目次

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

$$g_t^{xy} = \log x_t y_t - \log x_{t-1} y_{t-1} = (\log x_t - \log x_{t-1}) + \left(\log y_t - \log y_{t-1}\right) = g_t^x + g_t^y$$

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。

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. $g^{x/y} = g^x - g^y$

[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。

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。

$$MPK = \frac{\partial Y}{\partial K} = \alpha A K^{\alpha-1} L^{1-\alpha} = \alpha \frac{A K^{\alpha} L^{1-\alpha}}{K} = \alpha \frac{Y}{K}$$

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，

$$\alpha \frac{Y}{K} = r, \qquad (1-\alpha)\frac{Y}{L} = w$$

[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。

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，

$$\log Y_t - \log Y_{t-1} = (\log A_t - \log A_{t-1}) + \alpha (\log K_t - \log K_{t-1}) + (1-\alpha)(\log L_t - \log L_{t-1})$$

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。

[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%

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。

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。

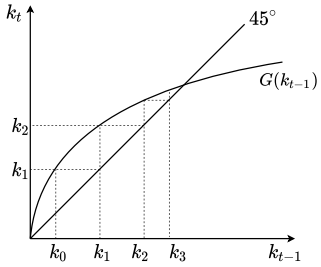


図 6.1: $k_t = G(k_{t-1})$

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。

$$\left[(1+g)(1+n) - (1-\delta)\right] k^* = sf(k^*)$$

$$\left(g + n + \delta + gn\right) k^* = sf(k^*)$$

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）。

$$k^* = k^*\left(\delta, g, n, s\right)$$

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。

$$Y_t = A_t L_t y^* = A_t L_t f(k^*)$$
$$\frac{Y_t}{L_t} = A_t f(k^*)$$

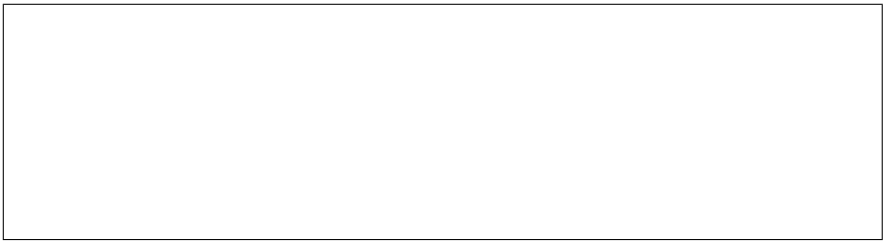
この関係から Y や Y/L の成長率を計算することができる。

$$\frac{Y_t - Y_{t-1}}{Y_{t-1}} = \frac{A_t L_t f(k^*) - A_{t-1} L_{t-1} f(k^*)}{A_{t-1} L_{t-1} f(k^*)} = g + n + gn$$

$$\frac{(Y_t/L_t) - (Y_{t-1}/L_{t-1})}{(Y_{t-1}/L_{t-1})} = \frac{A_t f(k^*) - A_{t-1} f(k^*)}{A_{t-1} f(k^*)} = g$$

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。

[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

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。

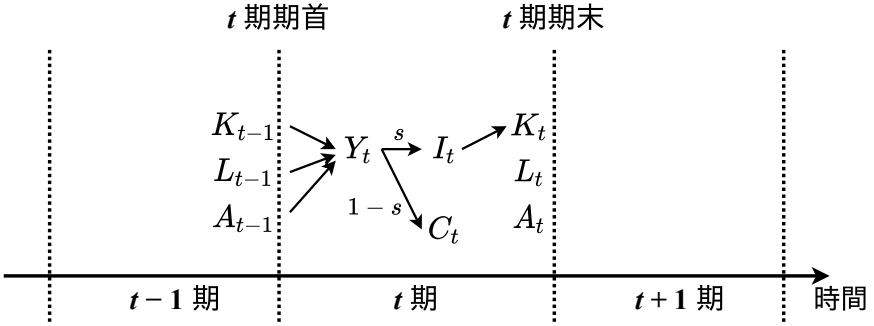


図 6.2: ソローモデルの変数間の関係

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。

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。

$$
\begin{aligned}
Y &= AK^{\alpha}L^{1-\alpha} \\
&= K^{\alpha}\left(A^{\frac{1}{1-\alpha}}L\right)^{1-\alpha}
\end{aligned}
$$

$B = A^{\frac{1}{1-\alpha}}$ 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，

[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。

- 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。

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
```

[8] R や Julia とは異なり，Python では明示的な `return` 文が必要である。

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)
```

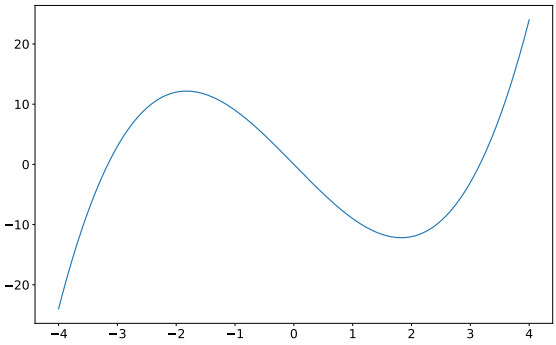


図 6.3: $f(x) = x^3 - 10x$ のグラフ

```
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。

```
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]。

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
```

[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。

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
```

[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

```
def fun(x):
    a = 0            # (2) a==0
    return x + a

print(fun(0))        # (3) a==0
```

```
0
```

```
print(a)             # (4) a==3
```

```
3
```

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。

[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。

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

$$F(K,L,A) = K^{\alpha}(AL)^{1-\alpha}$$

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()
```

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
```

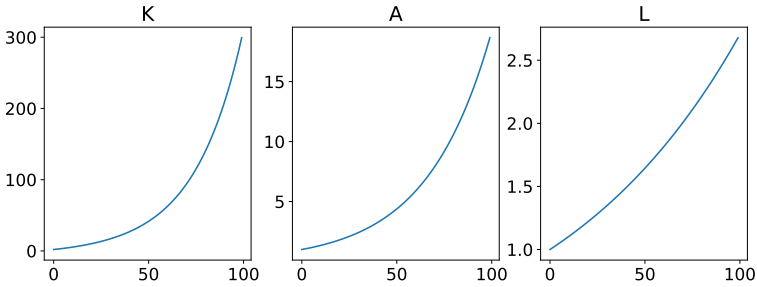


図 6.4: ソロー・モデルのシミュレーション結果

```
sol.k.plot(ax=ax)
plt.show()
```

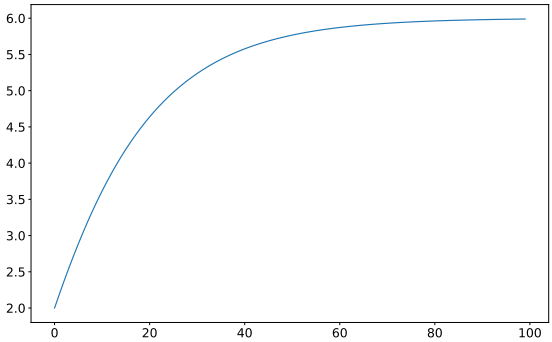


図 6.5: 効率労働あたりの資本

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，

- 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()
```

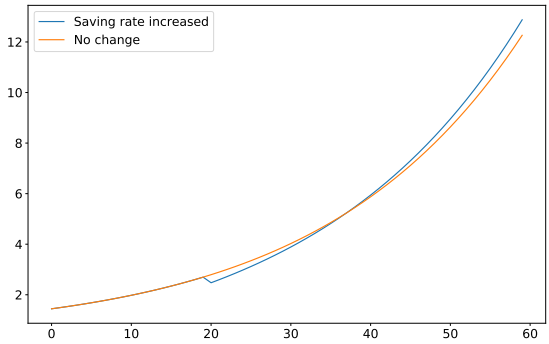


図 6.6: 貯蓄率の変化

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。