。
- `y[t] = update_rule(y[t-1])`：更新ルールはこの関数の引数なので，そのまま使う。

```
def simulate(update_rule, y0, T):
    y0 = np.asarray(y0)
    y = np.empty((T, *y0.shape))
    y[0] = y0
    for t in range(1, T):
        y[t] = update_rule(y[t-1])
    return y

G = makeG(a=0.6, b=1)
simulate(G, y0=10, T=20)
```

```
array([10.        ,  7.        ,  5.2       ,  4.12      ,
        3.472     ,  3.0832    ,  2.84992   ,  2.709952  ,
        2.6259712 ,  2.57558272,  2.54534963,  2.52720978,
        2.51632587,  2.50979552,  2.50587731,  2.50352639,
        2.50211583,  2.5012695 ,  2.5007617 ,  2.50045702])
```

定義済みの関数を少し変更する

先程の G はランダムな撹乱項が入っていなかったので，これを追加しよう。もう一度定義し直してもいいのだけど，撹乱項を足すだけの変更なので，元の定義は有効活用しよう。つまり，次のような変更をする。

$$G(y) = ay + b$$
$$\downarrow$$
$$G_\varepsilon(y) = G(y) + \varepsilon$$

これは関数 G を受け取って関数 G_ε を返す高階関数だ。

われわれは G は引数を 1 つだけ受け取ることを知っているが，これは修正を施したい関数 G によって違う。どんなケースにも対応できるようにしたい場合には，可変長引数を用いると実現できる。

```
def randomize(g, random):
    def randomized(*args, **kwargs):
        return g(*args, **kwargs) + random()
    return randomized

rng = np.random.default_rng(123)
Ge = randomize(G, random=rng.normal)
simulate(Ge, y0=10, T=20)
```

```
array([10.        ,  6.01087865,  4.23874054,  4.83116958,
        4.09267617,  4.3758366 ,  4.20260575,  2.8850998 ,
        3.2730121 ,  2.64721181,  2.26593797,  2.4567301 ,
        0.94810765,  2.7610307 ,  1.98552874,  3.19158667,
        3.05127312,  4.36279695,  2.95770876,  2.4628304 ])
```

`*args,**kwargs` を引数にするのは関数を受け取って関数を返す高階関数を作るときにはよく使われるので覚えておくとよい。詳しく知りたい人は「デコレータ」というキーワードで検索してみよう。

6.5. プログラミング：ソロー・モデル

理論パートでは，

- 成長会計
- ソロー・モデル

という 2 つのトピックを扱った。成長会計を実行するために必要なプログラミングの知識はすでに習得済みなので，改めて解説する必要もないだろう。

6.5.1. 成長会計

自力で実行できるだけの力がついているはずなので省略する。

6.5.2. ソロー・モデルのシミュレーション

ソロー・モデルを (6.2), (6.3), (6.4) を用いてシミュレーションしてみよう。

```
def solow(s, delta, g, n, F):
    def solow_g(x):
        K0, L0, A0 = x
        K1 = s * F(K0, L0, A0) + (1 - delta) * K0
        L1 = (1 + n) * L0
        A1 = (1 + g) * A0
        return np.array([K1, L1, A1])
    return solow_g
```

具体的なパラメータを設定してみよう。コブ=ダグラス型の生産関数はすでに作っておいたものを使う。

```
params = {'s': 0.3,
          'delta': 0.05,
          'g': 0.03,
          'n': 0.01,
          'F': cd_factory(0.33)}

G = solow(**params)
```

シミュレーションはすでに作っておいた simulate() 関数を使えば良い。初期値は適当に [2,1,1] にしておいた。シミュレーション期間は T = 100 とする。結果は 2 次元配列（行列形式）になるはずなので，pandas のデータフレームにしておくと便利だ。columns=['K', 'L', 'A'] と分かるのは，solow() 関数をそのように定義したからだ。

```
sol = pd.DataFrame(simulate(G, [2, 1, 1], 100),
                   columns=['K', 'L', 'A'])
sol.tail(5)
```

```
             K         L          A
95  255.371753  2.573538  16.578161
```

```
 96   265.704289   2.599273   17.075506
 97   276.452398   2.625266   17.587771
 98   287.632845   2.651518   18.115404
 99   299.263070   2.678033   18.658866
```

データフレームなら，プロットも簡単だ（図 6.4）。

```
fig, axes = plt.subplots(1, 3, figsize=(12, 4))

for var, ax in zip(['K', 'A', 'L'], axes):
    sol[var].plot(ax=ax, title=var)

plt.show()
```

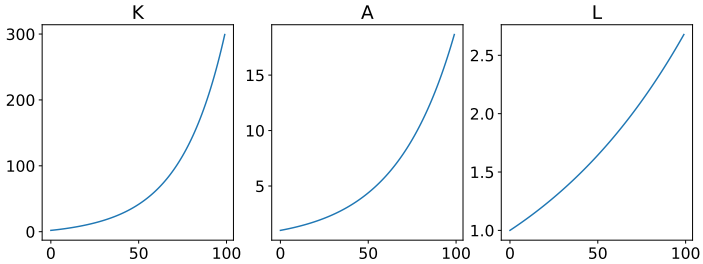


図 6.4.: ソロー・モデルのシミュレーション結果

効率労働あたりの資本は定常状態に収束しているだろうか。確認してみよう（図 6.5）。

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

sol['k'] = sol.K / sol.A / sol.L
sol.k.plot(ax=ax)
plt.show()
```

政策変化

ソロー・モデルの移行過程を学ぶときには，例えば，

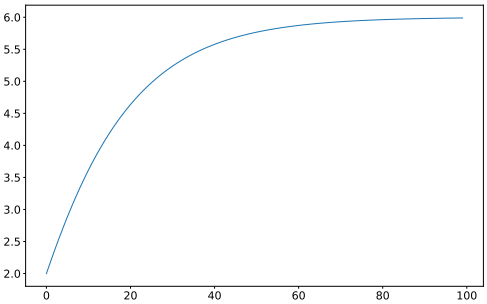


図 6.5.: 効率労働あたりの資本

- 貯蓄率の上昇が効率労働当たりの消費 $c = C/(AL)$ に下方ジャンプを引き起す,

といった分析をするのが一般的だが，これはマンキューの教科書に譲ることにする。

実際に関心があるのは総所得 Y, 消費 C など，および 1 人あたりの平均値 Y/L や C/L などであって，$C/(AL)$ 自体にはあまり関心がない。私たちはすでに，シミュレーションの技術を身に着けてしまったので，解析的に解けるかどうかという制約に縛られずに自由に分析することができる。やってみよう。

まず，最初のパラメータを設定しよう。意味はすでに明らかだろうから説明は省略する。

```
params = {'s': 0.2,
          'delta': 0.05,
          'g': 0.03,
          'n': 0.01,
          'F': cd_factory(0.33)}

G = solow(**params)
T = 20
initial = [6, 1, 1]
```

20 期経過したあとに貯蓄率（s）が 0.2 から 0.3 に上昇するとしよう。変化がないときと比べて消費の水準（経済の厚生に直結する）は増えるだろうか？あるいは，減るだろうか？

Y や C の計算を繰り返し行うのでヘルパー関数を定義しておく。

```
def computeYC(frame):
    frame['Y'] = params['F'](frame.K, frame.L, frame.A)
    frame['C'] = (1 - params['s']) * frame.Y
    return frame
```

ベンチマークとなる「変化なし」のケースのシミュレーションは次のように書ける。

```
nochange = pd.DataFrame(simulate(G, initial, 3*T),
                        columns=['K', 'L', 'A'],
                        index=range(3*T))
nochange = computeYC(nochange)
```

変化がある場合の，変化前のシミュレーションは次の通り。

```
before = pd.DataFrame(simulate(G, initial, T),
                      columns=['K', 'L', 'A'],
                      index=range(T))
before = computeYC(before)
before.tail()
```

	K	L	A	Y	C
15	7.806981	1.160969	1.557967	2.930646	2.344517
16	8.002761	1.172579	1.604706	3.033956	2.427165
17	8.209414	1.184304	1.652848	3.141662	2.513330
18	8.427276	1.196147	1.702433	3.253940	2.603152
19	8.656700	1.208109	1.753506	3.370973	2.696778

パラメータ変化が起こるので，パラメータと初期値を再設定する。初期値は，変化前のシミュレーションの最後の値だ。

```
params['s'] = 0.3
G_new = solow(**params)
new_initial = before.iloc[-1, :3].to_numpy()
```

シミュレーションをして before と after をつないでやる。ただし，before の最後と after の最初は同じものなので注意する。

```
after = pd.DataFrame(simulate(G_new, new_initial, 2 * T + 1),
```

```
                    columns=['K', 'L', 'A'],
                    index=range(T-1, 3*T))
after = computeYC(after)
saving_increased = pd.concat([before, after.iloc[1:]])
```

結果をプロットすると図 6.6 のようになる。貯蓄率の上昇によって一時的に消費が減少しているが，長期的に見ると資本蓄積の加速が効いてきて，変化がないケースよりも消費水準が大きくなる。長期の経済成長率は貯蓄率によらずに，技術進歩率と人口成長率のみで定まることを思い出そう。貯蓄率の上昇は，消費や所得の水準を高める効果はあっても，成長を高める効果は長期的には失われる。

```
saving_increased.C.plot()
nochange.C.plot();
plt.legend(['Saving rate increased', 'No change'])
plt.show()
```

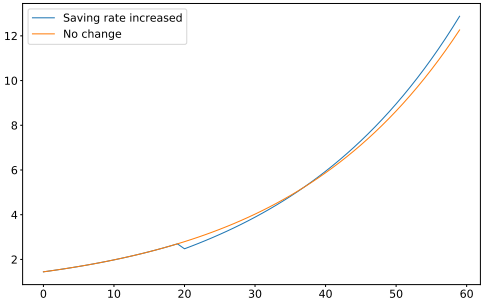


図 6.6.: 貯蓄率の変化

解析解 vs シミュレーション結果

さて，最後に，シミュレーションは万能ではないということに注意をしておこう。この章で学んだような初歩的なシミュレーションで得た結果は「ある特定のパラメータの組み合わせで成り立つ」ということしか言えない。何百通り，何千通りとシミュレーションを繰り返したとして，すべての実数パラメータの可能性を網羅す

ることはできない。もし，あなたがコンピュータがあれば数学はいらないなどと考えているようであれば，それは大きな間違いなので考えを改めよう。数学的な議論を無視して，コンピュータで（たまたま）出た結果を盲信してはいけない。

```
## Warning in file(con, "r"): cannot open file 'Python/ch007.py':

## Error in file(con, "r"): cannot open the connection
```

7. 2期間最適消費モデル

7.1. 概要

第 6 章のソロー・モデルの分析では，当期所得の一定割合 $(1-s)$ が消費され，残り s が貯蓄されると仮定した：

$$C_t = (1-s)Y_t, \qquad t = 1, 2, \ldots$$

この仮定には次のような批判があるだろう。消費は当期の所得だけでなく，将来所得にも影響されるはずだ。を将来の所得が高いと予想している個人は，現在借入れをしてでも消費を増やすだろうし，将来所得が小さくなると予想している個人は貯蓄を増やして消費を減らす行動を取るだろう。

ソロー・モデルを使うと，1 人あたり実質 GDP の長期的な成長率を決定する要因が技術進歩であることを示すことができる。この結果は基本的な成長会計とも整合的であるし，カルドアの事実のほとんどを説明する。使いやすいモデルではあるのだが，その良好な性能が「将来所得を考慮せずに当期の所得を決める」という強い仮定にあるのだとすれば，このモデルに対して,「たまたまうまくいっただけ」,「データに合うように都合よくモデルを作っているだけ」と考える人も現れるだろう。経済学者は最適化問題の解として説明できない仮定を受け入れないように訓練されているのだ。

この章から数章にわたって，消費者の意思決定の方法に関する仮定を緩めてもソロー・モデルの基本的な結論に影響しないことを説明する。導入部となるこの章では，もっとも単純な，代表的個人が 2 期間の消費計画を最適に決定する問題を紹介する。代表的個人とは，要するに，経済に人が 1 人しかいないということだ。そのような経済をイメージしにくければ，同じ選好を持つ多数の消費者がいる経済だと考えてもよい。この経済には獲得した財を貯蔵して増やす手段があって，1 期と 2 期の間で消費の配分を選ぶことができる。1 期にたくさん消費すると，2 期には消費を減らさなければならないので，消費者は**異時点間の資源配分**の問題に直面して

いるのだ。

この章で学ぶモデルは重要なモデルの基礎となる。計画期間を 2 から無限大に伸ばし，無限期間生きる代表的個人を考える。さらに，ソロー・モデルと同じ資本蓄積の方程式と合わせると，最適成長モデルあるいはラムゼー・モデルと呼ばれる成長モデルが得られる。計画期間を伸ばす代わりに，2 期間の最適化問題を解く経済主体が毎期生まれて，2 期生きたあとに死んでいくようにすると，世代重複モデルと呼ばれるモデルになる。

本章では，

7.2. 理論: ミクロ経済学の復習

7.2.1. 効用関数

異なる財を同時期に消費する計画を立てる静学的な問題から出発しよう。経済には財 1, 2 があって，消費量の配分を検討しているとしよう。消費量（より正確に言えば消費しようと計画する量）をそれぞれ x_1, x_2 のように下付き添字 1, 2 で表現する。消費者は「効用関数」（utility function）という選択の基準を持っているとする。効用関数は計画消費量 x_1, x_2 の関数として，

$$U(x_1, x_2)$$

と書く。U は計画消費量の組合せ (x_1, x_2) に点数を付けていく関数で，財 1, 2 ともに「消費量が増えるとうれしい」という普通の財であれば [1]，

$$x_1' \geq x_1 \quad \text{and} \quad x_2' \geq x_2 \quad \text{どちらか一方は厳密な不等式}$$

であるとき，

$$U(x_1', x_2') > U(x_1, x_2)$$

が成り立つ。これは「(x_1, x_2) 平面を右上に行けば行くほど効用が高まる」ことを表している。

その他もろもろの技術的な仮定を置くのだが，詳細は適当なミクロ経済学の教科書を参照してもらいたい。分析上で特に重要な仮定は，無差別曲線の形状に関する

[1] 財（goods）は良い（good）ものである。消費量を増やすと効用が下がる bads という概念もある。例えば，金融資産のリスクとか，大気汚染物質とか，労働とか。

ものである。ある点 (x_1', x_2') を通る無差別曲線は,

$$\left\{(x_1, x_2) \in \mathbb{R}^2_+ \mid U(x_1, x_2) = U(x_1', x_2')\right\}$$

なる集合のことである [2]。取りうる効用水準すべてについて無差別曲線が定義できるので，(x_1, x_2) 平面に無数の無差別曲線曲線を描くことができる。無差別曲線には次のことを仮定する。すべての無差別曲線が,

- 厚みのない「線」である,
- 他の無差別曲線と交わることがない,
- 原点に向かって厳密に凸である,
- なめらかな線である。

このような仮定の下で，無差別曲線は図 7.1 の原点に向かって凸な曲線のようになる。本当は無数に描けるのだが，3 本だけ描いていることに注意してほしい。右上に位置している無差別曲線の方が，左下に位置しているものよりも効用水準が高い。

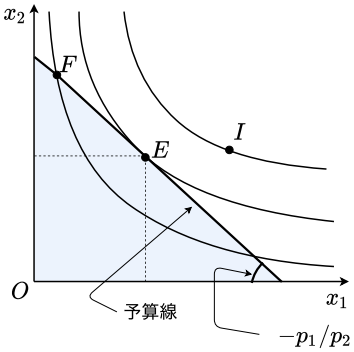


図 7.1.: 無差別曲線と最適消費点

任意の点 $(x_1', x_2'), U(x_1', x_2') = \bar{U}$, の近くに無差別曲線

$$\left\{(x_1, x_2) \mid U(x_1, x_2) = \bar{U}\right\}$$

の接線を引くことができる。接線の傾き dx_2/dx_1 を求めるには，全微分を用いて

[2] $\mathbb{R}^2_+$ は 2 次元空間（平面）の第 1 象限を表す。すなわち，$\mathbb{R}^2_+ = \left\{(x, y) \in \mathbb{R}^2 \mid x \geq 0,\ y \geq 0\right\}$.

次のような形式的な操作をするとよい[3]。

$$dU = \frac{\partial U}{\partial x_1} dx_1 + \frac{\partial U}{\partial x_2} dx_2$$

無差別曲線 「U = 一定」を保つ方向の微分を計算したいので，$dU = 0$ とすると，

$$\frac{dx_2}{dx_1} = -\frac{\left(\frac{\partial U}{\partial x_1}\right)}{\left(\frac{\partial U}{\partial x_2}\right)} = MRS$$

を得る。接線の傾きを**限界代替率**（marginal rate of substitution, MRS）と呼ぶ（図 7.2）。財 1 の消費を増やしたときに，財 2 の消費を減らさなければ効用が変化してしまう。効用を一定に保つために財 2 を減らさなければならない量が限界代替率である。効用水準を変えずに，財 2 から財 1 に代替しているのである。このことは全微分の公式に戻ってみると分かりやすい。

$$\frac{\partial U}{\partial x_1} \times 1 + \frac{\partial U}{\partial x_2} \times \underbrace{\left(-\frac{\partial U/\partial x_1}{\partial U/\partial x_2}\right)}_{=MRS} = 0$$

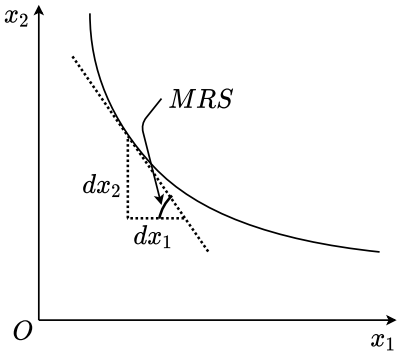


図 7.2.: 限界代替率

[3] 厳密な議論は陰関数定理による。

7.2.2. 予算制約

消費者は予算に限りがあるので，無尽蔵に消費を増やすわけにはいかない。財 1, 2 の価格をそれぞれ $p_1, p_2 > 0$, 消費者の予算を $m > 0$ としよう。すると，消費者は

$$p_1 x_1 + p_2 x_2 \leq m$$

が成り立つように x_1, x_2 を選ばなければならない。この不等式が成り立つ (x_1, x_2) の集合を「予算集合」(budget set) という。この不等式が等号で成り立つ集合（線分）を「予算線」(budget line) という。予算集合の不等式は

$$\frac{x_1}{m/p_1} + \frac{x_2}{m/p_2} \leq 1$$

と書けるので，(x_1, x_2) 平面に予算集合を描くと，横軸切片が m/p_1, 縦軸切片が m/p_2 であるような線分（傾きは $-p_1/p_2$）と，横軸，縦軸で囲まれた直角三角形の領域になる。図 7.1 の影付きの直角三角形の領域が予算集合で，斜辺が予算線である。

7.2.3. 効用最大化と均衡

消費者は，予算制約を満たすという条件の下で，効用 $U(x_1, x_2)$ が最大になるような (x_1, x_2) を選ぶ，という問題に直面している。これを数学的な表現を用いて次のように書く。

$$\begin{aligned} &\max_{x_1, x_2} U(x_1, x_2) \\ &\text{subject to} \qquad\qquad (P1) \\ &\quad p_1 x_1 + p_2 x_2 \leq m \end{aligned}$$

ここで，p_1, p_2, m は定数パラメータと考える。

効用を最大化する (x_1, x_2) を見つけることを「効用最大化問題を解く」，より一般に「最適化問題を解く」という。また，効用（より一般には「目的関数」）を最大化すると判明した (x_1, x_2) を「解」と呼ぶ。解を (x_1^*, x_2^*) と表現すると，

$$p_1 x_1^* + p_2 x_2^* \leq m$$

と

$$p_1 x_1 + p_2 x_2 \leq m \Longrightarrow U(x_1^*, x_2^*) \geq U(x_1, x_2)$$

が同時に成り立っている。つまり，

- 解は予算制約を満たす，
- 解は予算制約を満たす他の選択肢の中で，目的関数の値が最大である。

上記の効用最大化問題を解くためには，難しい数学は必要ない。次の 2 つのことがわかっていればよい [4]。

- 予算を余らせても効用に影響しないので，予算は使い切る方がよい。つまり，

$$p_1 x_1 + p_2 x_2 = m$$

- 無差別曲線はできるだけ右上にあるものの方がよいので，ギリギリ予算線に接するような無差別曲線を選ぶのがよい。つまり，無差別曲線の接線の傾き（MRS）が予算線の傾きと一致する。

$$MRS = -\frac{\partial U/\partial x_1}{\partial U/\partial x_2} = -\frac{p_1}{p_2} \tag{7.1}$$

図 7.1 の E 点が効用を最大化する消費点である。I 点は効用は高いものの，予算集合外にあるので解にならない。F 点は限界代替率と相対価格に関する条件 (7.1) を満たしていないので，解にならない。

条件 (7.1)

$$\frac{\partial U/\partial x_1}{\partial U/\partial x_2} = \frac{p_1}{p_2} \tag{7.2}$$

は特に重要なので経済的な意味を確認しておこう。右辺は財 1 の財 2 に対する相対価格であり，2 財に対する市場の評価を表している。財 1 を 1 単位多く消費するためには，財 2 は p_1/p_2 単位だけ減らさなければならない。

$$p_1 \times 1 + p_2 \times \left(-\frac{p_1}{p_2}\right) = 0$$

左辺の方はすでに説明したとおり，効用を変化させない代替の比率である。これは 2 財に対する消費者の主観的な評価を表している。式 (7.2) が成り立っていないと

[4] 無差別曲線の形状や配置に仮定したことに強く依存しているが，技術的になりすぎるので説明は省略する。

きには，消費の配分をかえることで効用を高めることができる。このとき，最大化は達成されていない。

「(7.2) が成り立たないなら最大ではない」からと言って,「(7.2) が成り立つときに最大である」とは言えないことに注意しよう。(7.2) は最大化問題の解が満たすべき必要条件であるが，十分条件であるとは限らない。しかし，われわれは無差別曲線に多くのことを仮定したので（原点に向かって厳密に凸な形状，右上の方が効用が高い），この問題はクリアされている。不安な読者はミクロ経済学か数理経済学の教科書を参考にするとよい。

問題 7.1. 予算線上にある消費点 (x_1, x_2) では,

$$\frac{\partial U/\partial x_1}{\partial U/\partial x_2} > \frac{p_1}{p_2}$$

である。このとき，x_1, x_2 をどのように変化させれば効用を高められるか。理由とともに答えなさい。（ヒント：図 7.1 の点 F から点 E に移動させることが必要である）

7.2.4. ラグランジュ未定乗数法

前述のような最適化問題を機械的に解く方法としてラグランジュ未定乗数法というものがあるので，簡単に解説しておこう。

$$\max_{x_1,x_2} U(x_1, x_2)$$

subject to

$$p_1 x_1 + p_2 x_2 \leq m$$

ラグランジアンと呼ばれる関数を次のように定義する。

$$\mathcal{L} = \underbrace{U(x_1, x_2)}_{\text{目的関数}} + \lambda \underbrace{[m - (p_1 x_1 + p_2 x_2)]}_{\text{制約条件, } \geq 0}$$

確実に $p_1 x_1 + p_2 x_2 = m$ となることを保証する条件が与えられている場合には，ラグランジアン $\mathcal{L}$ を x_1, x_2 と λ（未定乗数）で偏微分してゼロとなる点が解になる。

$$\frac{\partial \mathcal{L}}{\partial x_1} = \frac{\partial U}{\partial x_1} - \lambda p_1 = 0 \tag{7.3}$$

$$\frac{\partial \mathcal{L}}{\partial x_2} = \frac{\partial U}{\partial x_2} - \lambda p_2 = 0 \tag{7.4}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda} = m - (p_1 x_1 + p_2 x_2) \tag{7.5}$$

もし，$p_1 x_1 + p_2 x_2 < m$ となる可能性がある場合には，(7.5) の代わりに

$$\lambda [m - (p_1 x_1 + p_2 x_2)] = 0$$

という条件を使う。制約条件にゆとり（スラック）がある場合には λ がゼロになる。

例 7.1. 効用関数をコブダグラス型

$$U(x_1, x_2) = x_1^{\alpha} x_2^{1-\alpha} \tag{7.6}$$

に特定化しよう。このとき，

$$\mathcal{L} = x_1^{\alpha} x_2^{1-\alpha} + \lambda [m - (p_1 x_1 + p_2 x_2)]$$

$$\frac{\partial \mathcal{L}}{\partial x_1} = \alpha x_1^{\alpha-1} x_2^{1-\alpha} - \lambda p_1 = 0$$
$$\frac{\partial \mathcal{L}}{\partial x_2} = (1-\alpha) x_1^{\alpha} x_2^{-\alpha} - \lambda p_2 = 0$$

より，

$$\frac{\alpha}{1-\alpha} \frac{x_2}{x_1} = \frac{p_1}{p_2} \Longrightarrow \alpha p_2 x_2 = (1-\alpha) p_1 x_1 \tag{7.7}$$

を得る。さらに,

$$\frac{\partial \mathcal{L}}{\partial \lambda} = m - (p_1 x_1 + p_2 x_2) = 0$$

(7.7) を使うと,

$$x_1 = \frac{\alpha m}{p_1}, \qquad x_2 = \frac{(1-\alpha)m}{p_2} \tag{7.8}$$

を得る。これが財 1, 2 の需要関数になる。

未定乗数についても解いておこう。

$$\lambda = \frac{\alpha x_1^{\alpha-1} x_2^{1-\alpha}}{p_1} = \frac{\alpha}{p_1}\left(\frac{x_2}{x_1}\right)^{1-\alpha} = \frac{\alpha}{p_1}\left(\frac{(1-\alpha)p_1}{\alpha p_2}\right)^{1-\alpha} = \left(\frac{\alpha}{p_1}\right)^{\alpha}\left(\frac{1-\alpha}{p_2}\right)^{1-\alpha}$$

式 (7.6) と (7.8) を考慮して λ を解釈できる。λ は所得 m を 1 単位増やしたときに得られる追加的な効用（限界効用）である。追加的な 1 単位の所得は λ の分だけ効用を増加させる。

例 7.2. 効用関数を準線形と仮定しよう。

$$U(x_1, x_2) = \log x_1 + x_2$$

ただし，このような関数形の場合には $x_1 = 0$ や $x_2 = 0$ となる可能性がある（端点解）。こういうときには，$x_1 \geq 0, x_2 \geq 0$ という暗黙の制約条件をきちんと考慮してやる必要がある。ラグランジアンは

$$\mathcal{L} = u(x_1) + x_2 + \lambda\left[m - p_1 x_1 - p_2 x_2\right] + \lambda_1 x_1 + \lambda_2 x_2$$

になる。解が満たす条件は,

$$\begin{aligned}
\frac{1}{x_1} - \lambda p_1 + \lambda_1 &= 0 \\
1 - \lambda p_2 + \lambda_2 &= 0 \\
p_1 x_1 + p_2 x_2 &= m \\
\lambda_1 x_1 &= 0 \\
\lambda_2 x_2 &= 0
\end{aligned}$$

となる。少し整理すると，

$$\begin{aligned} p_1 x_1 + p_2 x_2 &= m \\ \lambda p_1 x_1 &= 1 \\ [\lambda p_2 - 1]\, x_2 &= 0 \end{aligned}$$

まず，$x_2 \geq 0$ であると仮定してみよう。このとき，

$$\lambda = \frac{1}{p_2}, \qquad x_1 = \frac{p_2}{p_1}, \qquad x_2 = \frac{m - p_2}{p_2}$$

となる。$x_2 \geq 0$ であるためには，$m \geq p_2$ でなければならないようだ。

では，$p_2 < m$ のときどうなるか。このとき，$x_2 = 0$ となるので，$\lambda p_2 - 1$ は不定である。しかし，$p_1 x_1 + p_2 x_2 = m$ から

$$x_1 = \frac{m}{p_1}$$

と分かる。

7.3. 理論: 2 期間の最適消費モデル

前節では，静学的な設定のもとで，2 財の選択に関する意思決定の性質を説明した。この節では，同じ財の 2 期間にわたる消費支出の配分を考える。動学的な計画問題であっても，均衡決定に関する考え方は同じであることを示す。しかし，時間経過には自然な順序関係があるので，分析に追加的な視点が加わることになる。例えば，消費量の経時的な増減を観察することで，経済成長についての含意を引き出せるかもしれない。

まずは，将来と未来をつなぐ「割引」について説明する。その後，2 期間の最適消費モデルを定式化し，均衡消費を求める。

7.3.1. 価値の割引

1 年後に 100 万円を受け取る契約を考えよう。この契約の，現時点における価値はいくらだろうか。100 万円より小さいということが直感的にも明らかだろう。1 年後に 100 万円受け取る権利をわざわざ 100 万円払って購入する人はいないだろ

う[5]。額面上の同額を翌年にスライドさせるよりも，利子の付く金融資産で 100 万円を運用する方が得だからだ。話を簡単にするために，マクロ経済には唯一の名目利子率があると仮定しよう[6]。名目利子率は貨幣価値を翌年に持ち越すときの額面評価額の拡大率のことである。今の 100 万円は 1 年後の $100(1+i)$ 万円と等価である。逆に言えば，来年の 100 万円は今の $100/(1+i)$ 万円と等価になる。将来の貨幣価値を現在の貨幣価値に換算する公式は

$$\text{現在の貨幣価値} = \frac{\text{将来の貨幣価値}}{1 + \text{名目利子率}}$$

であり，この操作を**割引**という。

貨幣価値はインフレ率の変化によって上下するので，名目利子率からインフレ率を除去した実質利子率を考えるのが便利である。p_t を t 期の価格，x_t を数量，i_t を名目利子率とする。これらはスカラー（ベクトルではない普通の数）である。貨幣価値は $p_t x_t$ のように書けるので，上記の割引の公式を置き換えて，

$$p_{t-1} x_{t-1} = \frac{p_t x_t}{1 + i_t}$$

とできる。

$$x_{t-1} = \frac{x_t}{(1 + i_t)\left(\frac{p_{t-1}}{p_t}\right)}$$

インフレ率を $\pi_t = (p_t / p_{t-1}) - 1$ とすれば，分母を次のように変形できる。

$$(1 + i_t)\left(\frac{p_{t-1}}{p_t}\right) = \frac{1 + i_t}{1 + \pi_t} = 1 + r_t$$

このように定義した r_t を用いると，実質的な価値の割引の公式を得る。

$$x_{t-1} = \frac{x_t}{1 + r_t}$$

r_t は実質利子率と呼ばれる。

[5] 犯罪多発などの理由でタンス預金が現実的でなければ，そのような契約もあるかもしれない。

[6] 実際には，借り手の信用力やマクロ経済環境に応じて各個人が支払わなければならない利子率は異なる。貸すときの利子率（預金につく利子率）は借りるときの利子率（ローンに課される利子率）よりもはるかに小さい。したがって，利子率が 1 つだけというのは現実には観測されないが，単純化の仮定として広く受け入れられている。

注意 7.1. 実質利子率の定義を近似的に,

$$r = \frac{1+i}{1+\pi} - 1 \approx i - \pi$$

とすることができる。定義式両辺の対数を取ると

$$\log(1+i) = \log(1+r) + \log(1+\pi)$$

となる。ここでも再び瞬時成長率と離散時間の実効成長率の差が現れる。瞬時成長率については,

$$i = r + \pi$$

が厳密な等号で成立している。名目利子率，実質利子率，インフレ率の関係を表すこの公式は**フィッシャー方程式**と呼ばれている。

7.3.2. 効用の割引

今年の消費から得られる効用と来年同じだけ消費することで得られる効用とを，今年のあなたが比較するとしよう。たとえば，今，寿司を食べに行くのと，1 年後に寿司を食べに行くことを計画するのでは，どちらが現在の効用を高めるだろうか。同じ消費行動をするのであれば，現在の消費と較べて将来の消費の方が効用に与える影響は小さくなるはずだ。価値と同様に，効用についても割り引く必要がある。

このような考察をもとに，次のような効用関数を使用することが考えられる。

$$U(c_1, c_2) = u(c_1) + \frac{u(c_2)}{1+\rho} = u(c_1) + \beta u(c_2)$$

$u(\cdot)$ を期間効用関数と呼ぶ。毎期の消費を当期の視点で評価したものである。将来の消費から得られる効用が現在の消費と比べて小さくなることを $1/(1+\rho)$ で掛けることで表現する。$\rho > 0$ を割引率，$1/(1+\rho) = \beta < 1$ を割引因子と呼ぶ。

7.3.3. 予算制約

ある消費者が効用

$$U(c_1, c_2) = u(c_1) + \beta u(c_2)$$

を最大化するように消費計画 c_1, c_2 を選びたいとしよう。c_1 は 1 期の実質消費，c_2 は 2 期の実質消費である。この消費者はどのような予算制約に直面しているだろ

うか。

この消費者は 1 期，2 期のそれぞれにおいて y_1, y_2 の実質労働所得を稼ぐとしよう。この経済では，期首に保有している貯蓄残高に対して利払いを受けられる。例えば，消費者の計画時点での貯蓄残高 s_0 に対して，利子 $r_1 s_0$ が 1 期に支払われる。1 期の期末にかけての貯蓄残高の変化 $s_1 - s_0$ は，1 期に受け取る労働所得 y_1 と利子所得 $r_1 s_0$ および 1 期に支払う消費支出 c_1 を用いて，次のように表現できる。

$$s_1 - s_0 = y_1 + r_1 s_0 - c_1$$

あるいは

$$c_1 = y_1 + (1 + r_1)s_0 - s_1$$

同様にして 2 期のお金の出入りは次のように表現できる。

$$c_2 = y_2 + (1 + r_2)s_1 - s_2$$

2 期の式を 1 期分割り引くと，

$$c_1 = y_1 + (1 + r_1)s_0 - s_1$$
$$\frac{c_2}{1 + r_2} = \frac{y_2}{1 + r_2} + s_1 - \frac{1}{1 + r_2}s_2$$

辺々足すと s_1 が消えて，

$$c_1 + \frac{c_2}{1 + r_2} + \frac{s_2}{1 + r_2} = (1 + r_1)s_0 + y_1 + \frac{y_2}{1 + r_2}$$

を得る。

貯蓄残高はマイナスにもなることに注意しよう。負の残高は債務を表している。この消費は c_1, c_2, s_2 を上手に選んで $U(c_1, c_2)$ を最大にしたい。しかし，s_2 がいくらでも大きなマイナスの値になるのであれば，効用関数は際限なく大きくできるだろう。最大化問題は意味をなさなくなってしまう。s_2 には下限を設定する必要がある。簡単のために，借金を残して死ぬ計画を立てないという条件を課そう。すなわち，

$$s_2 \geq 0$$

である。この条件の下では,

$$c_1 + \frac{c_2}{1+r_2} \leq (1+r_1)s_0 + y_1 + \frac{y_2}{1+r_2}$$

となる。表現を簡単にするため，$r_1 = 1, r_2 = r$ と書き直そう。考慮すべき制約条件は次のようになる。

$$c_1 + \frac{c_2}{1+r} \leq s_0 + y_1 + \frac{y_2}{1+r} \tag{7.9}$$

この式は

消費の割引現在価値の総和 ≤ 所得の割引現在価値の総和

という形式になっている。生涯所得以上には消費することはできないというまっとうな条件になっている。

なお，生涯所得に関する制約が成り立っている限りは 1 期末の借入れをいくらでも大きくすることができることに注意しよう。つまり，$s_2 \geq 0$ ではあるものの，s_1 については非常に大きなマイナスになっても構わない。この条件については,

$$s_1 \geq 0 \tag{7.10}$$

あるいは

$$s_1 \geq -\underline{s}$$

のような条件を付加して修正することがある。このような追加的な制約条件を借入制約（borrowing constraint）と呼ぶ。

問題 7.2. (7.9) を満たす c_1, c_2 の範囲を (c_1, c_2) 平面に描きなさい。

問題 7.3. (7.9) と (7.10) を満たす c_1, c_2 の範囲を (c_1, c_2) 平面に描きなさい。

7.3.4. 解法

借入制約のないケースを考えよう。消費者は

$$\begin{aligned} &\max u(c_1) + \beta u(c_2) \\ &\text{subject to} \qquad\qquad (P2) \\ &\quad c_1 + \frac{c_2}{1+r} \leq s_0 + y_1 + \frac{y_2}{1+r} \end{aligned}$$

を解く。ここで，この問題を解く際には s_0, y_1, y_2, r, β は定数パラメータと考える。

c_1 の価格が 1, c_2 の価格が $1/(1+r)$ と解釈すれば，問題 $(P1)$ と $(P2)$ は同じ形式を持っていることが分かるだろう。したがって，限界代替率が相対価格と一致するという条件が解を特徴づける。

$$MRS = \frac{u'(c_1)}{\beta u'(c_2)}$$

$$\text{相対価格} = \frac{1}{1/(1+r)} = 1 + r$$

したがって，最適性の必要条件は

$$\frac{u'(c_1)}{u'(c_2)} = \beta(1+r) \tag{7.11}$$

$$c_1 + \frac{c_2}{1+r} = s_0 + y_1 + \frac{y_2}{1+r}$$

となる。

問題 7.4. ラグランジュ未定乗数法を用いて同じ問題を解きなさい。また，ラグランジュ乗数 λ の経済的な意味を説明しなさい。

7.3.5. 解の性質

一般的な効用関数では解釈性が難しいので期間効用を特定化する。特に扱いが容易なのは，相対的危険回避度

$$\Theta(c) = -\frac{cu''(c)}{u'(c)}$$

が一定であるような効用関数である。これを CRRA 型効用関数（Constant Relative Risk Aversion）と呼ぶ。

$$u(c) = \begin{cases} \frac{c^{1-\theta}-1}{1-\theta} & \text{if} \quad \theta \neq 1 \\ \log c & \text{if} \quad \theta = 1 \end{cases}$$

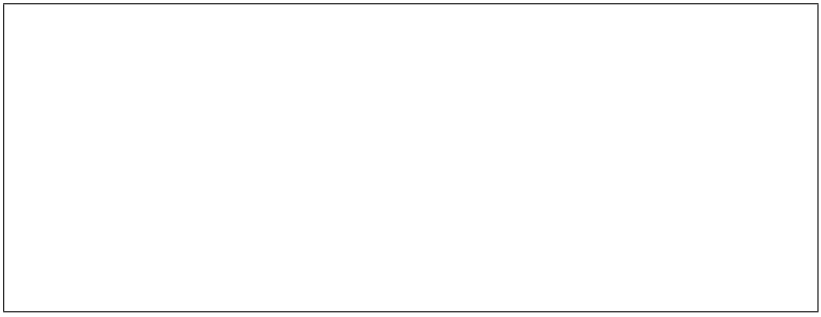
限界効用逓減を要請するので，$\theta > 0$ でなければならない [7]。

問題 7.5. CRRA 型効用関数に対して，

$$\Theta(c) = \theta$$

が成り立つことを確認しなさい。

[7] 期間効用関数の限界効用が逓減しないと，総効用 $U(c_1, c_2)$ が原点に向かって厳密に凸にならない。無差別曲線が直線的な形状（ファセットと呼ぶ）を持つと，解が無数に存在する，端点解が生じるといった厄介な問題が生じてくる。



CRRA 型を仮定すると，(7.11) の条件は

$$\left(\frac{c_2}{c_1}\right)^{\theta} = \beta(1+r) = \frac{1+r}{1+\rho}$$

とできる。

$$\frac{c_2}{c_1} = \left(\frac{1+r}{1+\rho}\right)^{1/\theta} \tag{7.12}$$

$\theta > 0$ より，次の命題を得る。

命題 7.1. *2 期間最適消費問題 (P2) の最適解 (c_1^*, c_2^*) は*

$$r \lessgtr \rho \Leftrightarrow c_2^* \lessgtr c_1^*$$

を満たす。

利子率 r は消費を遅らせることで追加的に得られる所得である。割引率 ρ は消費を遅らせることに対する効用のペナルティである。待つことの便益が相対的に小さいとき（$r < \rho$）には早めにたくさん消費するだろうから，$c_1^* > c_2^*$ が成り立つのである。

1 期の貯蓄 s_1 について見ておこう。毎期のお金の出入の式に $s_2 = 0$ を代入すると以下の式を得る。

$$c_1 = y_1 + s_0 - s_1$$
$$c_2 = y_2 + (1+r)s_1$$

ここで，(7.12) を使うと

$$\frac{y_2 + (1+r)s_1}{y_1 + s_0 - s_1} = \left(\frac{1+r}{1+\rho}\right)^{1/\theta}$$

これを s_1 について解く。

$$\begin{aligned} s_1 &= \frac{\left(\frac{1+r}{1+\rho}\right)^{1/\theta}(y_1 + s_0) - y_2}{(1+r) + \left(\frac{1+r}{1+\rho}\right)^{1/\theta}} \\ &= \frac{(1+r)^{1/\theta}(y_1 + s_0) - (1+\rho)^{1/\theta} y_2}{(1+r)(1+\rho)^{1/\theta} + (1+r)^{1/\theta}} \end{aligned}$$

消費者の効用最大化問題によって消費・貯蓄行動を特徴づけると，ソロー・モデルが想定するような単純な貯蓄関数と比べてはるかに複雑な貯蓄関数になる。

例 7.3. $y_2 = 0, s_0 = 0$ のケースは貯蓄関数についてもう少し考察をすすめることができる。この仮定は 2 期間世代重複モデルでも使われる。

$$s_1 = \left[\frac{(1+r)^{\frac{1-\theta}{\theta}}}{(1+\rho)^{\frac{1}{\theta}} + (1+r)^{\frac{1-\theta}{\theta}}}\right] y_1 \tag{7.13}$$

当期の所得の一部が貯蓄に割り振られるという式になっている。ただし，利子率 r に依存していることに注意しよう。ソロー・モデルの分析で学んだように，利子率は資本の限界生産性 MPK で決まるので，完全な定率という訳ではない。しかし，さらに $\theta = 1$ という条件をつければ利子率への依存性がなくなる。

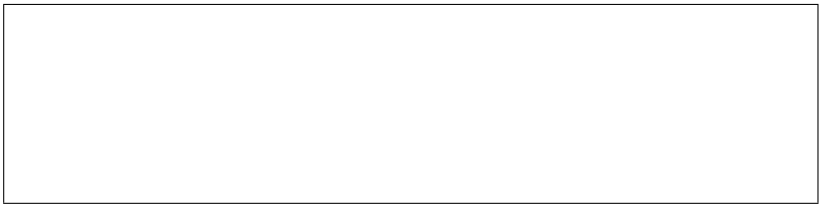
$$s_1 = \frac{1}{2+\rho} y_1$$

利子率の上昇は消費の意思決定において 2 つの効果を引き起こす。1 つは第 2 期の財価格（$1/(1+r)$）が下落することで，第 1 期の消費から第 2 期の消費への代替が進む。つまり第 1 期の消費を減少させる作用が働く。これを**代替効果**（substitution effect）という [8]。もう 1 つは**所得効果**（income effect）である。価格の下落によ

[8] 代替効果には，スルツキーの代替効果とヒックスの代替効果がある。スルツキーの代替効果を求めるときには，価格の変化が起こる前の最適消費点が価格変化後の予算線に乗るように所得の調整を行ったあとで，新しい予算線上に最適消費点を探す。スルツキーの代替効果は効用関数の形状によらずに必

り購買力が増加することで，第 1 期，第 2 期ともに消費を増加させる作用が働く。利子率 r の上昇に伴って貯蓄が増加する場合，第 1 期の消費が減少しているはずなので，代替効果が所得効果を上回っていることになる。逆に，利子率の上昇に伴って貯蓄が減少する場合，第 1 期の消費が増加しているので，所得効果が代替効果を上回っている。$\theta = 1$ のときには所得効果と代替効果が相殺し合う。

問題 7.6. (7.13) が r の増加関数になるのは $\theta > 1$ のときか，$\theta < 1$ のときか。判定せよ。



7.4. プログラミング

この章のプログラミングパートでは，Python のシンボリック計算ライブラリ **SymPy** と数値最適化のための scipy.optimize モジュール（**SciPy** ライブラリの一部）を紹介する。

7.4.1. シンボリック計算と数値計算

コンピュータを用いた数学的な計算には，シンボリックな計算と数値的な計算がある。シンボリックな計算というのは，記号の操作のみを用いて計算を行うことである。例えば，四則演算とべきからなる有限の操作だけを用いて計算をする代数計算や，初等関数の微分，一部の不定積分の計算などはシンボリックな計算の例である。数値的な計算というのは，具体的な数値と近似を用いて答えを導く操作と考えておけばよい。

シンボリック計算の身近な例としては，2 次方程式

$$ax^2 + bx + c = 0$$

ず非正になる。無限小の価格変化に関するスルツキー分解では 2 つの代替効果は一致する。

の求解問題がある。$a \neq 0$ のとき,

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

であることはよく知られている。係数の四則演算とべき根からなる有限回の操作によって解が表現されている。このような求解の手続きを**代数的に解く**と呼ぶ。得られた解表現を**解の公式**という。4 次方程式までは解の公式が存在することが知られている。5 次以上の一般の代数方程式（多項式 = 0 なる方程式）には解の公式が存在しない [9]。

5 次以上の方程式では解を見つけることはできないかというと，そうではない。解 x^* に収束する数列 $\{x_n\}$ をうまく見つけられれば（$\lim_{n\to\infty} x_n = x^*$），十分大きな N を用いて $x_N \approx x^*$ という近似式が成り立つ。$\lim_{n\to\infty} x_n = x^*$ が成り立つことを証明する手続きを**解析的に解く**という。解析的な解の近似によって解を求めることを**数値的に解く**，という [10]。

7.4.2. シンボリック計算

Python で代数計算（シンボリック計算）を行うときは **SymPy** というライブラリを用いる。このセクションのコードは Jupyter Notebook を用いて実行することを推奨する。次のコードを実行してインポートと，出力される数式の見栄えを調整する。

まずはシンボルとして用いる変数を定義する。アルファベットの単純なシンボルを定義するときには次のように `from sympy.abc import ...` というイディオムを使うとよい。

シンボリックな計算とは，次のような計算のことである。あまりにも自明の計算ではあるが，コンピュータにも私たちが行うような普通の，抽象的な計算ができるところに驚きがある。

2 次方程式も次のように解いてくれる。

シンボリック計算は代数計算だけではなく，微分の計算なども含まれる。公式の適用で機械的にできる操作はシンボリックな計算ができるのだ。

---

[9] アーベル=ルフィニの定理。「一般」という表現を誤解しないように。これは,「解の公式が存在しない代数方程式がある」ということ。解の公式が存在する部分集合の存在は否定していない。

[10] 解析的に解くという表現は必ずしも数値的に解くことと同義ではない。数列の各ステップ x_n の計算に無限の自由度を伴う変数（関数）の操作が必要な場合には数値的に解けない。コンピュータの物理的な制約で無限状態を表現できないからだ。したがって，あらゆる計算を有限近似で表現できたときにはじめて数値的に解くアルゴリズムを得たことになる。

これは，なんとなくおかしい気がする。分子と分母の x をまとめてくれたりはしないようだ。こういうときには simplify() というメソッドを使うとうまく整理してくれることがある。ただし，いつもうまくいくとは限らないことに注意しよう。

CRRA 型効用関数

$$u(c) = \frac{c^{1-\theta} - 1}{1 - \theta}$$

の相対的危険回避度 $-cu''/u'$ が定数であることを確認してみよう。ギリシャ文字は英語表記を使うとうまく表示してくれる。

シンボリックな関数に数値を代入したいときがある。例えば，ソロー・モデルの定常状態を考えてみよう。コブ=ダグラス型 $f(k) = k^{\alpha}$ を仮定すると，定常状態の効率労働あたり資本 k^* が満たす方程式は

$$sk^{\alpha} - (\delta + g + n + gn)k = 0$$

であり，**SymPy** では次のように求める。解きたい方程式を (式) = 0 の形にして，左辺の式のみを指定していることに注意しよう。

ここで，パラメータ α, s, δ, g, n を代入した値を計算したければ，次のように subs() メソッドを使うとよい。入力は辞書である。キーとしてパラメータのシンボル，値には代入したいパラメータ値を設定する。ss[0] としているのはリストの第 0 要素を抽出するお馴染みのコードだ。一般の方程式には複数個の解があるので，sp.solve() の結果は必ずリストになる。

すべてのパラメータに数値を代入するのではなく，例えば，パラメータ s を変化させたときに，定常状態の変化を見たいとしよう。s だけは代入せずにおいておく。

このオブジェクトを s の関数として扱いたいなら，**SymPy** のシンボリックなオブジェクトから数値的な関数を生成する必要がある [11]。これには sp.lambdify() という関数を使う。シンボリック計算から数値計算に橋渡しをしてくれる重要な関数である。

プロットしてみよう。

[11] 作図するだけなら sympy.plotting モジュールも使うことができる。https://docs.sympy.org/latest/modules/plotting.html

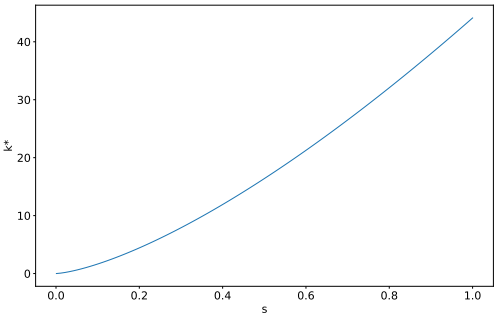


図 7.3.: s と k^* の関係

例 7.4 (ラグランジュ未定乗数法)**.** 次の最大化問題を解いてみよう。

$$\max x^{1/2}y^{1/2}$$
$$\text{subject to}$$
$$x + 3y = 100$$

ラグランジアン

$$\mathcal{L} = x^{1/2}y^{1/2} + \lambda(100 - x - 3y)$$

として,

$$\frac{\partial \mathcal{L}}{\partial x} = 0, \quad \frac{\partial \mathcal{L}}{\partial y} = 0, \quad \frac{\partial \mathcal{L}}{\partial \lambda} = 0$$

となる (x, y, λ) を求めればよい [12]。

うまくいった。しかし，この計算はいつでもうまくいくとは限らないことに注意しよう。シンボリックな計算では解けない方程式がある。

問題 7.7. 例 7.4 の数字を少し変えて，上記の手続きで解ける問題と解けない問題を 1 つずつ探しなさい。なお，**SymPy** では有理数は `sp.Rational(1/3)` などのようにして定義できる。

[12] λ の英語表記は lambda だが，Python のキーワードなので使えないため lamda としている。Unicode の文字名としては lamda が正しいそうなので，あながち間違いという訳でもないようだが。

例 7.5 (多項式近似)**.** シンボリックな計算が適用できる状況は限られているが，私たちがモデル構築に用いるような普通の関数の微分計算は安全に行える。例えば，

$$f(x) = \sin 3x + e^{x/2}$$

を $x = x_0$ のまわりで，2 次方程式で近似する問題を考えよう。

2 次までテイラー展開の公式は次の通りである。。

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2} f''(x_0)(x - x_0)^2$$

テイラー近似の公式に忠実に従えば，次のように書ける。

しかし，**SymPy** の `series()` メソッドを使うほうが簡単だろう。`removeO()` メソッドは近似誤差を無視するためのものだ。

近似がうまくいっていることを目視で確かめよう。$x = 2$ の周りで近似し，$1 < x < 3$ の範囲でプロットしてみよう。結果は図 7.4 のようになる。

7.4.3. 数値計算

最適化問題における最も重要な課題は，ある関数がゼロになる点を探すことである。本書では経済モデルの解を可視化したり数値的に評価したりすることに主眼を置いているので，求解は代数的な方法ではなく数値的（近似的）な方法で十分である。

最も簡単な 1 変数の実数値関数を用いて数値求解の概要を説明しよう。図 7.5 の太実線で描かれた曲線が $f(x) = 0$ になる x を探したい関数であるとする。適当な初期値 x_0 を設定する。この点で $f(x)$ の接線を引く。接線の方程式は，

$$y = f(x_0) + f'(x_0)(x - x_0)$$

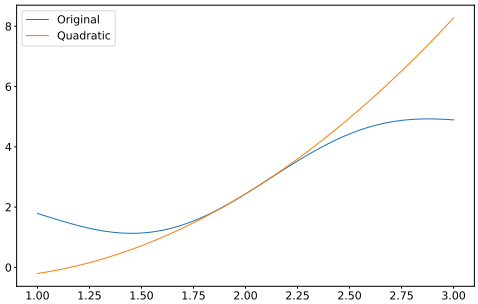


図 7.4.: 2 次の多項式近似

である。この接線の方程式の x 切片，つまり $y = 0$ となる x を探す。この点を x_1 と記すとすれば，

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

$(x_1, f(x_1))$ で $f(x)$ の接線を引いて同じことをすると，

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$$

を定義できる。図 7.5 では徐々に $f(x_*) = 0$ なる x_* に近づいている。これをさらに繰り返して，

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}, \qquad n = 1, 2, \ldots$$

を定義すると，関数 f に対する適切な仮定のもとで

$$\lim_{n \to \infty} x_n = x_*$$

となることが示される。この数値求解法を**ニュートン・ラフソン法**と呼ぶ。

本書はアルゴリズムの詳細な解説を目的としていないし，私では役者不足なので，これ以上解説を続けるのはやめておこう。数値球解法のユーザーとして必ず認識しておくべき教訓を述べるにとどめておく。

- 初期値を適切に選ばないと適切な解を見つけることができない。例えば，上

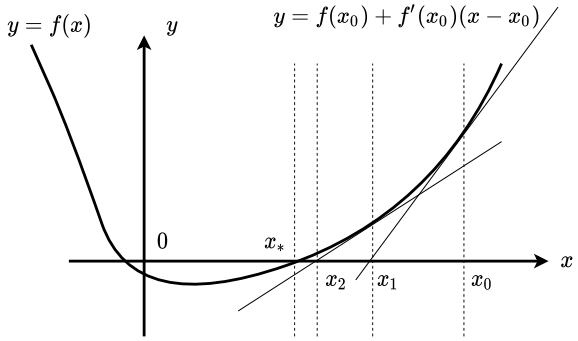


図 7.5.: ニュートン・ラフソン法

で説明した手続きでは図 7.5 の原点の左にあるもう 1 つの解は見つけられない。逆に初期値をもっと左の方に取ると x_* を見つけられない。

- 実はニュートン・ラフソン法で求めた数列 $(x_n)_{n\geq 0}$ が収束しないケースがある。したがって，必ず収束性のチェックをしなければならない [13]。
- 初期値の設定と，収束性に関するチェックは他の求解アルゴリズムを用いる場合にも忘れてはいけない。

Python で数値的に求解したいときは **SciPy** ライブラリの optimize モジュールを用いる。`newton()` と `fsolve()` を紹介しよう。

`newton()`

1 変数関数 $f : \mathbb{R} \to \mathbb{R}$ のゼロ点を探すときには `newton()` 関数を使う [14]。引数はゼロを探したい関数 `func`，初期値 `x0`，導関数 `fprime`　である。簡単な例

$$f(x) = x^2 - 3, \qquad f'(x) = 2x$$

を用いて試してみよう。

　初期値 `x0` に **NumPy** 配列を渡すと複数の初期値から出発して複数の解を見つけられる。

[13] Quora の質問「When does Newton Raphson fail?」に対する Alon Amit の解答を参照
`https://www.quora.com/When-does-Newton-Raphson-fail`（2020/07/03 閲覧）
[14] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.newton.html`

fsolve()

ベクトル値関数 $f : \mathbb{R}^n \to \mathbb{R}^n$ のゼロ点（ベクトルすべての要素がゼロになる点）を探すときには `fsolve()` 関数を使うとよい [15]。

異時点間の最適消費モデル

$$\begin{aligned} &\max \left[\log c_1 + \beta \log c_2\right] \\ &\text{subject to} \\ &\quad c_1 + \frac{c_2}{1+r} = 1000 \end{aligned}$$

を考えよう。$\beta = 0.95, r = 0.05$ とする。ラグランジアン

$$\mathcal{L} = \log c_1 + \beta \log c_2 + \lambda \left(1000 - c_1 - \frac{c_2}{1+r}\right)$$

を用いて,

$$\frac{\partial \mathcal{L}}{\partial c_1} = 0, \quad \frac{\partial \mathcal{L}}{\partial c_2} = 0, \quad \frac{\partial \mathcal{L}}{\partial \lambda} = 0$$

なる点 (c_1, c_2, λ) を探したい。最適のための 1 階条件は

$$\begin{aligned} \frac{1}{c_1} - \lambda &= 0 \\ \frac{\beta}{c_2} - \frac{\lambda}{1+r} &= 0 \\ 1000 - c_1 - \frac{c_2}{1+r} &= 0 \end{aligned}$$

である。したがって,

$$f(c_1, c_2, \lambda) = \begin{bmatrix} \frac{1}{c_1} - \lambda \\ \frac{\beta}{c_2} - \frac{\lambda}{1+r} \\ 1000 - c_1 - \frac{c_2}{1+r} \end{bmatrix} \tag{7.14}$$

がゼロになる点を探せばよい。

実はこの問題は数値計算をするまでもなく厳密に答えを見つけることができて,

[15] https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.fsolve.html

最適 $(c_1^*, c_2^*, \lambda^*)$ は

$$
\begin{aligned}
c_1^* &= \frac{1000}{1+\beta} \approx 512.821 \\
c_2^* &= \beta(1+r)c_1^* \approx 511.538 \\
\lambda^* &= 0.00195
\end{aligned}
$$

である。

さて，数値的にも解いてみよう。ヤコビアン行列というベクトル関数の微分を計算しておくと数値計算上都合がよい [16]。

$$
Df(c_1, c_2, \lambda) = \begin{bmatrix} -\frac{1}{c_1^2} & 0 & -1 \\ 0 & -\frac{\beta}{c_2^2} & -\frac{1}{1+r} \\ -1 & -\frac{1}{1+r} & 0 \end{bmatrix} \tag{7.15}
$$

準備が整ったので数値的に解いてみよう。 下のコードの重要な部分を簡単に説明しておく。

- 関数 `f` は (7.14) で定義されたもの。
- 関数 `df` は (7.15) で定義されたヤコビアン行列。
- 関数 `fsolve` に渡している `x0=[100, 100, 0.1]` は (c_1, c_2, λ) の初期値である。

7.4.4. シンボリック計算と数値計算のあわせ技

ようやく標準的なワークフローを説明する準備が整った。

Python を使って最適化問題を解く手順は次のようになる。**SymPy** と **SciPy** の使い訳を確認してほしい。

1. 最適化問題をセットアップする。（紙とペン）

[16] ベクトル関数

$$
f(x_1, x_2, x_3) = \begin{bmatrix} f_1(x_1, x_2, x_3) \\ f_2(x_1, x_2, x_3) \\ f_3(x_1, x_2, x_3) \end{bmatrix}
$$

に対して，ヤコビアン Df は次のように定義される。

$$
Df(x_1, x_2, x_3) = \begin{bmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 & \partial f_1/\partial x_3 \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 & \partial f_2/\partial x_3 \\ \partial f_3/\partial x_1 & \partial f_3/\partial x_2 & \partial f_3/\partial x_3 \end{bmatrix}
$$

本文の最適化問題では f 自体もラグランジアン $\mathcal{L}$ の微分（勾配）によって定義される。これを $\nabla\mathcal{L}$ と書く，そのヤコビアンは $\mathcal{L}$ のヘッセ行列であり，$\nabla^2\mathcal{L}$ と書くことが多い。∇ は ナブラと読む。

2. ラグランジアン $\mathcal{L}$ を定義する。（紙とペン）
3. ラグランジアンの勾配 $f = \nabla\mathcal{L}$ と，そのヤコビアン $Df = \nabla^2\mathcal{L}$ を計算する。（**SymPy**）
4. パラメータを固定して，$f = \nabla\mathcal{L}$ と $Df = \nabla^2\mathcal{L}$ を数値関数に変換する。（**SymPy**）
5. 適当な初期値と $Df = \nabla^2\mathcal{L}$ を与えて，$f = \nabla\mathcal{L}$ がゼロになる点を見つける。（**SciPy**）
6. その後，必要な数値的な分析を行う。（**NumPy**, **Pandas**）

Step 1 再び，異時点間の最適消費モデルを考えよう。効用関数は一般の CRRA 型とする。

$$\max \left[\left(\frac{c_1^{1-\theta} - 1}{1-\theta} \right) + \beta \left(\frac{c_2^{1-\theta} - 1}{1-\theta} \right) \right]$$

subject to

$$c_1 + \frac{c_2}{1+r} = 1000$$

Step 2 ラグランジアンを定義する。λ が正になるように制約条件の扱い方に気をつけよう。

$$\mathcal{L} = \left(\frac{c_1^{1-\theta} - 1}{1-\theta} \right) + \beta \left(\frac{c_2^{1-\theta} - 1}{1-\theta} \right) + \lambda \left(1000 - c_1 - \frac{c_2}{1+r} \right)$$

Step 3 微分 $\nabla\mathcal{L}$ と $\nabla^2\mathcal{L}$ を計算する。これは，

$$\nabla\mathcal{L} = 0$$

となる点を探したいのと，最適化ルーチーン $\nabla^2\mathcal{L}$ を渡すため。

まずは，$f = \nabla\mathcal{L}$ の計算。`sp.Matrix()` で行列形式にしておくと見やすい。`[L.diff(v) for v in (c1, c2, lamda)]` は `(c1, c2, lamda)` に含まれる各シンボルで微分したものをリストにまとめるという意味。

続いて $Df = \nabla^2\mathcal{L}$ の計算。これも `sp.Matrix(sp.BlockMatrix())` で行列形式にしておく。

Step 4　続いてパラメータの代入と数値関数への変換を行う。$\beta = 0.9$, $\theta = 0.3$, $r = 0.1$ という値を代入している。f の方だけ `np.squeeze()` をかぶせているのは，shape が `(3,)` の配列を入出力にとる関数にしたいから。

Step 5　最後に `fsolve()` で最適化問題を解く。

Step 6　どのような計算でもよいのだが，例えば消費の変化率を計算したいとしよう。次のようにすればよい。

パラメータを変えて最適消費点の変化を見たいときには，Step 4 と Step 5 を繰り返してデータを作っていけばよい。

例えば，次のコードは $\theta = 0.3, 1, 5$ のそれぞれに対して，r と 消費の変化率 $(c_2^* - c_1^*)/c_1^*$ の関係を計算し，プロットする。θ が大きくなると，消費に変動がなくなってくることが分かる。θ は同じ水準の消費を続けたいという消費者の選好を表しているのである。

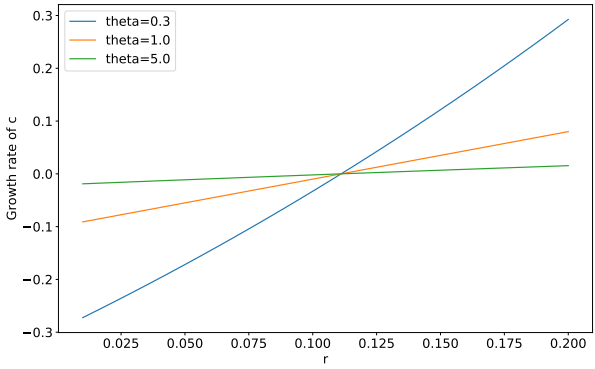


図 7.6.: θ, r と消費の変化率の関係

なお，ここで考えている最適化問題は実質利子率が変わっても生涯所得には変化がないことに注意しよう。問題 $(P2)$ をきちんと解く代わりに少し簡単な問題を解いたのである。問題 $(P2)$ を完全な形で解くことは読者の練習問題にしよう。

8. 最適成長モデル（ラムゼー・モデル）

8.1. 概要

この章では，2 期間の最適消費モデルの知識をもとに，無限期間の異時点間の最適消費問題を解く。企業が生産活動を行っている経済に最適な代表的消費者のモデルを組み込むことで，最適成長モデル（ラムゼー・モデル）を構築する。これにより，ソロー・モデルの重要な前提であった貯蓄率一定という仮定が取り除かれる。

プログラミングのセクションでは，最適成長モデルの簡便な解法を紹介する。シンボリック計算と数値最適化の手法を用いる。最後に，発展的な学習に向けて少数の参考文献を紹介する。

8.2. 不確実性のないラムゼー問題

8.2.1. セットアップ

無限期間生きる代表的消費者を考える。消費者は現在 0 期の期末にいて，1 期以降の消費の計画を立てている。1 期以降の労働所得が $y_1, y_2, \ldots$ であることを知っている。所得のうち一部を消費し残りを貯蓄する。$(t-1)$ 期末の貯蓄残高 s_{t-1} に対しては，利子所得 $r_t s_{t-1}$ を t 期に受け取る。毎期の消費から期間効用 $u(c_t)$ を得る。$u(c)$ はその期に消費していたら得たであろう効用の水準である。0 期末で消費計画を立てている消費者は将来の効用を $\beta = 1/(1+\rho)$ で割り引く。したがって，t 期の消費 c_t の現在効用は $\beta^t u(c_t)$ となる。効用関数 u には標準的な仮定を課す[1]。消費者の目的関数は割引効用の総和を最大にすることとする。

以上のセッティングを効用最大化問題として定式化すると次のようになる。

[1] $u'(\cdot) > 0, u''(\cdot) < 0.$

$$\max_{(c_t)_{t\geq 1}} U = \sum_{t=1}^{\infty} \beta^t u(c_t)$$

subject to

$$c_t + (s_t - s_{t-1}) = w_t + r_t s_{t-1}, \quad t = 1, 2, \ldots$$

s_0 : given.

ここで，w_t, r_t $(t = 1, 2, \ldots)$ と s_0 は既知のパラメータであるとする。このモデルでは将来の所得に関する完全な情報を持っている。つまり，**完全予見**を仮定する。

実はまだ 1 つ足りていない条件がある。これを見るために毎期の入出金の式（フロー予算制約, flow budget constraint）

$$c_t + (s_t - s_{t-1}) = w_t + r_t s_{t-1} \tag{8.1}$$

から生涯消費と生涯所得の関係式を導出しよう。記号を簡単にするため,

$$1 + R_t = (1 + r_1)(1 + r_2) \times \cdots \times (1 + r_t)$$

と定義する。ただし，$R_0 = 0$ とする。R_t は t 期間の累積的な利子率であり，t 期の価値を計画時点（0 期）の価値に換算する割引率である。式 (8.1) を

$$c_t + s_t = w_t + (1 + r_t) s_{t-1}$$

と変形した上で，t 期分割り戻す。

$$\frac{c_t}{1 + R_t} + \frac{s_t}{1 + R_t} = \frac{w_t}{1 + R_t} + \frac{s_{t-1}}{1 + R_{t-1}}$$

これを $t = 1, 2, \ldots$ で足し上げる。左辺と右辺に同じ式がある場合はキャンセルさ

れる。

$$\begin{aligned}
\frac{c_1}{1+R_1} + \cancel{\frac{s_1}{1+R_1}} &= \frac{w_1}{1+R_1} + \frac{s_0}{1+R_0} \\
\frac{c_2}{1+R_2} + \cancel{\frac{s_2}{1+R_2}} &= \frac{w_2}{1+R_2} + \cancel{\frac{s_1}{1+R_1}} \\
\frac{c_3}{1+R_3} + \frac{s_3}{1+R_3} &= \frac{w_3}{1+R_3} + \cancel{\frac{s_2}{1+R_2}} \\
&\vdots
\end{aligned}$$

$t = T$ まで足し上げると，次の等式を得る。

$$\sum_{t=1}^{T} \frac{c_t}{1+R_t} + \frac{s_T}{1+R_T} = \sum_{t=1}^{T} \frac{w_t}{1+R_t} + s_0$$

$T \to \infty$ の極限を取ると，

$$\sum_{t=1}^{\infty} \frac{c_t}{1+R_t} + \lim_{t\to\infty} \frac{s_t}{1+R_t} = \sum_{t=1}^{\infty} \frac{w_t}{1+R_t} + s_0 \tag{8.2}$$

を得る。無限級数は収束することを前提としている。特に，$\sum_{t=1}^{\infty} \frac{y_t}{1+R_t} < \infty$ は仮定しなければならない。このことは，

$$\lim_{t\to\infty} \frac{y_t}{1+R_t} = 0$$

を意味する。もし，利子よりも早く所得が増えるのであれば，生涯所得は無限大になり，効用最大化問題は有限の解を持たない。

仮に，無限期先の将来の貯蓄残高の割引現在価値を負にすること，つまり，

$$\lim_{t\to\infty} \frac{s_t}{1+R_t} < 0$$

が許されるとしてみよう。このとき，生涯所得よりも大きな消費ができる。つまり，

$$\sum_{t=1}^{\infty} \frac{c_t}{1+R_t} > \sum_{t=1}^{\infty} \frac{w_t}{1+R_t} + s_0$$

とできる。したがって，$\lim_{t\to\infty} \frac{s_t}{1+R_t}$ をいくらでも大きな負の値にできる場合，効

用を最大化する消費を見つけることはできない。なぜなら，消費をどんどん大きくできるので，効用も無限大に発散してしまうからだ。効用最大化問題が意味のある解を持つためには，$\lim_{t\to\infty} \frac{s_t}{1+R_t}$ は下限を持つ必要がある。通常，

$$\lim_{t\to\infty} \frac{s_t}{1+R_t} \geq 0 \tag{8.3}$$

を仮定する。(8.3) は非ポンジー・ゲーム条件（No Ponzi Game 条件）と呼ぶ[2]。非ポンジー・ゲーム条件は，借金を残して死なないという条件ではない。負債残高が利子率と同程度以上のスピードで拡大し続けることを禁止する条件である。例えば，1 期に作った負債をまったく返済せずに借り換えし続けるような行動を取れば非ポンジー・ゲーム条件に違反してしまう。

これで条件が出揃った。解くべき最大化問題は次のものである。

$$\max_{(c_t)_{t\geq 1}} U = \sum_{t=1}^{\infty} \beta^t u(c_t)$$

subject to

$$c_t + (s_t - s_{t-1}) = w_t + r_t s_{t-1}, \quad t = 1, 2, \ldots$$

$$\lim_{t\to\infty} \frac{s_t}{1+R_t} \geq 0$$

s_0 : given.

あるいは生涯所得制約で置き換えて，

$$\max_{(c_t)_{t\geq 1}} U = \sum_{t=1}^{\infty} \beta^t u(c_t) \tag{8.4}$$

subject to

$$\sum_{t=1}^{\infty} \frac{c_t}{1+R_t} \leq \sum_{t=1}^{\infty} \frac{w_t}{1+R_t} + s_0 \tag{8.5}$$

s_0 : given.

としても同じ解を得られる。

[2] ポンジー・ゲームあるいはポンジー・スキームというのは，無限連鎖講（いわゆるネズミ講）のこと。

仮定 8.1. 効用最大化問題 *(8.5), (8.5)* には解が存在して，最適消費計画の総効用は有限である。つまり，$|U| < +\infty$ が成り立つ。

効用関数 u が $u' > 0, u'' < 0$ を満たすとき，u は厳密に凹な関数である。すなわち，任意の x, y と $0 < \lambda < 1$ に対して，

$$u(\lambda x + (1-\lambda)y) > \lambda u(x) + (1-\lambda)u(y)$$

が成り立つ。この事実を使うと，解が一意であることが示せる。

命題 8.1. 効用最大化問題 *(8.5), (8.5)* の解は s_0 ごとに一意に定まる。

証明. $(c_t')_{t\geq 1}$ と $(c_t'')_{t\geq 1}$ がともに効用最大化解であるとしよう。$\sum_{t=1}^{\infty}\beta^t u(c_t') = \sum_{t=1}^{\infty}\beta^t u(c_t'')$ が成り立ち，少なくとも 1 つの $t \geq 1$ について，$c_t' \neq c_t''$ が成り立つとする。異なる 2 つの解が存在すると仮定して矛盾を導こう。

任意に $0 < \lambda < 1$ を選ぶ。$(c_t')_{t\geq 1}$ と $(c_t'')_{t\geq 1}$ はどちらも制約条件を満たすので，

$$\sum_{t=1}^{\infty}\frac{\lambda c_t'}{1+R_t} \leq \sum_{t=1}^{\infty}\frac{\lambda w_t}{1+R_t} + \lambda s_0$$

$$\sum_{t=1}^{\infty}\frac{(1-\lambda)c_t''}{1+R_t} \leq \sum_{t=1}^{\infty}\frac{(1-\lambda)w_t}{1+R_t} + (1-\lambda)s_0$$

が成り立つ。辺々足すと，

$$\sum_{t=1}^{\infty}\frac{\lambda c_t' + (1-\lambda)c_t''}{1+R_t} \leq \sum_{t=1}^{\infty}\frac{w_t}{1+R_t} + s_0$$

を得る。これは，$\left(\lambda c_t' + (1-\lambda)c_t\right)_{t\geq 1}$ で定義される消費計画が制約条件を満たすことを意味している。u が厳密に凹であるから，$c_t' \neq c_t''$ なる t については，

$$u\left(\lambda c_t' + (1-\lambda)c_t\right) > \lambda u\left(c_t'\right) + (1-\lambda)u\left(c_t''\right)$$

が成り立っている。したがって，

$$\sum_{t=1}^{\infty}\beta^t u\left(\lambda c_t' + (1-\lambda)c_t''\right) > \sum_{t=1}^{\infty}\beta^t u(c_t') = \sum_{t=1}^{\infty}\beta^t u(c_t'')$$

となる。これは，最初に解として選んだはずの $(c_t^{'})_{t\geq 1}$ と $(c_t^{''})_{t\geq 1}$ が効用を最大化できていないということだ。異なる 2 つの解が存在すると仮定すると，実はそれらは解ではなかったという矛盾にたどりついてしまう。これで，解は 1 つしかないと結論付けられる。 □

8.2.2. オイラー条件

限界代替率と相対価格が一致するところで最適消費が決まる。t 期と $(t+1)$ 期の消費の限界代替率は (8.4) から，

$$MRS_{t,t+1} = \frac{\partial U/\partial c_t}{\partial U/\partial c_{t+1}} = \frac{\beta^t u'(c_t)}{\beta^{t+1} u'(c_{t+1})} = \frac{u'(c_t)}{\beta u'(c_{t+1})}$$

相対価格に相当するものは，(8.5) の c_{t-1}, c_t に掛かる係数を見ればよい。

$$\text{相対価格} = \frac{1/(1+R_t)}{1/(1+R_{t+1})} = 1 + r_{t+1}$$

したがって，最適に選ばれた消費計画は

$$\frac{u'(c_t)}{u'(c_{t+1})} = \beta(1 + r_{t+1}) \tag{8.6}$$

を満たす。(8.6) を**オイラー条件**と呼ぶ。

期間効用関数を CRRA 型 (第 7 章を参照) と仮定すると（$\theta > 0$ を相対的危険回避度の係数とする），$u'(c) = c^{-\theta}$ と特定できるので，

$$c_{t+1} = [\beta(1 + r_{t+1})]^{1/\theta} c_t \tag{8.7}$$

が分かる。

8.2.3. 最適消費経路

式 (8.6) や (8.7) は今期の消費から次期の消費を作る方程式になっている。しかし，これだけではシミュレーションを実行することはできない。なぜなら，消費系列の初期値 c_1 は問題定式化時点では与えられておらず，効用最大化問題の解として定まるものだからである。シミュレーションを行うには，**どうにかして c_1 を定めなければならない。**

式 (8.2) をもう一度見てみよう。無限期先の将来の貯蓄残高

$$\lim_{t\to\infty}\frac{s_t}{1+R_t}$$

の割引現在価値は非ポンジー・ゲーム条件によって非負であることが分かっている。果たして，正の値を取ることはあるだろうか？

$$\lim_{t\to\infty}\frac{s_t}{1+R_t}>0\Longrightarrow\sum_{t=1}^{\infty}\frac{c_t}{1+R_t}<\sum_{t=1}^{\infty}\frac{w_t}{1+R_t}+s_0$$

であるから，生涯所得より小さな消費で済ませていることになる。これは効用最大化の観点からは望ましくない。なぜなら，貯蓄を減らして消費を増やせば必ず効用を高めることができるからだ。したがって，$\lim_{t\to\infty}\frac{s_t}{1+R_t}>0$ となるような消費計画は最適ではありえない。オイラー条件と $\lim_{t\to\infty}\frac{s_t}{1+R_t}\leq 0$ が成り立つときに消費計画が最適になることが知られている。将来貯蓄に関するこの不等式条件は**横断性条件**（transversality condition）と呼ばれている。非ポンジー・ゲーム条件から将来貯蓄が負になることは排除されているので，次の命題にまとめることができる。

命題 8.2. オイラー条件 *(8.6)* と横断性条件

$$\lim_{t\to\infty}\frac{s_t}{1+R_t}=0 \tag{8.8}$$

を満たす消費計画が最適である。

(8.8) が成り立っていれば，

$$\begin{aligned}\lim_{t\to\infty}\frac{s_t}{1+R_t}&=\lim_{t\to\infty}\frac{w_t-c_t+(1+r_t)s_{t-1}}{1+R_t}\\&=\lim_{t\to\infty}\frac{w_t-c_t}{1+R_t}+\lim_{t\to\infty}\frac{s_{t-1}}{1+R_{t-1}}\end{aligned}$$

より，

$$\lim_{t\to\infty}\frac{w_t-c_t}{1+R_t}=0$$

が成り立つ。消費の一般的な極限挙動について議論することは難しい。一例を挙げるとするならば，所得の増加率と消費と増加率が極限で一致する均整成長のケースで，利子率が成長率を平均的に上回るなら，横断性条件が成り立つ。

オイラー条件 (8.6) より，

$$\frac{1}{1+r_{t+1}} = \frac{\beta u'(c_{t+1})}{u'(c_t)}$$

したがって，

$$\begin{aligned}\frac{1}{1+R_t} &= \left(\frac{1}{1+r_1}\right)\left(\frac{1}{1+r_2}\right)\left(\frac{1}{1+r_3}\right)\times\cdots\times\left(\frac{1}{1+r_t}\right)\\ &= \left(\frac{1}{1+r_1}\right)\left(\frac{\beta u'(c_2)}{u'(c_1)}\right)\left(\frac{\beta u'(c_3)}{u'(c_2)}\right)\times\cdots\times\left(\frac{\beta u'(c_t)}{u'(c_{t-1})}\right)\\ &= \left(\frac{1}{1+r_1}\right)\frac{\beta^{t-1}u'(c_t)}{u'(c_1)}\end{aligned}$$

横断性条件は次のように書き換えることができる。

$$\lim_{t\to\infty}\beta^{t-1}u'(c_t)s_t = 0$$

$\beta^{t-1}u'(c_t)$ は効用で測った貯蓄の機会費用になっている。これは帰属計算された資産価格であり，保有資産の割引現在価値がゼロに収束することを意味する。

式 (8.6) を，未来が現在を決める方程式として眺めてみよう。

$$(1+r_1)u'(c_1) = (1+R_t)\beta^{t-1}u'(c_t)$$

この式が任意の t で成り立つので $t\to\infty$ としても成り立つはずだ。すなわち，

$$(1+r_1)u'(c_1) = \lim_{t\to\infty}(1+R_t)\beta^{t-1}u'(c_t) \tag{8.9}$$

c_1 を決定するためにこの式を使えそうである。しかし，右辺の極限が収束するかどうかは自明ではない。

予想 8.1. オイラー条件方程式 *(8.6)* と横断性条件 *(8.8)* が成り立つとき，*(8.9)* の右辺が収束して初期値 c_1 が一意に定まる。

証明. To be completed。

$$(1+r_1)u'(c_1) = \lim_{t\to\infty}(1+R_t)\beta^{t-1}u'(c_t)\frac{s_t}{s_t} = \lim_{t\to\infty}\frac{\beta^{t-1}u'(c_t)s_t}{s_t/(1+R_t)}$$

不定形の極限で，同じスピードで変化しているので，収束するはず ‥‥ 。Stolz–Cesaro の定理に類似した結果（極限がわからなくても収束することさえ言えればOK）を探す。Cauchy 列になることを示す？

□

8.3. 最適成長モデル

最適成長モデルまたはラムゼー・モデルは前節の最適消費モデルを成長モデルに組み込んだものである。

8.3.1. 企業の行動と資本蓄積

企業行動はソローモデルとほとんど同じである。総生産は

$$Y_t = F(K_{t-1}, A_{t-1}L_{t-1})$$

によって定まる。Y_t, K_t, L_t, A_t はそれぞれ，t 期の生産，資本ストック，労働需要，技術水準である。F は収穫一定の生産関数である。議論を簡単にするため，ストック変数は各期の期末に測る。すなわち，各期の期首のストックを用いて生産が行われる。

仮定 8.2. 労働 L と技術 A は以下の外生的な成長ルールに従うと仮定する。

$$L_t = (1+n)L_{t-1}$$
$$A_t = (1+g)A_{t-1}$$

n は人口成長率，g は技術進歩率である。

仮定 8.3. 労働需要は完全に非弾力的であり，いつでも，どんな要素価格でも労働供給と需要が一致するとしよう [3]。

利潤最大化条件により，

$$\frac{\partial F}{\partial K}(K_{t-1}, A_{t-1}L_{t-1}) = r_t + \delta$$
$$\frac{\partial F}{\partial L}(K_{t-1}, A_{t-1}L_{t-1}) = w_t$$

[3] 多くのマクロモデルの基礎となっているリアル・ビジネスサイクル・モデルは労働供給が賃金に依存して決まるように拡張したものだ。

が成り立つ。ただし，r_t, w_t, δ はそれぞれ実質利子率，実質賃金，資本減耗率である。

資本蓄積の方程式は，

$$K_t - K_{t-1} = Y_t - C_t - \delta K_{t-1}$$

となる。ここで，C_t は総消費であり，政府のない閉鎖経済を考えているので，$Y_t - C_t$ は t 期の粗投資と一致する。t 気に起こる資本ストックの変化 $K_t - K_{t-1}$ は粗投資から資本減耗を差し引いたものになる。資本蓄積方程式を $A_t L_t$ で除すと

$$\frac{K_t}{A_t L_t} = \frac{Y_t}{A_t L_t} - \frac{C_t}{A_t L_t} + (1-\delta)\frac{K_{t-1}}{A_t L_t}$$

ここで，効率労働あたりの資本，生産，消費

$$k_t = \frac{K_t}{A_t L_t}, \quad y_t = f(k_{t-1}) = \frac{Y_t}{A_{t-1} L_{t-1}}, \quad c_t = \frac{C_t}{A_{t-1} L_{t-1}}$$

を導入すると，資本蓄積方程式は，

$$k_t = \frac{f(k_{t-1}) - c_t + (1-\delta)k_{t-1}}{(1+g)(1+n)} \tag{8.10}$$

と書き換えることができる [4]。

ソロー・モデルとは違って投資の水準が内生的に決まるため，これだけではダイナミクスを決定することはできない。これには家計の行動を記述する必要がある。

8.3.2. 家計

家計の行動はラムゼー問題の解によって定まるとする。ここでは，経済のすべての経済主体が 1 つの家計に属していると考えよう。個々の経済主体は t 期に

$$\bar{c}_t = \frac{C_t}{L_{t-1}}$$

を消費し，その消費によって $u(\bar{c}_t)$ の当期効用を得る [5]。家計全体の効用は時間方向には割引現在効用の総和，空間方向には総人口についての総和を取ったものとな

[4] 自分に対するコメント：フロー変数の効率労働あたりの量を $A_{t-1}L_{t-1}$ で除している。ソローモデルの章でもそうした方がよかったのかもしれない。あるいはストックを期首で測る。チェックする。

[5] t 期に生まれる個人は t 期には消費しないとしよう。

る。したがって，家計は目的関数

$$\sum_{t=1}^{\infty} \beta^{t-1} L_{t-1} u(\bar{c}_t)$$

を最大化したい。$L_t = (1+n)^t L_0$ であることに注意すると，

$$\sum_{t=1}^{\infty} \{\beta(1+n)\}^{t-1} u(\bar{c}_t)$$

を最大化することと同じである。ここで 1 つ仮定を置く必要がある。

仮定 8.4. 割引因子 β と人口成長率 n は以下の関係を満たす。

$$\beta < 1 + n.$$

各期の予算制約は次のように書ける。S_t は家計（マクロ経済）の総貯蓄残高である。

$$\bar{c}_t L_{t-1} + (S_t - S_{t-1}) = r_t S_{t-1} + w_t L_{t-1}$$

1 人あたりに換算すると（$\bar{s}_t = S_t / L_{t-1}$ と置く），

$$\bar{c}_t + \frac{S_t}{L_{t-1}} = \frac{(1+r_t) S_{t-1}}{L_{t-1}} + w_t$$

$$\bar{c}_t + \bar{s}_t = \left(\frac{1+r_t}{1+n}\right) \bar{s}_{t-1} + w_t$$

累積的な利子率を

$$1 + R_t = \left(\frac{1+r_1}{1+n}\right)\left(\frac{1+r_2}{1+n}\right) \times \cdots \times \left(\frac{1+r_t}{1+n}\right)$$

と定義すれば，前節の問題と同じように扱える。割引現在価値を $t = T$ まで足し上げると，次の等式を得る。

$$\sum_{t=1}^{T} \frac{\bar{c}_t}{1+R_t} + \frac{\bar{s}_T}{1+R_T} = \sum_{t=1}^{T} \frac{w_t}{1+R_t} + \bar{s}_0$$

$T \to \infty$ の極限を取ると，

$$\sum_{t=1}^{\infty} \frac{\bar{c}_t}{1+R_t} + \lim_{t\to\infty} \frac{\bar{s}_t}{1+R_t} = \sum_{t=1}^{\infty} \frac{w_t}{1+R_t} + \bar{s}_0 \tag{8.11}$$

非ポンジー・ゲーム条件を課すと，次の生涯の制約条件を得る。

$$\sum_{t=1}^{\infty} \frac{\bar{c}_t}{1+R_t} \leq \sum_{t=1}^{\infty} \frac{w_t}{1+R_t} + \bar{s}_0$$

したがって，家計は次の制約条件付き最大化問題を解いて消費計画を立てる。

$$\max \sum_{t=1}^{\infty} \{\beta(1+n)\}^{t-1} u(\bar{c}_t)$$

subject to

$$\sum_{t=1}^{\infty} \frac{\bar{c}_t}{1+R_t} \leq \sum_{t=1}^{\infty} \frac{w_t}{1+R_t} + \bar{s}_0$$

この問題のオイラー条件は

$$\frac{\{\beta(1+n)\}^{t} u'(\bar{c}_t)}{\{\beta(1+n)\}^{t+1} u'(\bar{c}_{t+1})} = \frac{1/(1+R_t)}{1/(1+R_{t+1})}$$

すなわち，

$$\frac{u'(\bar{c}_t)}{u'(\bar{c}_{t+1})} = \beta(1+r_{t+1}) \tag{8.12}$$

である。

ここで，CRRA 型の効用関数を仮定しよう。相対的危険回避度のパラメータを $\theta > 0$ とすると，

$$u'(c) = c^{-\theta}$$

である。したがって，(8.12) は次のように変形できる。

$$\frac{\bar{c}_{t+1}}{\bar{c}_t} = \{\beta(1+r_{t+1})\}^{1/\theta}$$

$c_t = \bar{c}_t / A_{t-1}$ と $A_t = (1+g)A_{t-1}$ に注意して効率労働あたりの式に変形すると，

$$\frac{\bar{c}_{t+1}/A_t}{\bar{c}_t/A_t} = \{\beta(1+r_{t+1})\}^{1/\theta}$$

したがって，

$$\frac{c_{t+1}}{c_t} = \frac{\{\beta(1+r_{t+1})\}^{1/\theta}}{1+g} \tag{8.13}$$

を得る。このオイラー条件と横断性条件が同時に満たされているときに最適である。

8.3.3. 最適経路

最適成長モデルの均衡は次の 3 つを同時に満たしている状態である。

- 企業は毎期の利潤最大化条件を満たしている，かつ，
- 家計は生涯消費を最適に選んでいる。
- 財市場，資本市場の需給が一致している。

定理 8.1. 最適成長モデルの均衡（あるいは，最適経路，*optimal path*）は次の動学方程式で特徴付けられる。

$$k_t = \frac{f(k_{t-1}) - c_t + (1-\delta)k_{t-1}}{(1+g)(1+n)} \tag{8.14}$$

$$\frac{c_{t+1}}{c_t} = \frac{[\beta\{f'(k_t)+1-\delta\}]^{1/\theta}}{1+g} \tag{8.15}$$

k_0 は所与で，c_1 は横断性条件

$$\lim_{t\to\infty} \frac{k_t}{1+R_t} = 0$$

によって定まる。

証明. 式 (8.10), (8.13) および資本の限界生産性が利子率と一致することを用いている。資本市場精算条件より $S_t = K_t$ が成り立っている。 □

もし，初期値 (k_0, c_1) が決まれば，以後の変数値を順番に計算していくことができる。

- (k_0, c_1) が決まれば，(8.14) より k_1 が決まる。さらに，(8.14) より，c_2 が決まる。
- (k_1, c_2) が決まったので，(8.14) より k_2 が決まる。さらに，(8.14) より，c_3 が決まる。
- 以下同様。

しかし，(8.14), (8.15) は初期値 $k_0 = K_0/(A_0L_0)$ のみ与えられており，消費の初期値 c_1 は横断性条件によって定められなければならない。これが最適成長モデル，ひいてはマクロ経済モデルをシミュレーションすることの難しさである。

8.3.4. 長期均衡

この問題の設定では，定常状態 (k^*, c^*) に収束する経路が最適経路になる。なお，(k^*, c^*) は次の方程式を満たす。

$$k^* = \frac{f(k^*) - c^* + (1-\delta)k^*}{(1+g)(1+n)}$$
$$1 = \frac{[\beta\{f'(k^*) + 1 - \delta\}]^{1/\theta}}{1+g}.$$

パラメータ（f, δ, β, g, θ）が与えられれば，コンピュータを用いて数値的に解 (k^*, c^*) を求められる。

最適成長モデルの最適経路 $\{(k_t, c_{t+1})\}_{t=0}^{\infty}$ は，

$$\lim_{t\to\infty} k_t = k^*, \qquad \lim_{t\to\infty} c_t = c^*$$

を満たす。効率労働あたりの変数が収束したあとの状態を長期均衡と呼ぶ。経済は十分長い時間がたったあとには長期均衡に収束するので，収束後の状態が経済の長期的な動向を示していると考えるのである。

マクロレベルの集計変数については，長期では

$$K_t \approx A_t L_t k^*, \quad C_t \approx A_{t-1} L_{t-1} c^*$$

などが成り立つ。また，1 人あたりの変数については

$$\frac{K_t}{L_t} \approx A_t k^*, \quad \frac{C_t}{L_{t-1}} \approx A_{t-1} c^*$$

などが成り立っている。重要なのは成長率であり，次のような性質が成り立つ。

定理 8.2. 最適成長モデルの長期均衡は以下の性質を持つ。

- 効率労働あたりの変数は定数である。
- *1* 人あたりの変数は g の率で成長する。
- 集計量は $g + n + gn$ の率で成長する。

系 8.1. 最適成長モデルの長期均衡では，次のことが言える。

- 実質利子率は定数である。
- 賃金は g の率で成長する。

つまり，長期均衡における成長率に注目する限り，最適成長モデルとソロー・モデルでは同じ結論が成り立っている。

8.4. プログラミング

(8.14), (8.15) のような式があれば，シミュレーションのコードを自力で書けそうな気がするだろうか。

実は，このモデルのシミュレーションは，これまでに学んできた方法では実行できない。なぜなら，2 変数の動学方程式に対して初期値が 1 つ（k_0）しか与えられていないからだ。消費の「初期値」c_1 は横断性条件を用いなければ決まらない。つまり，適当に c_1 を選んでシミュレーションして $k_t \to k^*$ と $c_t \to c^*$ が成り立っていることを確認して，ようやくそれが最適経路であるとわかる。手当たりしだいに c_1 を変化させてシミュレーションすることもできるが効率がよくないので，ここでは 蓮見 (2020) で紹介されている方法を採用しよう。

動学方程式は

$$k_t = \frac{f(k_{t-1}) - c_t + (1-\delta)k_{t-1}}{(1+g)(1+n)} \qquad c_{t+1} = \frac{c_t \left[\beta \left\{f'(k_t) + 1 - \delta\right\}\right]^{1/\theta}}{1+g}$$

で与えられている。$t = 0$ に対して，

$$k_1 = \frac{f(k_0) - c_1 + (1-\delta)k_0}{(1+g)(1+n)} \qquad c_2 = \frac{c_1 \left[\beta \left\{f'(k_1) + 1 - \delta\right\}\right]^{1/\theta}}{1+g}$$

$t = 1, 2, \ldots, T-1$ に対して，

$$k_2 = \frac{f(k_1) - c_2 + (1-\delta)k_1}{(1+g)(1+n)} \qquad c_3 = \frac{c_2\left[\beta\left\{f'(k_2)+1-\delta\right\}\right]^{1/\theta}}{1+g}$$

$$k_3 = \frac{f(k_2) - c_3 + (1-\delta)k_2}{(1+g)(1+n)} \qquad c_4 = \frac{c_3\left[\beta\left\{f'(k_3)+1-\delta\right\}\right]^{1/\theta}}{1+g}$$

$$\vdots$$

$$k_T = \frac{f(k_{T-1}) - c_T + (1-\delta)k_{T-1}}{(1+g)(1+n)} \qquad c_{T+1} = \frac{c_T\left[\beta\left\{f'(k_T)+1-\delta\right\}\right]^{1/\theta}}{1+g}$$

$t = 0$ から $t = T-1$ までの $2T$ 本の方程式がある。変数は 2 種類，$2T+2$ 個含まれている：

- $k_0, k_1, \ldots k_T$ で $T+1$ 個
- $c_1, c_2, \ldots, c_{T+1}$ で $T+1$ 個

初期条件 k_0 と，終端条件 $c_{T+1} = c^*$ を代入することで，未知変数の数を $2T$ 個に減らしてやれば，最適経路を近似的に導くことができる。$k_T = k^*$ を代入しないことに注意せよ [6]。

準備

必要なライブラリと関数をインポートする。

```
import numpy as np
from scipy.optimize import fsolve
import matplotlib.pyplot as plt
import sympy as sp
sp.init_printing()
```

動学方程式 (8.10), (8.15) を定義するために必要最小限のシンボルを導入しよう。

```
k1, k0, c1, c0, g, n = sp.symbols("k1 k0 c1 c0 g n")
alpha, beta, delta, theta = sp.symbols("alpha beta delta theta")
```

[6] これをやってしまうと解けなくなる。

生産関数はコブ＝ダグラス型とする。

```
f = sp.Lambda(k0, k0**alpha)
f(k0)
```

$$k_0^{\alpha}$$

動学方程式

動学方程式 (8.14), (8.13) において，ゼロになるべき関数を定義する。

```
EK = k1 - (f(k0) - c0 + (1 - delta) * k0) / (1 + g) / (1 + n)
EK
```

$$k_1 - \frac{-c_0 + k_0 (1 - \delta) + k_0^{\alpha}}{(g + 1) (n + 1)}$$

および,

```
EC = c1 - c0 * (beta * (f(k0).diff(k0) + 1 - delta)) ** (1 / theta) / (1 + g)
EC
```

$$-\frac{c_0 \left(\beta \left(\frac{\alpha k_0^{\alpha}}{k_0} - \delta + 1\right)\right)^{\frac{1}{\theta}}}{g + 1} + c_1$$

定常状態 (k^*, c^*) を求めるためには，次のベクトル関数がゼロになる点を探せばよい。

```
E = sp.Matrix([EK, EC]).subs({k1: k0, c1: c0})
E
```

$$\begin{bmatrix} k_0 - \frac{-c_0+k_0(1-\delta)+k_0^{\alpha}}{(g+1)(n+1)} \\ -\frac{c_0\left(\beta\left(\frac{\alpha k_0^{\alpha}}{k_0}-\delta+1\right)\right)^{\frac{1}{\theta}}}{g+1} + c_0 \end{bmatrix}$$

定常状態を数値的に求める

以下のパラメータを用いる。

```
params = {
```

```
    alpha: 0.33,
    delta: 0.03,
    g: 0.02,
    n: 0.01,
    theta: 0.8,
    beta: 0.98
}
```

数値関数に変換する。これには，sp.lambdify() を用いる。ヤコビ行列も求めておこう。

```
E_lam = sp.lambdify([[k0, c0]], np.squeeze(E.subs(params)))
J_lam = sp.lambdify([[k0, c0]], E.jacobian([k0, c0]).subs(params))
```

fsolve() を用いて定常状態を求める。

```
ss = fsolve(func=E_lam, x0=[4, 4], fprime=J_lam)
ss
```

```
[10.87371171  1.54328611]
```

定常状態の値は $(k^*, c^*) \approx (10.874, 1.543)$ であることが分かる。

最適経路を求める

たくさんのシンボル変数を作るには次のようにすればよい[7]。シンボルのタプルが生成される。負数のインデックスを使うと「最後から n 個目」のような指定ができることを思い出そう。

```
T = 80
c = sp.symbols(f"c[:{T+2}]")
k = sp.symbols(f"k[:{T+1}]")

c[:3], c[-3:]
```

$$((c[0],\ c[1],\ c[2]),\ (c[79],\ c[80],\ c[81]))$$

最適経路上では次の行列値関数がゼロになる。数が多いので最初の 3 行のみ出力する。

[7] c[0] は使わない。

```
eqm = sp.Matrix([[EK.subs({k0: k[i], k1: k[i+1], c0: c[i+1], c1: c[i+2]}),
                  EC.subs({k0: k[i], k1: k[i+1], c0: c[i+1], c1: c[i+2]})]
                  for i in range(T)])
eqm[:3, :]
```

$$\begin{bmatrix} k[1] - \frac{-c[1]+k[0](1-\delta)+k[0]^{\alpha}}{(g+1)(n+1)} & -\frac{c[1]\left(\beta\left(\frac{\alpha k[0]^{\alpha}}{k[0]}-\delta+1\right)\right)^{\frac{1}{\theta}}}{g+1}+c[2] \\ k[2] - \frac{-c[2]+k[1](1-\delta)+k[1]^{\alpha}}{(g+1)(n+1)} & -\frac{c[2]\left(\beta\left(\frac{\alpha k[1]^{\alpha}}{k[1]}-\delta+1\right)\right)^{\frac{1}{\theta}}}{g+1}+c[3] \\ k[3] - \frac{-c[3]+k[2](1-\delta)+k[2]^{\alpha}}{(g+1)(n+1)} & -\frac{c[3]\left(\beta\left(\frac{\alpha k[2]^{\alpha}}{k[2]}-\delta+1\right)\right)^{\frac{1}{\theta}}}{g+1}+c[4] \end{bmatrix}$$

列ベクトルに変換しておこう。.T で転置行列を作ってからリシェイプする。

```
eqm_col = eqm.T.reshape(2*T, 1)
eqm_col[:3, :], eqm_col[T:T+3, :]
```

$$\left(\begin{bmatrix} k[1] - \frac{-c[1]+k[0](1-\delta)+k[0]^{\alpha}}{(g+1)(n+1)} \\ k[2] - \frac{-c[2]+k[1](1-\delta)+k[1]^{\alpha}}{(g+1)(n+1)} \\ k[3] - \frac{-c[3]+k[2](1-\delta)+k[2]^{\alpha}}{(g+1)(n+1)} \end{bmatrix}, \begin{bmatrix} -\frac{c[1]\left(\beta\left(\frac{\alpha k[0]^{\alpha}}{k[0]}-\delta+1\right)\right)^{\frac{1}{\theta}}}{g+1}+c[2] \\ -\frac{c[2]\left(\beta\left(\frac{\alpha k[1]^{\alpha}}{k[1]}-\delta+1\right)\right)^{\frac{1}{\theta}}}{g+1}+c[3] \\ -\frac{c[3]\left(\beta\left(\frac{\alpha k[2]^{\alpha}}{k[2]}-\delta+1\right)\right)^{\frac{1}{\theta}}}{g+1}+c[4] \end{bmatrix}\right)$$

初期値 k_0 と終端条件 $c_{T+1} = c^*$ を与えて数値関数に変換し，解を求める。ヤコビ行列の自動計算は時間がかかるので省略しよう [8]。

```
k_init = 1.0

eqm_param = eqm_col.subs(params).subs({k[0]: k_init, c[-1]: ss[1]})
eqm_param = np.squeeze(eqm_param)
eqm_num = sp.lambdify([[*k[1:], *c[1:-1]]], eqm_param)
solution = fsolve(eqm_num, x0=np.ones(2*T))
```

得られた結果には k_0 と c_{T+1} が含まれていないので，これを補う。

```
k_sol = np.r_[k_init, solution[:T]]
c_sol = np.r_[solution[T:], ss[1]]
```

[8] 均衡を規定する方程式の構造に注目すれば効率よくヤコビ行列を定義できる。

結果をプロットしたものが図 8.1 である。次のコードで生成した。

```
plt.quiver(k_sol[:-1], c_sol[:-1],
           k_sol[1:] - k_sol[:-1],
           c_sol[1:] - c_sol[:-1],
           scale_units='xy', angles='xy', scale=1)

plt.xlabel(r"$k_{t}$")
plt.ylabel(r"$c_{t+1}$")
plt.show()
```

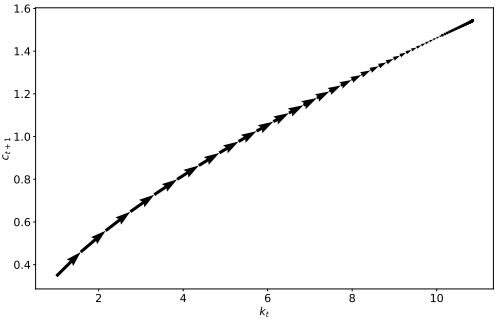


図 8.1.: 最適成長経路

関数にまとめる

試行錯誤がある程度落ち着いたら一連の作業を関数にまとめておこう。2 つの関数を作る。

- `solve_ss`: 資本蓄積方程式およびオイラー条件のシンボリックな表現，モデルのパラメータを引数に取り，定常状態 (k^*, c^*) の値を出力する関数
- `solve_optimal_growth`: 資本蓄積方程式およびオイラー条件のシンボリックな表現，k の初期値，c の終端条件，モデルのパラメータを引数に取り，最適経路 $\left(\{k_t\}_{t=0}^{T}, \{c_t\}_{t=1}^{T+1}\right)$ を出力する関数

solve_ss 定常状態を求める関数は次のように定義する。

```
def solve_ss(eqm_k, eqm_c, param, x0):
    c, k = sp.symbols("c, k")
    eqm = sp.Matrix([eqm_k.subs({k0: k, k1: k, c0: c, c1: c}),
                     eqm_c.subs({k0: k, k1: k, c0: c, c1: c})])

    eqm_lam = sp.lambdify([[k, c]], np.squeeze(eqm.subs(param)))
    J_lam = sp.lambdify([[k, c]], eqm.jacobian([k, c]).subs(param))

    ss = fsolve(func=eqm_lam, x0=x0, fprime=J_lam)
    return ss
```

仮引数は次のような意味である。

- eqm_k: 均衡条件 (8.10) のシンボリックな表現
- eqm_c: 均衡条件 (8.15) のシンボリックな表現
- param: モデルのパラメータを格納する辞書
- x0: 解探索の初期値

本体部分は上で説明したものとほとんど同じなので，解説は省略する。

solve_optimal_growth 均衡経路を求める関数を次のように定義する。

```
def solve_optimal_growth(eqm_k, eqm_c, k_init, c_final, param, T):

    c = sp.symbols(f"c[:{T+2}]")
    k = sp.symbols(f"k[:{T+1}]")
    eqm = sp.Matrix([[eqm_k.subs({k0: k[i], k1: k[i+1], c0: c[i+1], c1: c[i+2]}),
                      eqm_c.subs({k0: k[i], k1: k[i+1], c0: c[i+1], c1: c[i+2]})]
                      for i in range(T)])
    eqm_col = eqm.T.reshape(2*T, 1)

    eqm_param = eqm_col.subs(param).subs({k[0]: k_init, c[-1]: c_final})
    eqm_param = np.squeeze(eqm_param)

    eqm_num = sp.lambdify([[*k[1:], *c[1:-1]]], eqm_param)
    solution = fsolve(eqm_num, x0=np.ones(2*T))

    k_sol = np.r_[k_init, solution[:T]]
    c_sol = np.r_[solution[T:], c_final]
```

```
    return (k_sol, c_sol)
```

仮引数は次のような意味である。

- eqm_k: 均衡条件 (8.10) のシンボリックな表現
- eqm_c: 均衡条件 (8.15) のシンボリックな表現
- k_init: k の初期値
- c_final: 消費の終端条件。定常状態を事前に求めておく。
- param: モデルのパラメータを格納する辞書
- T: シミュレーションの期間

使用例　定義した関数は，次のように使用する（結果は図 8.2）。$k_0 > k^*$ から出発しても同じ定常状態に収束していく様子を確認できる。パラメータを変更するなど，各自で色々と試してほしい。

```
ss1 = solve_ss(EK, EC, params, x0=[4, 4])
k_sol, c_sol = solve_optimal_growth(EK, EC, 30.0, ss1[1], params, 50)

plt.quiver(k_sol[:-1], c_sol[:-1],
           k_sol[1:] - k_sol[:-1],
           c_sol[1:] - c_sol[:-1],
           scale_units='xy', angles='xy', scale=1)

plt.xlabel(r"$k_{t}$")
plt.ylabel(r"$c_{t+1}$")
plt.show()
```

比較動学分析

長期均衡にある経済に対してパラメータ変化のショックが生じると，その後，経済はどのように変化するだろうか。この手の分析を比較動学分析と呼ぶ。

例えば，次のようなシナリオを考えよう。

- 元のパラメータに対する定常状態にある経済に対して，
- 技術進歩を加速するイノベーションが起こり，g が 0.02 から 0.03 に上昇した。

やることは 3 つ。

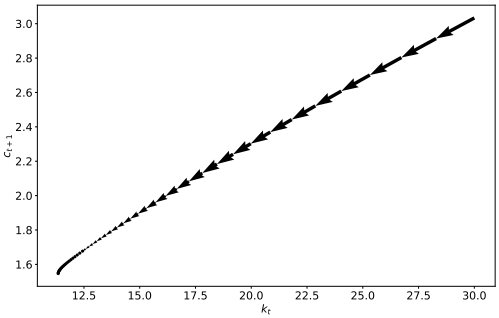


図 8.2.: 最適成長経路

1. 変化前のパラメータに対応する定常状態を求める。（ss1 として実行済み）
2. 変化後のパラメータに対応する定常状態を求める。
3. 変化後のパラメータに対して最適経路を求める。

パラメータの更新と新しい定常状態は次のように計算できる。

```
params.update({g: 0.03})
ss2 = solve_ss(EK, EC, params, x0=[4, 4])
```

パラメータ変化後の最適経路を導出する。変化前の定常状態を初期値として付加しておこう。

```
sol2 = solve_optimal_growth(EK, EC, ss1[0], ss2[1], params, 80)
k_sol2 = np.r_[ss1[0], sol2[0]]
c_sol2 = np.r_[ss1[1], sol2[1]]
```

最後に可視化する（結果は図 8.3）。パラメータ変化のタイミングで効率労働あたりの消費がジャンプしていることに気がつくだろう。将来の経済状況に関する予想が修正されると，フロー変数には大きな変化が起こる。ソロー・モデルにはない，ラムゼー・モデルの特徴である。

```
plt.quiver(k_sol2[:-1], c_sol2[:-1],
           k_sol2[1:] - k_sol2[:-1],
           c_sol2[1:] - c_sol2[:-1],
           scale_units='xy', angles='xy', scale=1)
```

```
5
plt.xlabel(r"$k_{t}$")
plt.ylabel(r"$c_{t+1}$")
plt.show()
```

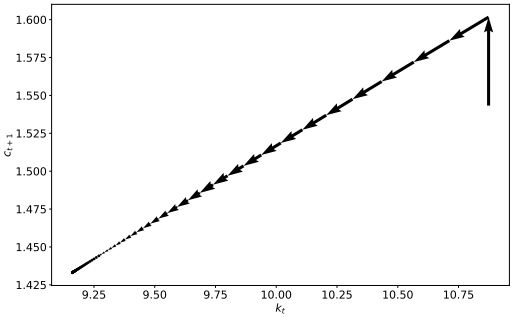


図 8.3.: 移行過程

8.5. まとめと注意

この章では，最適成長モデルは均衡経路が近似的に満たすべき連立方程式を用いて均衡経路を数値的に求めた。ソローモデルのシミュレーションとは違って，すべての時点の解が一斉に求まるような解法になっている。最適成長モデルは将来の情報に依存して現在の行動が定まるようなモデルだからだ。未来が現在を定める性質を持つモデルを，フォワード・ルッキングなモデルと呼ぶ。

フォワード・ルッキングなモデルの解法はこの章で紹介した方法がベストという訳ではない。うまく行かない例を見つけるには，さきほどのコードを T を小さくして再実行してみてほしい。定常状態周りの挙動が安定しないことに気づくと思う。定常状態に有限ステップでたどり着くような点はそれ自身を除いて存在しないので，$c_{T+1} = c^*$ が成り立つなら必ず $k_T \neq k^*$ になってしまうのだ。

この章で用いた解法は理論的に高度なことが要求されないという大きな利点があるものの，定常状態の周りで不安定になるので，少し使いにくいケースもある。マクロ経済分析の多くは定常状態の周りの状況を分析することを主眼としているか

らだ。定常状態の周りの分析をきちんと実行するためには，別の手法を用いる方がよい。非線形性と真っ向勝負するなら動的計画法を用いる (Stachurski, 2009)。関数解析の入門的な知識が必要になるのでじっくり取り組もう。非線形性が強くない場合や，おおよその傾向を掴むことが目的であれば，定常状態周りの線形近似を用いることもできる (加藤, 2006)。こちらは線形代数学のやや高度な技術が駆使される。いずれの方向に進むにせよ，蓮見 (2020) はよい出発点だろう。

A. ギリシャ文字

経済学ではギリシャ文字を多用する。例えば，価格は英語で price だから p を使うと，利潤 profit に対して p が使えないので代わりに p に対応するギリシャ文字の π を使うといった具合である。カタカナ表記ではなく英語綴りを覚えておくと，対応関係を理解しやすくなる。

$A\ \alpha$	$B\ \beta$	$\Gamma\ \gamma$	$\Delta\ \delta$
alpha アルファ	beta ベータ	gamma ガンマ	delta デルタ
$E\ \epsilon/\varepsilon$	$Z\ \zeta$	$H\ \eta$	$\Theta\ \theta$
epsilon イプシロン	zeta ゼータ	eta イータ	theta シータ
$I\ \iota$	$K\ \kappa$	$\Lambda\ \lambda$	$M\ \mu$
iota イオタ	kappa カッパ	lambda ラムダ	mu ミュー
$N\ \nu$	$\Xi\ \xi$	$O\ o$	$\Pi\ \pi$
nu ニュー	xi グザイ/クサイ	omicron オミクロン	pi パイ
$P\ \rho$	$\Sigma\ \sigma/\varsigma$	$T\ \tau$	$\Upsilon\ \upsilon$
rho ロー	sigma シグマ	tau タウ	upsilon ウプシロン/ユプシロン
$\Phi\ \phi/\varphi$	$X\ \chi$	$\Psi\ \psi$	$\Omega\ \omega$
phi ファイ	chi カイ	psi プサイ	omega オメガ

参考文献

北川源四郎 (2005)『時系列解析入門』，岩波書店.

久保川達也・国友直人 (2016)『統計学』，東京大学出版会.

McKinney，Wes・瀬戸山雅人・小林儀匡・滝口開資 (2018)『Python によるデータ分析入門 第 2 版』，オライリー・ジャパン.

R Core Team (2019) *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, URL: `https://www.R-project.org/`.

Stachurski, John (2009) *Economic Dynamics: Theory and Computation*: MIT Press.

Ushey, Kevin, JJ Allaire, and Yuan Tang (2019) *reticulate: Interface to 'Python'*, URL: `https://CRAN.R-project.org/package=reticulate`, R package version 1.12.

Xie, Yihui (2015) *Dynamic Documents with R and knitr*, Boca Raton, Florida: Chapman and Hall/CRC, 2nd edition, URL: `https://yihui.name/knitr/`, ISBN 978-1498716963.

——— (2019) *knitr: A General-Purpose Package for Dynamic Report Generation in R*, URL: `https://yihui.name/knitr/`, R package version 1.23.

沖本竜義 (2010)『経済・ファイナンスデータの計量時系列分析』，朝倉書店.

加藤涼 (2006)『現代マクロ経済学講義—動学的一般均衡モデル入門』，東洋経済新報.

蓮見亮 (2020)『動学マクロ経済学へのいざない』，日本評論社.