統計分析ソフトウェア R の使用法

西山　茂

2001 年 9 月

概 要

R はフリーで配布されている高水準の統計分析ソフトウェアです。**R** の基本設計は世界中で広く利用されている **S** や **Splus** を元にしています。本資料は **R** や **S** のマニュアルを通読する前に、最低限知っておいたほうが良い点をまとめたものです。ソフトウェアは利用しながら覚えていくものです。マニュアルを読み通すまで利用できないのは問題です。経済分析を進める際に本資料が役立つなら幸いです。

1　はじめに—起動と終了—

R を起動すると、以下のようなウィンドウが表示されます。これを「コンソール画面」と呼びます。赤字の「＞」はプロンプト記号であり、この後にコマンドを入力していくことにより、色々なデータの入力、計算、統計分析、グラフ作成などを行わせることができます。

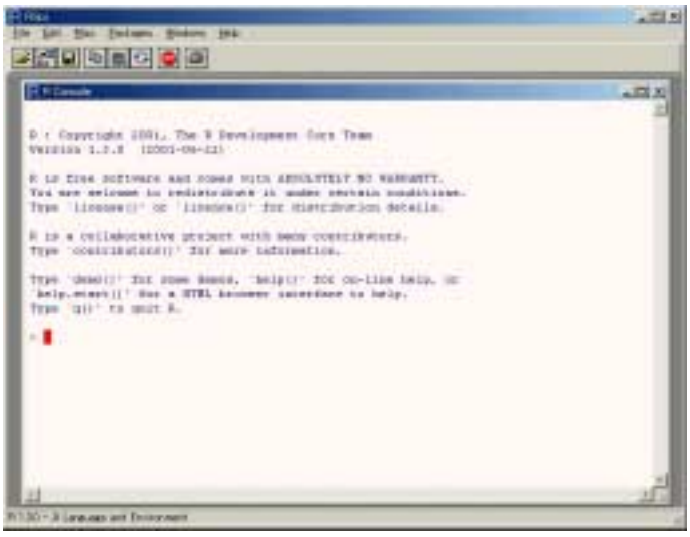

図 1: **R** の起動画面

それでは、終了しましょう。終了するには、第 2 図のように、[File] メニューから [Exit] を選

んでもいいし、あるいは、コマンド `q( )` を打ち込んでも終了できます。

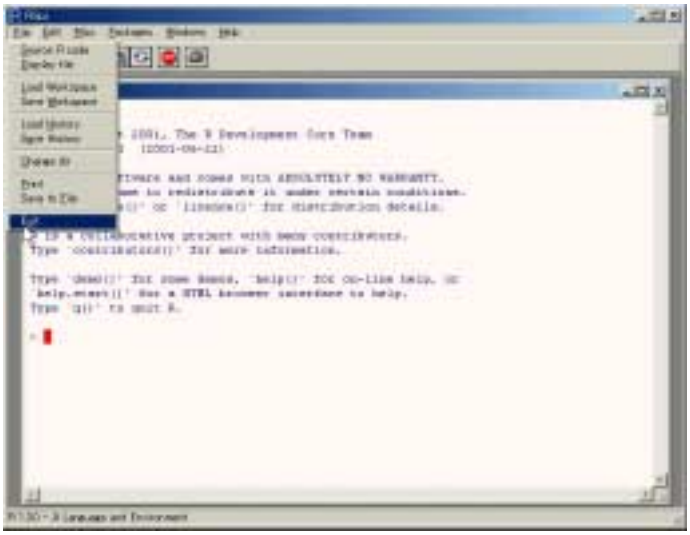

図 2: **R** の終了

保存した変数の内容など作業領域を保存するかどうか、確認するメッセージが現れますから、通常の場合は、「はい」をクリックします。これで終了です。

2 基本的なコマンドと変数への保存

この節では、**R** のコンソール画面にいくつか簡単なコマンドを入力してみます。**R** には多くのコマンドが用意されており、一度にすべてを記憶することは到底無理ですが、日常の作業は少数の基本的なコマンドを身に付けることで不自由なく実行できます。それでは、**R** を起動してください。

まず、最初にカレントディレクトリーを作業用ディレクトリーに変更しておきましょう。統計分析は、通常、データファイルからデータを入力したり、分析結果や描画したグラフを保存したりします。これらのファイルが保存される場所が「作業用ディレクトリー」です。起動直後は、ソフトウェア R が格納されているディレクトリー（たとえば、`C:\Program Files\R\rw1030`）がカレントディレクトリーになっていますから、データを読んだり、結果を保存したりするのに不便です。

作業用ディレクトリーを変更するには、[File] メニューから [Change Dir] を選んで行います。たとえば、`c:\datadir` に移りたいときには、このとおりディレクトリー名を指定して OK を押します。

それでは簡単な計算から始めましょう。

2.1 簡単な計算

コマンドプロンプト「＞」の後に、

```
7+3
```

と打ち込んでリターンキーを押してください。[1]

10 という答えが表示されたはずです。順に、7-3、7*3、7/3、7^3 と入力してみましょう。回答が得られましたか？次のような回答が得られるはずです。[2]

```
> 7+3
[1] 10
> 7-3
[1] 4
> 7*3
[1] 21
> 7/3
[1] 2.333333
> 7^3
[1] 343
```

乗算は「×」の代わりに「＊」を、除算は「÷」の代わりに「／」を使用します。また、最後の「＾」はべき乗を表す演算記号であることが分かりますね。

次に、演算の優先順位、括弧の使用を確認しておきましょう。

```
> 7*(1+2)
[1] 21
> 2*6+2
[1] 14
> 6/3*2
[1] 4
> 2^3*4
[1] 32
```

このとおり、数学で通常想定される演算順位が **R** に組み込まれていることが分かります。

[1] リターンキーを押すことで、**R** がコマンドを処理します。
[2] 回答の最初に示されている [1] については今は気にしないでください。

2.2 変数への値の保存

この辺りで値の保存について知っておきましょう。同じ値を反復して何度も使いたい場合があります。たとえば、同じ幅で高さが異なる長方形の面積を計算してみます。高さは 2.5 に固定しておいて、幅を 3 から 10 まで変化させながら、それぞれの幅に対する面積を計算しましょう。単純にやれば、

```
> 2.5*3
[1] 7.5
> 2.5*4
[1] 10
> 2.5*5
[1] 12.5
> 2.5*6
[1] 15
> 2.5*7
[1] 17.5
> 2.5*8
[1] 20
> 2.5*9
[1] 22.5
> 2.5*10
[1] 25
```

このように、その都度、高さ 2.5 をキーボードから入れなければなりません。それより

```
> h<-2.5
```

と「＜−」を使って、変数 h に値 2.5 を保存しておきましょう。記号「＜−」は代入記号で、右辺の値を左辺の変数に保存することを意味します。[3]そうすると、次のように、いつでも変数名 h によって保存された値を使うことができます。

```
> h<-2.5
> h*3
[1] 7.5
> h*4
[1] 10
> h*5
```

[3] ＜ − で結ばれる式を代入式といいます。右辺と左辺を入れ替え、− ＞と逆にしても同じ結果を得られます。

```
[1] 12.5
> h*6
[1] 15
```

一定の値は変数に保存して使ったほうが、その後の計算が容易になるし、入力ミスを防ぐことにもつながります。

2.3 ベクトルとマトリックス

2.3.1 ベクトルの作り方：c( )、seq( )、rep( )

高さ 2.5 は h と入力すればいつでも利用できるようになりました。しかし、複数の幅の値に対する面積を求める作業が大して楽になっているわけではありません。それは、それぞれの幅の値に対して、h*幅という式をいちいち入力しなければならないからです。

幅がとりうる値をワンセットにして、一つの変数に保存することができます。複数の値をワンセットにまとめるには、下のように関数 c( ) を使います。[4]

```
> w<-c(3,4,5,6,7,8,9,10)
> w
[1]  3  4  5  6  7  8  9 10
> h*w
[1]  7.5 10.0 12.5 15.0 17.5 20.0 22.5 25.0
```

すべての幅の値に対して面積を求める作業がずいぶん楽になりましたね。変数 w は 3 から 10 までの値をワンセットにしたもので、このような変数をベクトルといいます。それに対して、変数 h のように 1 個の値をもつものをスカラーと呼んでいます。統計分析とはデータを分析することです。データは多くの値から構成されるものですから、日常的にベクトルを扱うことになります。上の式は、スカラーとベクトルの乗算になっているわけです。数学での取り扱いのとおり、すべてのベクトルの成分にスカラーがかけられることになります。

それでは、幅の値の範囲として 1 から 100 までの整数をとってみましょう。これらの値をベクトルとして保存しようとすれば、

```
> w<-c(1,2,3,4,5,...,99,100)
```

のようにしなければなりません。これも結構大変です。

このような場合は関数 seq( ) を使います。たとえば 1 から 100 までの整数値をベクトルにしたいのであれば、

[4] c( ) の c は concatenate（つなぐ）の意味です。

```
> w<-seq(1,100)
> w
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
 [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
 [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
 [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
 [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
 [91]  91  92  93  94  95  96  97  98  99 100
```

変数名を入力すれば常に保存されている内容が表示されます。[5]各行の先頭にある [1] や [19] は、その行が何番目の成分から開始されているかを示しています。[6]

他にも、

```
> v<-seq(100,110)
> v
 [1] 100 101 102 103 104 105 106 107 108 109 110
> seq(100,110,2)
[1] 100 102 104 106 108 110
> seq(1,10)/10
 [1] 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
> seq(1,2,length=20)
 [1] 1.000000 1.052632 1.105263 1.157895 1.210526 1.263158 1.315789 1.368421
 [9] 1.421053 1.473684 1.526316 1.578947 1.631579 1.684211 1.736842 1.789474
[17] 1.842105 1.894737 1.947368 2.000000
```

こんなパターンが利用できます。一番上の例のように、簡単に seq(s,e) とすれば、数値 s から数値 e までが 1 刻みでベクトルにされます。刻み幅を 2 にしたいのであれば、二番目のように、seq(s,e,step) とします。三番目の例は小数の刻み幅にしたいときの方法。最後の例は、ある数値からある数値までの範囲を何等分かしたい時に利用します。

もう一つベクトルの作り方があります。それは関数 rep( ) です。たとえば rep(2,4) と入力してください。ベクトル (2,2,2,2) が作れることが分かります。関数 rep( ) の使い方を他にも示しておきましょう。

```
> rep(2,4)
[1] 2 2 2 2
> rep(c(0,1),3)
[1] 0 1 0 1 0 1
> rep(c(-1,0,1),c(1,2,3))
[1] -1  0  0  1  1  1
```

ベクトル自体を反復することもできます（二番目の例）。面白いのは、反復回数をベクトルで指定するパターンです。こうすると第 1 成分は最初の回数、第 2 成分は次の回数、第 3 成分は三番目の回数、という具合に、各成分の反復回数を別々に指定することができます。

[5] うっかり 100 でなく 1000 を入力してしまったときは、長さが 1000 のベクトルになりますから、内容表示が延々と続くことになります。

[6] これまでは結果が短い表示ですみましたから [1] だけでしたね。

2.3.2 ベクトルの成分の取り出し方

ベクトルの特定の成分が必要なことがあります。ここで

```
> w[3]
```

と入力してください。3と表示されるはずです。これはベクトルwの第3成分が3であることを示しています。

一般にベクトルの中から必要な成分を取り出すときには、w[3]のように[ ]の中に必要な成分を指定します。上のように成分の番号を指定するのは、もっとも単純なケースです。

成分の取り出し方には、他にもいろいろなパターンがあります。

```
> w<-seq(1,100)
> w[3]
[1] 3
> w[11:20]
 [1] 11 12 13 14 15 16 17 18 19 20
> w[c(1,3,7,10)]
[1]  1  3  7 10
> w[seq(50,60)]
 [1] 50 51 52 53 54 55 56 57 58 59 60
> w[seq(1,20,2)]
 [1]  1  3  5  7  9 11 13 15 17 19
```

これで分かるように、何番目から何番目までというパターンで成分を抽出するには「:」を使います。w[11:20]は11番目から20番目までの成分になります。c( )を使ってベクトルで成分番号を指定することも役に立ちます。seq( )関数も使えます。上の例のようにseq(1,20,2)とすると、1からステップ幅を2にして20までの成分を指定することになります。

面白いのは、下の第3図のように、たとえばw>50という不等式を[ ]内に入れる方法です。

50を超える成分がすべて表示されています。一体、w>50という不等式は、どの成分番号を指定したことになっているのでしょうか？試しに、w>50を入力してください。

TRUEやFALSEが表示されました。この二つは数値ではなく、論理値として区分されるものです。数値は量を表すもので大小がありますが、たとえば「Wの第1成分は50を超えているか」という問いには「超えている」か「超えていない」の二通りの答えしかありません。不等式は、超えているなら「真」、超えていないなら「偽」ということになります。式w>50は一つ一つの成分ごとに、50を超えているかどうかという不等式を評価して、不等式が真ならTRUE、不等式が偽ならFALSEという値を表示することになったのです。

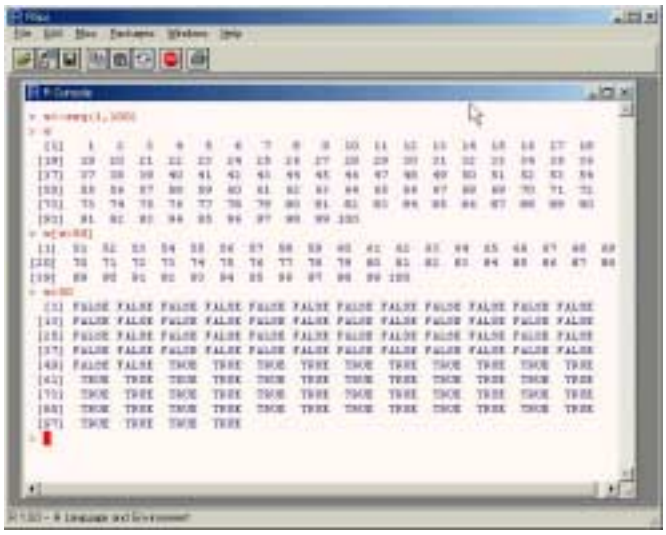

図 3: 論理式による成分抽出

答えが数値ではなく真か偽という論理値になる式を論理式といいます。論理式でベクトルの成分を指定すると TRUE に対応した成分だけが抽出されるということです。

論理式を構成する演算子をここで掲げておきましょう。

記号	意味
<	左辺は右辺より小
>	左辺は右辺より大
<=	左辺は右辺以下
>=	左辺は右辺以上
==	左辺と右辺は同じ値
!=	左辺と右辺は異なる
&	かつ、and
|	あるいは、or

たとえば、次のような抽出ができます。

```
> w<-seq(1,10)
> w
 [1]  1  2  3  4  5  6  7  8  9 10
> w[w>=5 & w<=7]
[1] 5 6 7
```

このように、ベクトルの成分を取り出す方法には色々なものがあります。データを分析する場合、データの値に応じて、色々なやり方で一部分を抽出することがよく行われます。そのよ

うな際には、この節で説明したことを思い出して、必要な成分だけを取り出すようにしてください。

もちろん特定の成分を抽出するのとは逆に、特定の成分だけを除くこともできます。たとえば、

```
> w<-seq(1,10)
> w
 [1]  1  2  3  4  5  6  7  8  9 10
> w[-2]
[1]  1  3  4  5  6  7  8  9 10
> w[-c(1,3,5)]
[1]  2  4  6  7  8  9 10
> w[-2:9]
Error: only 0's may mix with negative subscripts
> w[-(2:9)]
[1]  1 10
```

このように除きたい成分をベクトルによって複数指定することも可能です。ただ、2:9 のように範囲を指定する時は、括弧を付けて負値を示すマイナス記号と混同されないようにしなければなりません。[7]

2.3.3 ベクトル同士の計算、そしてマトリックス

ベクトルの成分すべてを一律に何倍かしたり、特定の数で割るという計算はスカラー×ベクトルというパターンで行えます。これは前に述べました。では、全ての成分に特定の数値を加えたり、特定の数値を引いたりする計算はどうすればいいのでしょうか？これもスカラーをかけたり、スカラーで割ったりする場合と同じで、単にベクトルに加減したい値を +、− しておけばいいだけです。

```
 w<-seq(1,10)
> w
 [1]  1  2  3  4  5  6  7  8  9 10
> w+2
 [1]  3  4  5  6  7  8  9 10 11 12
> w-5
 [1] -4 -3 -2 -1  0  1  2  3  4  5
```

[7] 三番目の事例は「成分番号が− 2 番目から 9 番目までを削除」という意味にとられます。最初の成分は 1 番目です。

では、ベクトル同士で計算するとどんな結果になるでしょうか？数学での取り扱いとは多少違っています。基本的には、成分同士の計算が行われると覚えておいてください。たとえば次のようになります。

```
> x<-c(1,2,3,4,5); y<-c(1,3,5,7,9)
> x+y
[1]  2  5  8 11 14
> x-y
[1]  0 -1 -2 -3 -4
> x*y
[1]  1  6 15 28 45
> x/y
[1] 1.0000000 0.6666667 0.6000000 0.5714286 0.5555556
```

上の例でも使われていますが、コマンドの後ろに「；」を付けるときがあります。これは 1 行に複数の文を入力する場合に用いる文の区切り記号です。

この辺りで、ベクトルを複数並べたマトリックスを導入しましょう。**R** では、変数の型として、一般に**ベクトル**、**行列（マトリックス）**、**配列（アレイ）**、**データフレーム**、**リスト**を利用することができます。このうち、ベクトルは既に説明しました。いくつかの数値がワンセットにまとめられたものです。[8]数値が 1 列に並んだものがベクトルとすれば、マトリックスは数値が縦横に並んだものです。ベクトルは何個の数値がならんだものかによってベクトルの長さを求めることができますが、マトリックスは縦と横の長さを測ることで寸法を求めます。

マトリックスは、今の段階では使うことはないだろうと思うかもしれませんが、実は縦横に並んだ数値の集まりは、データ分析では日常的に登場します。たとえば、表を思い浮かべてください。消費関数を推定するために、第 1 列目に名目 GDP、第 2 列目に GDP デフレーター、第 3 列目に名目家計消費、第 4 列目に消費デフレーターを並べて、データの表を作成したりします。1 行目の変数名を別にすれば、残りは数値のマトリックスそのものです。実際には、経済データはマトリックスよりも扱いやすい型であるデータフレームにして分析を始めるのが定石ですが、データフレームはマトリックスと大変よく似ています。まずマトリックスに慣れておくと、あとが楽になります。

ベクトルの作成には主に `c( )` 関数を利用しましたが、マトリックスでこれに対応するのは `matrix( )` 関数です。利用する際は、`matrix(ベクトル,nrow=行数,ncol=列数,byrow=F or T)` のように、それぞれ値を指定します。たとえば、範囲を指定する簡便法 `1:6` を利用してベクトルを作り、

```
> matrix(1:6,nrow=2)
     [,1] [,2] [,3]
[1,]    1    3    5
```

[8]実は、数値だけではなく、文字列をいくつかまとめてワンセットにすることもできます。

```
[2,]    2    4    6
> matrix(1:6,nrow=3)
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

このようにすると、デフォールトでは、[9]データとして指定された数値ベクトルが縦方向に並べられ、行列を構成することになります。上の例では nrow の値を行数として指定しています。成分の個数が分かっていますから行数が決まれば、列数は自動的に決まります。ですから、nrow と ncol を両方とも指定する必要はあまりありません。では、byrow を指定したらどうなるでしょう。これは論理値を指定します。つまり T か F を指定します。デフォールトでは F になっています。上の例でこれを T にしてみます。

```
> matrix(1:6,nrow=2,byrow=T)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
> matrix(1:6,nrow=3,byrow=T)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

このように、指定した数値が横方向に並んでいくことになります。[10]

マトリックスといえば、ベクトルを含めて、色々な計算が可能なことは線形代数学で勉強したことと思います。いくつか試してみましょう。

```
> A<-matrix(1:6,nrow=3); B<-matrix(c(1,3,5,7,9,11),nrow=3)
> A
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> B
     [,1] [,2]
[1,]    1    7
```

[9]何も指定しなければ自動的に指定される値。

[10]最初の 1 から最後の 6 までを順番に並べるのですが、読んだ順に、縦に並べていくか、横に並べていくかの違いを byrow で指定するわけです。

```
[2,]    3    9
[3,]    5   11
> A+B
     [,1] [,2]
[1,]    2   11
[2,]    5   14
[3,]    8   17
> A-B
     [,1] [,2]
[1,]    0   -3
[2,]   -1   -4
[3,]   -2   -5
> A*B
     [,1] [,2]
[1,]    1   28
[2,]    6   45
[3,]   15   66
> A/B
          [,1]      [,2]
[1,] 1.0000000 0.5714286
[2,] 0.6666667 0.5555556
[3,] 0.6000000 0.5454545
```

行列同士の加算、減算は数学での取り扱いのとおりです。成分同士の計算になります。マトリックス同士の乗算、除算は線形代数では面倒でしたね。**R** では簡単です。成分同士の積、商を求めるだけです。これはベクトル同士の計算でもいえることでした。上には示しませんでしたが、スカラーとマトリックスの計算もベクトルの場合と同じです。自分で試してみてください。それでは、線形代数のようにマトリックスをかける時はどうすればいいのでしょうか。それには演算子「%*%」を使います。記号三つで一つの演算子であることに注意してください。

それでは A と B をかけてみましょう。

```
> A %*% B
Error in A %*% B : non-conformable arguments
```

エラーが出てしまいました。どうやら『引数が計算に合っていない』[11]ようです。線形代数で二つの行列をかける場合は、前の行列の列数と後の行列の行数が等しくなければいけませんでしたね。今の場合、A も B も 3×2 行列ですからマトリックス乗算は計算不能なのです。しかし、AB' なら計算できるはずです。B の転置行列は t(B) で表します。やってみましょう。

[11]関数や計算式の対象を引数（ヒキスウ）と呼びます。

```
> A %*% t(B)
     [,1] [,2] [,3]
[1,]   29   39   49
[2,]   37   51   65
[3,]   45   63   81
> t(A) %*% B
     [,1] [,2]
[1,]   22   58
[2,]   49  139
```

3×2 行列と 2×3 行列の積は 3×3 行列、2×3 行列と 3×2 行列の積は 2×2 行列になることに注意してください。

では行列 A にベクトル c(1,2) をかけてみましょう。ベクトルも行数が 1、あるいは列数が 1 の特別なマトリックスですから、乗算は演算子「%*%」を使います。

```
> x<-c(1,2)
> A %*% x
     [,1]
[1,]    9
[2,]   12
[3,]   15
```

次に、c(1,2,3) を A の前からかけてみましょう。

```
> y<-c(1,2,3)
> y %*% A
     [,1] [,2]
[1,]   14   32
```

単にベクトルと言えば列ベクトルを指すことが普通です。しかし、すぐ上の例では行列 A の前から y をかけています。これは y を長さ 3 の行ベクトル、つまり 1×3 行列とみなしていることになります。ですから式で表現すれば $y'A$ を計算したわけです。このように **R** では、一々、ベクトルを転置しなくとも自動的に行ベクトルか列ベクトルかを解釈して計算結果を出すようになっているのです。

では次に実際のデータを利用しながらマトリックスを活用してみましょう。

2.4 データフレームの活用

データフレームの前にマトリックスを利用して実際のデータを扱ってみましょう。原データは 1990 年度から 2000 年度までの実質 GDP と消費支出です。

多くの数値を入力するには通常はデータを別のファイルに保存しておいて、**R** からそのデータファイルを読みこむ方法をとります。実際、これから使用するデータはファイル gdpcons.txt を読んで入力しました。[12]しかし、ファイルの読み込み、書き出しについては、あとで解説することにして、本節では既にデータが入力され、多少の変更を加えたあとの段階から出発しましょう。

まず、どのような変数[13]にデータが保存されているかを調べてみましょう。それには関数 ls( ) を使います。

```
> ls()
[1] "gdpcons"      "last.warning" "year"
```

このように gdpcons と year が作成されています。[14]gdpcons の内容を表示してみましょう。内容を表示するには変数名を打ち込めばいいのでしたね。

```
> gdpcons
           GDP     CONS
 [1,] 312712.7 174382.7
 [2,] 321490.5 177074.9
 [3,] 331710.7 184799.3
 [4,] 339823.8 189292.0
 [5,] 353436.2 194237.4
 [6,] 368184.1 201627.8
 [7,] 379895.7 209050.0
 [8,] 399442.3 217356.6
 [9,] 424657.3 229129.5
[10,] 445468.8 238784.9
[11,] 469780.5 248840.1
[12,] 481660.7 256905.6
[13,] 483375.6 261560.2
[14,] 485498.4 266385.2
[15,] 490730.7 272342.2
[16,] 502794.3 277906.5
[17,] 520053.8 284766.8
[18,] 521315.1 281393.7
[19,] 518380.7 285094.0
[20,] 525695.8 289454.2
[21,] 530312.8 288981.1
> year
 [1] 1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994
[16] 1995 1996 1997 1998 1999 2000
```

[12] gdpcons.txt の内容は最後のページに添付しています。
[13] 変数や後で解説する自作の関数など作成されたものの総称として**オブジェクト**という名が使われます。
[14] 逆に、不要になったオブジェクトを削除するには関数 rm( ) を使います。

gdpcons はマトリックス、year はベクトルであることが分かります。それぞれのサイズを求めておきましょう。[15]それには、関数 dim( )、length( ) を使います。

```
> dim(gdpcons)
[1] 21  2
> dim(year)
NULL
> length(year)
[1] 21
```

このようにマトリックスの行数と列数は dim( ) 関数で、ベクトルの長さは length( ) で求めることができます。gdpcons の行数と year の長さは等しくなってますね。

ところで、gdpcons の第 1 列には GDP、第 2 列には CONS と名称が入っています。これは、それぞれ何のデータかを示すものですが、何もない単なるマトリックスよりは、ずっと見やすい形になっています。このようにマトリクスの行、列には名称をつけることが可能です。名称を調べるには

```
> dimnames(gdpcons)
[[1]]
NULL

[[2]]
[1] "GDP"  "CONS"
```

このように dimnames( ) 関数を使います。[[1]] とあるのは行につけられた名称、[[2]] とあるのは列につけられた名称を示します。[16]行に付けられた名称は NULL、つまり名称がまだつけられていないことがわかります。列には、第 1 列に “GDP”、第 2 列に “CONS” という名が付けられていることを教えてくれています。gdpcons の行にはやはり年度が表示されたほうが見やすいですから、行の名称に年度の数字を入れましょう。それには

```
> dimnames(gdpcons)[[1]]<-year
> gdpcons
          GDP     CONS
1980 312712.7 174382.7
1981 321490.5 177074.9
1982 331710.7 184799.3
1983 339823.8 189292.0
```

[15]ベクトルの長さ、マトリックスの行数、列数をサイズ、寸法などと言ったりします。
[16]名称は全体が**リスト**の型になっています。

```
1984 353436.2 194237.4
1985 368184.1 201627.8
1986 379895.7 209050.0
1987 399442.3 217356.6
1988 424657.3 229129.5
1989 445468.8 238784.9
1990 469780.5 248840.1
1991 481660.7 256905.6
1992 483375.6 261560.2
1993 485498.4 266385.2
1994 490730.7 272342.2
1995 502794.3 277906.5
1996 520053.8 284766.8
1997 521315.1 281393.7
1998 518380.7 285094.0
1999 525695.8 289454.2
2000 530312.8 288981.1
```

これでできます。dimnames( ) の使い方に注意してください。[17]

さて、gdpcons の中の実質 GDP、あるいは民間最終消費支出だけを取り出して、増加率を計算したり、グラフを描いたり、平均値を求めたりしたいことがあります。たとえば、GDP の系列だけを取り出すということは、マトリックス gdpcons の第 1 列を抽出することです。

一般にマトリックスの成分を取り出すにはベクトルと同じで記号 [ ] を使います。たとえば、A[1,1] とすれば 1 行 1 列成分 A(1,1) が取り出せます。今は、第 1 列すべての成分を表示したいわけです。このような場合も含めて、いくつかマトリックスの成分の取り出し方を例示しておきましょう。一番下で GDP 系列全体を取り出しています。

```
> gdpcons[1,1]
[1] 312712.7
> gdpcons[1:3,1]
    1980     1981     1982
312712.7 321490.5 331710.7
> gdpcons[,1]
    1980     1981     1982     1983     1984     1985     1986     1987
312712.7 321490.5 331710.7 339823.8 353436.2 368184.1 379895.7 399442.3
    1988     1989     1990     1991     1992     1993     1994     1995
424657.3 445468.8 469780.5 481660.7 483375.6 485498.4 490730.7 502794.3
    1996     1997     1998     1999     2000
520053.8 521315.1 518380.7 525695.8 530312.8
```

上の最初の例では 1 行 1 列成分 gdpcons(1,1) を、二番目の例では行指定として 1:3 と範囲

[17] 記号 [[ ]] が出てきていますが、これはリストと呼ばれる変数の第何番目の成分を取り出すかに使われる記号です。リストについては後の 2.5 節で簡単に紹介し詳しい説明は省きます。

を指定していますから第 1 行から第 3 行の部分の 1 列目が表示されます。一番下の例では、行番号が指定されてませんね。ブランクのまま、コンマが続いています。これは「すべての行」を取り出すという意味になります。このパターンは頻繁に使いますから覚えておいて下さい。もし、gdpcons の第 3 行全体を取り出したいなら、gdpcons[3,] とすればいいわけです。

行と列の名前を指定することによっても成分を取り出すことはできます。ただし、名称は文字列ですから、"GDP"や"1980"のように引用符で囲んでください。[18]

```
> gdpcons["2000","CONS"]
[1] 288981.1
> gdpcons["1995",]
     GDP     CONS
502794.3 277906.5
```

このように行、列番号を指定しても、その名称を指定しても、同じ結果が得られます。

さて、これまでは扱っている変数がマトリックスかデータフレームかを意識することなく説明を続けてきました。そろそろ両者の違いについて話しておきましょう。

マトリックスは全ての成分が数値か、文字列でなければなりません。この制約はベクトルにも当てはまります。数値と文字列が混在していてはベクトルやマトリックスという型はとれないのです。しかし、コンソール画面から x<-c("a",1,2,3,"b") というコマンドを打ち込んでもエラーにはなりません。文字列と数値が混在できると思うかもしれませんね。確かめてみましょう。

```
> x<-c("a",1,2,3,"b")
> x
[1] "a" "1" "2" "3" "b"
```

このように全ての成分が自動的に文字列として扱われていることが確認できます。この事情はマトリックスについても当てはまります。成分のどれかが文字列になっているとマトリックス全ての成分が文字列に自動的に変換されてしまうのです。つまり、マトリックスでは実質 GDP や民間最終消費支出のような数値に混ぜて文字列型のデータを保存することはできない、ということになります。

しかし、「その年度がバブル景気にあたっていたかどうか」という情報を文字列で与えたいことがあります。たとえば、バブル景気に該当する年度には"bubble"、その前の年度には"b"、その後の年度には"a"という文字列を与えましょう。バブル景気をいつからいつまでと考えるかについては議論もあるでしょうが、ここでは 1987 年度から地価がピークアウトした 1991 年度までとしておきましょう。したがって、年度区分は 1980 年度から 1986 年度までが"b"、1987 年度から 1991 年度までが"bubble"、それ以降が"a"という値になります。これを保存するベクトルの名称を id"としましょう。

[18] **R** では大きく区分して数値と文字列を扱います。文字列は二重引用符で囲んで値を示します。

```
> id<-c("b","b","b","b","b","b","b","bubble","bubble","bubble","bubble","bubble",
+ "a","a","a","a","a","a","a","a","a")
> length(id)
[1] 21
```

最後にすべての年度について区分を入力したかどうかを確かめるため、ベクトルの長さが 21 になっているかどうかを調べています。このように長いベクトルを入力する時には、途中でリターンキーを押して改行してもかまいません。コマンドがまだ完結していないと **R** の方が判断すれば、プロンプト記号が「＋」に変わって、続きを促します。

データは一つの表になっていたほうが見やすいので、いま作った id を gdpcons の第 3 列として加えることにします。マトリックスに行や列を追加するときには、関数 rbind( )、cbind( ) を利用します。いまは列を追加しますから cbind( ) を使います。しかし、

```
> cbind(gdpcons,id)
     gdp         cons        id
1980 "312712.7" "174382.7" "b"
1981 "321490.5" "177074.9" "b"
1982 "331710.7" "184799.3" "b"
1983 "339823.8" "189292"   "b"
1984 "353436.2" "194237.4" "b"
1985 "368184.1" "201627.8" "b"
1986 "379895.7" "209050"   "b"
1987 "399442.3" "217356.6" "bubble"
1988 "424657.3" "229129.5" "bubble"
1989 "445468.8" "238784.9" "bubble"
1990 "469780.5" "248840.1" "bubble"
1991 "481660.7" "256905.6" "bubble"
1992 "483375.6" "261560.2" "a"
1993 "485498.4" "266385.2" "a"
1994 "490730.7" "272342.2" "a"
1995 "502794.3" "277906.5" "a"
1996 "520053.8" "284766.8" "a"
1997 "521315.1" "281393.7" "a"
1998 "518380.7" "285094"   "a"
1999 "525695.8" "289454.2" "a"
2000 "530312.8" "288981.1" "a"
```

このように数値も文字列に自動変換されてしまいます。これでは計算ができなくなりますから困ります。マトリックスには、このような大きな制限があります。数値系列と文字列系列を混在して扱いたいという場合には、データフレームという型が利用されます。データフレーム型の変数を作るためには、関数 data.frame( ) を使います。

```
> gdpcons<-data.frame(gdpcons,id)
> gdpcons
          gdp     cons     id
1980 312712.7 174382.7      b
1981 321490.5 177074.9      b
1982 331710.7 184799.3      b
1983 339823.8 189292.0      b
1984 353436.2 194237.4      b
1985 368184.1 201627.8      b
1986 379895.7 209050.0      b
1987 399442.3 217356.6 bubble
1988 424657.3 229129.5 bubble
1989 445468.8 238784.9 bubble
1990 469780.5 248840.1 bubble
1991 481660.7 256905.6 bubble
1992 483375.6 261560.2      a
1993 485498.4 266385.2      a
1994 490730.7 272342.2      a
1995 502794.3 277906.5      a
1996 520053.8 284766.8      a
1997 521315.1 281393.7      a
1998 518380.7 285094.0      a
1999 525695.8 289454.2      a
2000 530312.8 288981.1      a
```

このように data.frame(マトリックス、ベクトル）という操作をすることでデータフレームを作ることができました。data.frame( ) の引数には三つ以上の変数を指定しても構いません。既に作ったデータフレームにマトリックスやベクトルを追加することもできます。

データフレームにしておくと色々と便利な使い方ができます。表の中から gdp を取り出す方法を並べて見ましょう。

```
> gdpcons[,1]
 [1] 312712.7 321490.5 331710.7 339823.8 353436.2 368184.1 379895.7 399442.3
 [9] 424657.3 445468.8 469780.5 481660.7 483375.6 485498.4 490730.7 502794.3
[17] 520053.8 521315.1 518380.7 525695.8 530312.8
> gdpcons$gdp
 [1] 312712.7 321490.5 331710.7 339823.8 353436.2 368184.1 379895.7 399442.3
 [9] 424657.3 445468.8 469780.5 481660.7 483375.6 485498.4 490730.7 502794.3
[17] 520053.8 521315.1 518380.7 525695.8 530312.8
> attach(gdpcons)
> gdp
 [1] 312712.7 321490.5 331710.7 339823.8 353436.2 368184.1 379895.7 399442.3
 [9] 424657.3 445468.8 469780.5 481660.7 483375.6 485498.4 490730.7 502794.3
[17] 520053.8 521315.1 518380.7 525695.8 530312.8
```

一番上の方法はマトリックスでも使った方法です。データフレームは数値と文字列が混在したマトリックスですから、マトリックスで使える操作はデータフレームでも大体使えます。

記号「$」で必要な列の名称を指定する二番目の方法も便利です。

しかし、最も便利なのは関数 attach( ) で変数名探索先にデータフレームを追加しておいて、あとは列の名称を変数であるかのように使う最後の方法です。[19]attach( ) で加えた探索先をはずして元に戻すには関数 detach( ) を使って detach(gdpcons) とします。

それでは attach(gdpcons) のままの状態から id と打ち込んで、ちゃんと文字列になっているか、確認してみましょう。

```
> id
 [1] "b"      "b"      "b"      "b"      "b"      "b"      "b"      "bubble"
 [9] "bubble" "bubble" "bubble" "bubble" "a"      "a"      "a"      "a"
[17] "a"      "a"      "a"      "a"      "a"
```

データフレームという型の特徴がわかりましたね。

2.5 配列とリストについて

これまではベクトル、マトリックス、データフレームについて説明してきました。この他に **R** では配列（アレイ）とリストという型の変数が作れます。リストは何かの計算の結果としては、よく目にしますから、馴染みが出てきますが、自らリスト型の変数を作るという場面は当分の間はないと思います。配列は、マトリックスを三次元以上に拡張したものです。三次元の配列とは、縦横だけではなく奥行きの方向にも数値や文字列が並んでいる変数です。イメージとしては、同じ形のマトリックスが紙に印刷されて、それが何枚も机の上に重なっている、と考えるといいでしょう。**R** は 4 次元以上の配列も作れます。しかし、実際は三次元配列をごくたまに利用する程度です。

リストは、少しだけ、どういうものか見ておきましょう。

```
> list(gdp,cons,id)
[[1]]
 [1] 312712.7 321490.5 331710.7 339823.8 353436.2 368184.1 379895.7 399442.3
 [9] 424657.3 445468.8 469780.5 481660.7 483375.6 485498.4 490730.7 502794.3
[17] 520053.8 521315.1 518380.7 525695.8 530312.8

[[2]]
 [1] 174382.7 177074.9 184799.3 189292.0 194237.4 201627.8 209050.0 217356.6
 [9] 229129.5 238784.9 248840.1 256905.6 261560.2 266385.2 272342.2 277906.5
[17] 284766.8 281393.7 285094.0 289454.2 288981.1
```

[19]何か変数名を打てば、変数の内容が表示されますね。もし未作成の変数名を打ち込めば「そんなオブジェクトは見つからない」とメッセージが出ます。どこを探して変数名を見つけるか、あらかじめ決まっています。その探索先として、データフレームの列名称を加えるわけです。

```
[[3]]
 [1] "b"      "b"      "b"      "b"      "b"      "b"      "b"      "bubble"
 [9] "bubble" "bubble" "bubble" "bubble" "a"      "a"      "a"      "a"
[17] "a"      "a"      "a"      "a"      "a"
```

上の例では gdp、cons、id という三つのベクトルから list( ) 関数を使って、リストを作っています。どうですか？データフレームとは表示のされ方が違ってますね。リストは、ベクトル、データフレームなどをそのまま何もせずに一つの箱に入れるようなものです。箱の中に入った順番に、[[1]]、[[2]]、[[3]] と識別番号がふられていきます。ですから、リストの何番目の要素を取り出すか、list(gdp,cons,id)[[2]] のように番号を示せばよいわけです。この場合は、cons が表示されます。リストは一つの箱ですから、他のリストを入れても構いません。データフレームは、自由なようでも形としてはマトリックスのようにデータが並んでいる、どの列も同じ長さである、という制約があります。リストにはその制限もありません。単に、いくつかの変数を一つの箱に入れたもの、と考えておいてください。

3 グラフの描画

実質 GDP と民間最終消費支出のデータを統計的に分析してみましょう。統計分析を行う時は、データ全体から分かることをまず大づかみに知っておくことです。そのためにはグラフが大いに役立ちます。**R** では多彩なグラフ描画機能が備えられています。

3.1 簡単なグラフ

まず簡単なグラフをいくつか描いて見ましょう。次のコマンドを入力してみます。

```
> attach(gdpcons)
> plot(year,gdp)
```

横軸に年度、縦軸に実質 GDP が対応付けられて第 4 図のようなグラフが別のウィンドウに表示されます。

このようにグラフを描画する時に利用する基本的なコマンドは plot( ) です。引数は X 軸となるベクトル、Y 軸となるベクトルの順で指定します。この plot( ) には多くのオプションを引数の中で指定することで、グラフの外観を変えることができます。第 4 図では ∘ でデータが表されていますが[20]、これを通常の折れ線グラフにしてみましょう。それからタイトルも付けましょう。それには

```
> plot(year,gdp,type="l",main="GDP from 1980 to 2000")
```

[20] pch="."を引数として指定すれば、∘ でなくドットでデータがプロットされます。

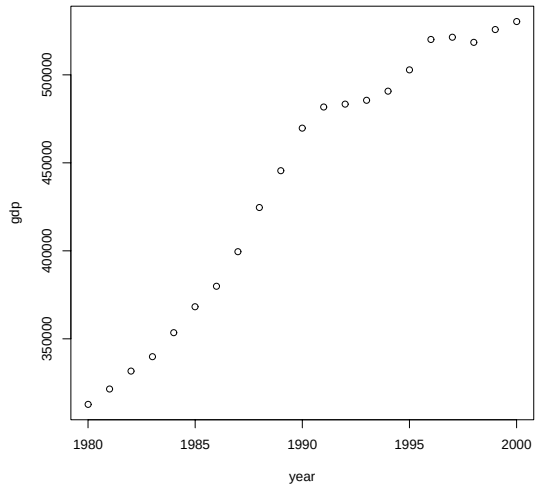

図 4: 実質 GDP の推移

で作成することができます。type はグラフの形式を指定するオプショナル・パラメーターです。何も指定しなければ散布図が描かれます。上の例のように type="l"を指定すると折れ線グラフになります。他に、どのような選択があるか、help(plot) を入力すれば詳細な説明が得られます。[21]

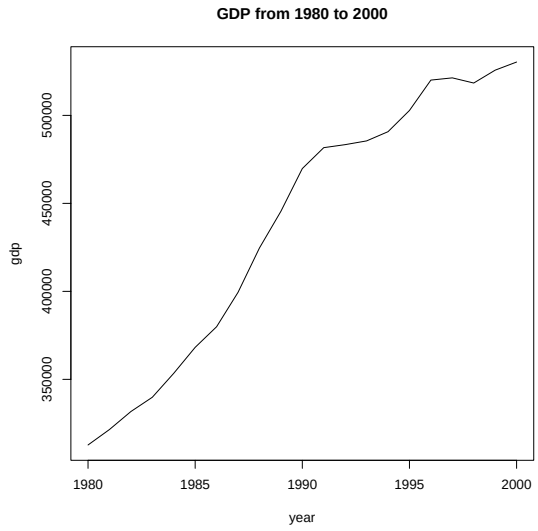

図 5: 実質 GDP の推移 (折れ線グラフ)

R の Version 1.3.0 では、日本語も少し扱えるのですが、残念ながらエクセルで作成するようには行かないようです。その代わり、自由自在に自分の作りたいグラフを作ることが可能です。このグラフはクリップボードにメタファイルとしてコピーすれば、Microsoft Word の

[21] help.start( ) を入力すれば、Internet Explorer が起動してグラフィックなヘルプ画面が表示されます。

文書中に貼り付けることもできます。また、ps、jpeg など頻繁に利用される画像ファイルで保存することもできます。

それでは、実質 GDP の折れ線に加えて、民間最終消費支出も加えて描画してみましょう。複数の折れ線を並置するには一工夫がいります。というのは、たとえば plot(year,gdp,cons) としてもエラーになってしまうからです。

第 6 図を描くには以下のように入力します。

```
> plot(year,gdp,main="GDP and Consumption",type="l",
+ ylim=c(100000,600000))
> lines(year,cons,lty=2)
```

このように、最初に plot( ) で GDP のグラフを描き、追加するグラフを lines( ) で描いていく方法をとります。2 行目の ylim=c(100000,600000) は、縦軸の目盛りを 100000 から 600000 の範囲とする意味です。GDP だけなら 300000 以上の目盛りを作っておけばいいし、実際、何も指定しなければ小さい値の目盛りは余計ですからカットされてしまいます。ところが、消費 cons の折れ線グラフも追加するのであれば、目盛りを自動作成しては困るのです。というのは、cons のほうは 170000 から 300000 程度までの値をとりますから、最初に GDP のグラフを自動作成しておいて、次に lines(year,cons,...) とやって線を追加しようとしても、目盛りの外になりますから、画面には何も表示されません。だから、あらかじめ必要な目盛りの範囲を具体的に指定する必要があるのです。lines( ) の中の引数 lty は線のパターンを指定するオプショナル・パラメーターで「２」を指定すると点線になります。[22]

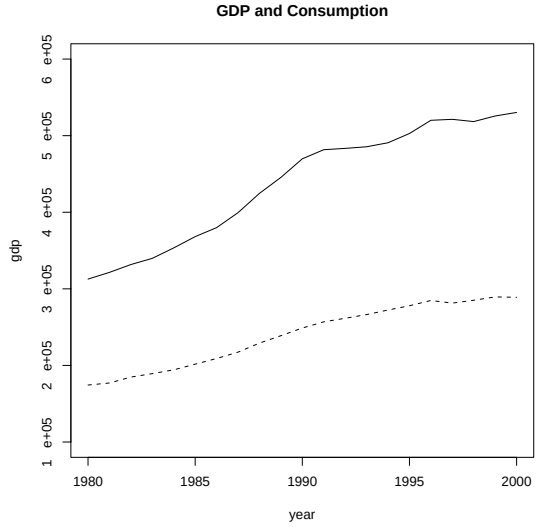

図 6: GDP と消費支出の推移

[22]実線が GDP、点線が CONS を示すという凡例をグラフ中に加えることも容易にできますが、これ以上の説明はマニュアルとヘルプ画面にまかせましょう。

3.2 組み合わせグラフの作成

これまでグラフを三枚作成しましたが、互いに関連のあるグラフは組み合わせて作成したくなることもよくあります。今度はその練習をしてみましょう。

組み合わせグラフを描くには、最初に横にいくつ並べるか、縦にいくつ並べるかを指定しないといけません。指定したあと、順番に plot( ) していくと、指定したパターンで複数のグラフをまとめることができます。

ここでは

- GDP（実線）と消費（点線）の前年度比増加率の折れ線グラフ
- マクロの粗貯蓄率 $\left(\frac{\text{gdp}-\text{cons}}{\text{gdp}}\right)$ の折れ線グラフ

の二つのグラフを横に並べることにしましょう。それには次のステートメントを入力します。[23]

```
> par(mfrow=c(1,2))
> growth.gdp<-(gdp[2:21]-gdp[1:20])/gdp[2:21]*100
> growth.cons<-(cons[2:21]-cons[1:20])/cons[2:21]*100
> plot(year[2:21],growth.gdp,main="GDP and Consumption Growth",
+ type="l")
> lines(year[2:21],growth.cons,lty=2)
> saving.rate<-(gdp-cons)/gdp*100
> plot(year,saving.rate,main="Macro Saving Rate",
+ type="l")
```

最初にグラフ・ウィンドウを 1 行 2 列に分割、つまり横に二つ並べて表示するという指定を行っています。これに限らず、グラフ描画の際の細かい指定は主に par( ) 関数を使って行います。

その次のコマンド二つは、GDP と消費の増加率を計算するものです。増加率は（今年の数値−前年の数値）÷今年の数値を 100 倍したものです。ベクトルの添字番号を 1 だけずらしているのは、この計算を行うためである点に注意してください。一つ目の plot( ) は最初のグラフ描画エリアに、二つ目の plot( ) は二番目のグラフ描画エリアに描かれます。lines( ) は、直前に plot されたグラフに折れ線を加えることも、よく覚えておいてください。

第 7 図は、縦長に過ぎるのが気になりますが、一枚のグラフ用紙に二つのグラフを描いたものと考えてください。[24]

[23] コマンドがいくつもあって長くなる場合は、あらかじめコマンドをプログラムファイルとして書いておいて、**R** からそのプログラムファイルを読んで実行させることも可能ですし、その方が便利です。これについては最後の

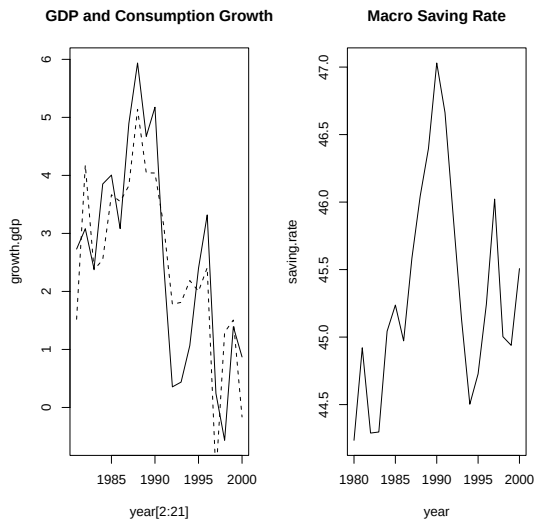

図 7: 組み合わせグラフ

4 統計分析のアウトライン

実質 GDP と消費支出の大体の動向はわかりましたから、今度はもう少し詳しく、統計的な分析を加えてみましょう。まず、GDP と消費をそれぞれ一変量として、その特徴を色々な記述統計量でみてみましょう。ついで、GDP と消費の相互関係、貯蓄率の趨勢的な特徴を順に見ていきましょう。

4.1 記述統計量の計算

データが与えられたとき直ちに計算しておかなければならない特性値は統計学の授業でも話したように**平均値**と**標準偏差**です。平均値は `mean( )` 関数で求めます。標準偏差は直接には出てきませんから、まず分散を `var( )` で求めておいて、その平方根を `sqrt( )` で計算します。もう一つ、注意しなければならないのは、分散といっても偏差二乗和をデータ数で割る標本分散と（データ数− 1）で割る不偏推定値の二つの区別です。関数 `var( )` は不偏推定値ですから、標本分散を求めたいときはデータ数を `n` として `(n-1)/n` を乗じておく必要があります。

```
> mean(growth.gdp);mean(growth.cons)
[1] 2.589886
[1] 2.482524
> n<-length(growth.gdp)
```

章で説明しましょう。

[24] 1 枚のグラフ用紙ですから、一つのファイルに書き出すことになるし、クリップボードにコピーしたり、ワープロの文書に貼り付ける場合も、一回の操作で行うわけです。

```
> sqrt(var(growth.gdp)*(n-1)/n)
[1] 1.78986
> sqrt(var(growth.cons)*(n-1)/n)
[1] 1.494700
```

コマンドを二つ、セミコロン（；）で区切って入力すると、それぞれの結果が表示されます。最初の結果を見ると、GDP成長率よりは消費の成長率のほうが、平均的に少し低いようです。次に、成長率の標本分散から標準偏差を求めています。消費支出のほうが標準偏差が小さく、成長率の変動の度合いが小さいことが分かります。標準偏差を計算する式は長く、GDPと消費と同じような式を入力するのは面倒です。矢印キー「↑」を押すと直前に入力したコマンドが順に遡って表示されます。これを**ヒストリー機能**と呼んでいますが、これを利用すると一部の修正だけで同じようなコマンドを反復して入力することができます。

1980年代と1990年代とで成長率のパターンはどのように違うでしょうか。よく「失われた10年」と言われるくらいですから、1990年代の成長率のほうが低いことが予想されます。それを確かめるにはどうしたらいいでしょうか。

```
> ysub<-seq(1981,2000)
> mean(growth.gdp[ysub<=1989])
[1] 3.849082
> mean(growth.gdp[ysub>1989])
[1] 1.559634
> mean(growth.cons[ysub<=1989])
[1] 3.426539
> mean(growth.cons[ysub>1989])
[1] 1.710148
> sqrt(var(growth.gdp[ysub<=1989]))
[1] 1.157558
> sqrt(var(growth.gdp[ysub>1989]))
[1] 1.655496
> sqrt(var(growth.cons[ysub<=1989]))
[1] 1.096810
> sqrt(var(growth.cons[ysub>1989]))
[1] 1.430941
```

分散は分母を調整しないで var( ) をそのまま使っています。

最初に変数ysubを定義しているのは、元データが1980年から2000年までであったため成長率は1981年から2000年までになっているためです。論理式を使ってベクトルの成分を抽出する時には、その論理式による論理値の系列が、成分を取り出すベクトルと同じ長さになっていることがポイントです。今は成長率の値は20個。ですから20個の値それぞれについて20個

の T、F の組み合わせを与える必要があるのです。長さが違ってもエラーにはなりませんが、どうなるか自分で確かめてみるといいでしょう。

上の結果を見ると、確かに 1990 年代以降の 10 年間は 80 年代に比べて平均成長率が大きく低下していることが分かります。ほぼ成長率が半減したと言えるほどです。大事なことは、成長率が低下したことに加えて、標準偏差が逆に高まっていることです。株価でも値動きの激しい銘柄はリスクが高いことは知られています。成長率の標準偏差が高いということは成長率で測定される好景気、不景気の振幅が激しくなっていることを示しています。これを経済学では Volatility と言っていますが、1990 年代に入って以降、マクロ経済の動向は不安定になっているわけです。

4.2 分布のヴィジュアル化

いまの成長率の例では、データ数もそれほど大きくはないので、期間を区分して計算することにより大体の様子はわかりますが、一般にはデータ全体の分布のパターンを把握しておくことが重要になります。それには hist( ) 関数を使ってヒストグラムを描くのが定石です。

たとえば、実質 GDP 成長率については

```
> hist(growth.gdp)
```

によって下の第 8 図が得られます。特に平均成長率を中心に正規型で分布しているわけではなく、むしろ一様分布にも近いといえるパターンを示しています。

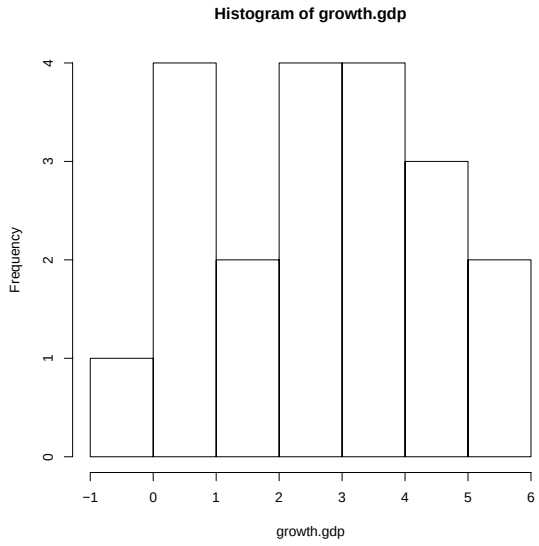

図 8: GDP 成長率の分布

4.3 回帰分析

統計分析といえば回帰分析というくらいに経済学では回帰分析を多用します。ここでは

- GDP 成長率と消費成長率にどの程度の関係があるか
- 貯蓄率に長期的な趨勢はあったか

の二点を簡単な方法でとりあげてみましょう。

4.3.1 GDP と消費

まず GDP と消費には双方とも増加トレンドがあることは元データをみれば直ちにわかります。問題は、この二つに因果関係、言い換えれば一方が他方を予測するための有用な情報となっているかということです。

GDP 成長率と消費成長率の散布図を描いて見ましょう。

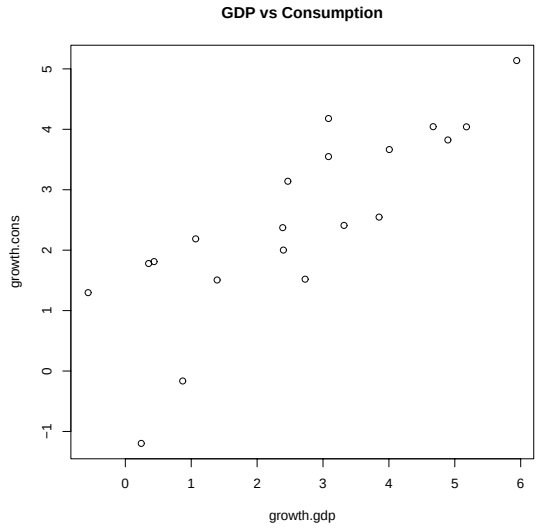

図 9: GDP と消費の成長率

これをみると GDP が大きく増えるほど消費の成長率も高いことがわかります。しかし、有効需要の原理によれば消費が増えれば生産を押し上げて GDP も増えるわけですから、同じ年の両者の成長率を散布図に描いた時、正の相関が認められるのは当然ともいえます。

理論的に考えてみましょう。このところ頻繁に応用されている消費の動学モデルから導出されるオイラー方程式から、ある条件の下で代表的家計は、何ら新たな撹乱がなければ、一定の消費支出を維持する。言い換えると、前年に比べて消費の変動が生じるのは、前年までに予想

されていなかった新たな撹乱がその年に発生したためである。とすれば、各年の消費の増減は前年までの情報からは独立のはずである。こういう理論的結論が最近新しく発展してきた新古典派モデルから得られています。

素朴な特定化ですがこれを確かめてみましょう。方法としては、回帰モデル

$$\frac{\Delta C}{C} = \alpha + \beta \left(\frac{\Delta Y}{Y}\right)_{-1}$$

を推定して、仮説検定

$$\mathrm{H}_0 : \ \beta = 0 \quad vs \quad \mathrm{H}_1 : \ \beta \neq 0$$

を行うことにします。

```
> x<-growth.gdp[1:19]
> y<-growth.cons[2:20]
> reg1<-lm(y~x)
> summary(reg1)

Call:
lm(formula = y ~ x)

Residuals:
    Min      1Q  Median      3Q     Max
-4.0393 -0.2069  0.2463  0.5585  1.6227

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   1.2425     0.5446   2.282   0.0357 *
x             0.4815     0.1689   2.850   0.0111 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1.319 on 17 degrees of freedom
Multiple R-Squared: 0.3234,     Adjusted R-squared: 0.2836
F-statistic: 8.125 on 1 and 17 DF,  p-value: 0.01106
```

まず、GDP成長率で1年のラグをとるので最初のデータは利用できません。二年目以降を利用することになります。だから標本期間は19年分になり、GDPは`gdp[1:19]`、[25]消費は`cons[2:20]`が最小二乗推定の対象になります。

[25]二年目からということは一年ラグをとると一年目のデータからということになります。

最小二乗推定には関数 lm( ) を利用します。結果は、他の検定で残差を利用することもありますから、変数に一括保存しておきましょう。結果の概要を見るには summary( ) を使います。lm( ) の引数の y ~ x はモデル定義式と呼ばれており、「被説明変数 ~ 説明変数」のパターンで記述します。複数の説明変数がある場合は、y~$x_1 + x_2 + x_3 + \cdots$ などとプラス記号を用います。定数項はデフォールトで含まれることになっています。

結果を見ればわかるように、GDP の一年ラグにつく係数は推定値で 0.4815、T 値が 2.850 ですから、5%で有意になります。つまりゼロとは見なせないわけで消費の動学的最適化仮説は棄却されることになります。[26]

4.3.2 マクロの粗貯蓄率

それではマクロの粗貯蓄率について長期的な趨勢があったかどうかを簡単にチェックしてみましょう。趨勢がないということは一定の平均値の周りで貯蓄率が変動していたことになるし、趨勢があるということは一定の増加傾向か、減少傾向に服していたということになります。だから、

$$\frac{S}{Y} = \alpha + \beta\, T$$

を推定して係数 β のゼロ検定を行えばよいことになります。ここで変数 T はトレンド変数、つまり T=1,2,3,.. です。

結果を示しましょう。

```
> length(saving.rate)
[1] 21
> trend<-seq(1,21)
> reg1<-lm(saving.rate~trend)
> summary(reg1)

Call:
lm(formula = saving.rate ~ trend)

Residuals:
    Min      1Q  Median      3Q     Max
```

[26]経済分析としては色々な問題を含んでいます。まず、最適化条件は家計について成立するものですから、マクロの GDP と民間最終消費を使うべきではありません。更に、集計レベルではなく世帯当り、ないしは一人当たりの数値を使うべきでしょう。また、不均一分散、系列相関などの存在に全く注意を払っていません。これらによっては係数の有意性検定は大きく左右されてしまいます。**R** の使い方に関する素朴な事例として解釈しておいてください。

```
-0.9506 -0.6835 -0.1488  0.4676  1.7126

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 44.94598    0.35065 128.178   <2e-16 ***
trend        0.03382    0.02793   1.211    0.241
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.7749 on 19 degrees of freedom
Multiple R-Squared: 0.07166,    Adjusted R-squared: 0.0228
F-statistic: 1.467 on 1 and 19 DF,  p-value: 0.2407
```

これをみると、変数 trend の係数推定値は 0.034、T 値は 1.211 になっています。いうまでもなく有意ではありませんから帰無仮説を棄却できません。つまり、1980 年から 2000 年にかけて日本のマクロの粗貯蓄率は一定であったと考えてもよい、という結論になります。[27]

5 ファイルの入出力

データ gdpcons はデータファイルから入力したことは既に述べました。その時、データは別のファイルから入力するのが通常の方法であることも説明しました。最後にファイルの入出力について簡単に解説しておきましょう。といっても、日常的に利用するのは二つの方法だけだといっても十分でしょう。

1. 関数 read.table( ) を使う。
   通常のデータファイルは、エクセルで作る時にもそうですが、1 行目に変数の名称（その列の数値の名前）、2 行目から数値が続くわけです。[28] このような形のファイルを読み取るときには

   ```
   gdpcons<-read.table("gdpcons.txt",header=T)
   ```

   とすればデータフレーム型として入力することができます。列名称は 1 行目の名称がつけられます。[29]
2. 関数 scan( ) を使う。
   簡単な入力はファイルを使わずキーボードから入れてしまったほうが簡単です。こんな場合は関数 scan( ) を利用します。たとえば、

[27] これも経済分析としては問題を残しています。その年々の貯蓄率がどのように決定されるのかというモデルがなければ、経済的な仮説を検定したということにはなりません。

[28] gdpcons を入力した時のデータファイルの内容を別添しておきます。

[29] read.table( ) は実は色々なデフォールトの約束事があって、使いやすい関数ともいえません。ここで述べた形のデータファイルが通常の作り方ですから、上のパターンを守って利用したほうがいいでしょう。

```
> x<-scan()
1: 1
2: 2
3: 5
4: 6
5: 8
6: 9
7:
Read 6 items
> x
[1] 1 2 5 6 8 9
```

このように保存する変数名を決めて関数 scan() を入力すると、上に表示されているようなプロンプトが出てきます。1 個データを入力するごとに番号は更新されます。最後に何も入力せずリターンキーだけを押すと、それまでに入力された数値がベクトルとして変数 x に保存されます。[30]

では次にファイルへの出力を簡単に述べておきましょう。ファイルへの出力はそれほど必要になることはありません。結果は **R** からワープロの文書にコピー・ペーストができるし、グラフもファイルに保存して、他のアプリケーションで利用が可能だからです。それでも計算して得られたマトリックスやデータフレームの結果をそのままファイルに書き出して、エクセルなどで読み取りたいこともあるでしょう。

マトリックスの書き出しは関数 write( ) を使います。しかし、write( ) の動き方は少しユニークです。特に行列を書き出すときに、縦方向に見ながら成分を書き出してしまいますから、出来たファイルでは転置行列になってしまうのが欠点です。[31]

だから、マトリックスの書き出しは次のようにします。

```
write(t(x),"filename",ncol=ncol(x))
```

t(x) はマトリックス x の転置行列です。つまり行列を横方向に読みながら、元の行列と列数を一致させて、ファイルに書き出していきなさいという意味です。

ファイルを利用するもう一つの目的は 24 ページの脚注でも言及した長いコマンドをプログラムファイルに書いておいて自動的に実行させる方式です。

プログラムファイルが特に有用なのは、この導入マニュアルでは説明をしませんが、自前の関数を作って関数定義を一挙に読み込ませておく場合でしょう。たとえば、月次系列を四半期系列に転換したり、年次系列にまとめたりするのは、**R** に元々備わっている関数ではできませ

[30] scan( ) は非常に柔軟な入力が可能な多数のオプションをもったコマンドです。詳しくは help(scan) して調べてください。

[31] **R** ではマトリックスは列方向優先の順番でメモリー内に記憶されていると覚えておいてください。

ん。[32]しかし、自前の関数を作って四半期への変換、年次への変換を行うことは別に何でもないことです。このように自前の関数を自在に作り、それをプログラムファイルに書いておくと、自分好みの **R** に改造していくことができます。

たとえば、

```
m2y <- function(x){
  if((length(x) %% 12) != 0) return("Inconsistent Number of Data");
  n <- length(x) / 12;
  y <- 1:n;
  for(i in 1:n){
  b <- 12*(i-1)+1;
  e <- 12*(i-1)+12;
  y[i] <- mean(x[b:e]);
  }
  y
}
```

という関数定義をファイル `m2y.r` に保存しておき、**R** から

```
> source("m2y.r")
```

と入力すれば、これ以後、月次系列 `x.month` を

```
x.year<-m2y(x.month)
```

このように関数 `m2y( )` を使うことで年次系列にすることができるようになります。[33]

最後に、**R** の作業領域の保存について補足しておきます。最初に述べたように **R** を終了する時には、`Save workspace image?`というメッセージが出ます。これは変数として保存された値や、関数として定義された内容を、再開時にまた使えるように保存するかどうかを聞いているわけです。ここで `OK` をクリックすると `.Rdata` という名のファイルに保存されることになります。

さて、この `.Rdata` ですが、**R** を起動すると自動的に読み込まれますから、作成した変数などが直ちに利用できる状況になります。しかし、色々な計算を平行してすすめている場合は、データファイル、プログラムファイルは異なったディレクトリーに保存し、作業の対象を切り替えるごとに、カレントディレクトリーを切り替えるのが普通のパターンです。ディレクトリー

[32] タイムシリーズという型にしておくと、この種の操作ができるようにはなりますが、割愛します。

[33] [File] メニューから [Source R code] を選択し、ダイアログウィンドウで必要なプログラムファイルをマークして OK をクリックすることでも同じことができます。

切り替えをした時には、そのディレクトリーに入っている.Rdataは自動的に読み込まれず、変数の内容はそれまでのままですから、ここで改めて.Rdataを読んでおかないといけません。

これを行うのが[File]メニューの[Load Workspace]で、保存するのが[Save Workspace]です。[Save Workspace]を選ぶとカレントディレクトリーに.Rdataが保存されます。起動直後には、このファイルが保存されているディレクトリーにはいないはずですから、切り替え後にこの.Rdataを読み込むステップが必要です。[34]

[34].Rdataとは異なった名前を付けて別のファイルに作業結果を保存しても構わないことは勿論です。

別表：データファイルの内容

YEAR	GDP	CONS
1980	312712.7	174382.7
1981	321490.5	177074.9
1982	331710.7	184799.3
1983	339823.8	189292.0
1984	353436.2	194237.4
1985	368184.1	201627.8
1986	379895.7	209050.0
1987	399442.3	217356.6
1988	424657.3	229129.5
1989	445468.8	238784.9
1990	469780.5	248840.1
1991	481660.7	256905.6
1992	483375.6	261560.2
1993	485498.4	266385.2
1994	490730.7	272342.2
1995	502794.3	277906.5
1996	520053.8	284766.8
1997	521315.1	281393.7
1998	518380.7	285094.0
1999	525695.8	289454.2
2000	530312.8	288981.1