OX によるプログラミング入門

大森裕浩

2002 年 2 月

目次

第 1 章

OX 入門

1.1 OX 言語とは？

経済分析のためにエクセルや TSP,STATA というソフトウェアをよく使いますが、いずれもすでによく使われている分析方法しか用意されていません。最新の研究に基づいた分析方法がこうしたソフトウェアに搭載されるには長く時間がかかるため、搭載された頃にはすでに古い道具になってしまうのです。

これは例えば洋服を買うときに、お店に並んでいる洋服の中から気に入った服を選んで買うことにあたります。もし最新の流行を取り入れた洋服をいちはやく着たいならば、自分で服を作ってしまわなくてはなりません。この自家製プログラムを作る道具が、プログラミング言語 OX （オックス）です。

OX （オックス）はオックスフォード大学の Jurgen Doornik によって開発されたプログラミング言語です。プログラムで用いられる変数は行列を基本としていて、行列の演算を通してさまざまな計算を行うので行列言語と呼ばれます。基本的なプログラムの構造は C や C++ という、より緻密な基礎言語と同様な構造をもっているのでこの言語を習得することはまた C や C++ 言語の入門となります。

OX と性質のよく似た行列言語に IML (SAS), S-plus, Gauss, Matlab などがあります。 IML は SAS 言語の補完的な行列言語であり、単体の言語としてはあまり使われていません。 S-plus はグラフィック機能にすぐれたデータ解析プログラムであり、主としてプルダウン型のメニューから分析方法を選んでデータの分析をすすめていくソフトウェアで、行列言語としての機能も持っていますが計算速度が遅いので、計算負荷の高いプログラムには向いていません。 Gauss と Matlab は OX とともにプログラミング専用の言語で、どれもよく似ていますが、それらのなかでも OX は特に計算の実行速度が速いことで知られており、最近発展の著しいシミュレーションに基づくさまざまな計算に適しています。また、 OX のあるバージョン（コンソールバージョンなど）は教育目的には無料 (!) で使うことができるで自習にも最適なプログラムです。インターアクティブな画面でグラフを見ることはできませんが、ファイルに出力すれば他のプログラム (Ghostview など）を使うことで見ることができます。

もし使用目的が研究目的ではなく（従って入手費用が高くなります）、計算負荷の高くないプログラムを使いたいということであれば付属プログラムが多いという点で Gauss, Matlab や S-plus もよいでしょう。

1.2 OX をインストールする

1.2.1 OX 本体のインストール

Doornik のホームページ **http://www.nuff.ox.ac.uk/Users/Doornik/index.html** からダウンロードします。そして oxcons300.exe をダブルクリックするとインストールされます。

1.2.2 OxEdit のインストール

次に OX でプログラムを書くときに便利なソフトウェア（プログラムの編集ソフトウェアです） OxEdit をインストールします。 OxEdit のホームページ **http://www.oxedit.com/** から oxedit161.exe というファイルをダウンロードします。そして oxedit161.exe をダブルクリックすればインストールされます。その後、最初に Oxedit を起動するときに Preferences → Add Predefined Module で OX をチェックします。そして OX フォルダの BIN フォルダを指定してやれば次回の起動から問題なくできます。さらに Preferences → Tool Bars として Module 1 というボタン（つまり OX) をツールバーに登録しておくと実行が便利です。はじめは File メニューから New を選び、ファイルにコマンドを書き込んでいきます。試しにサンプルプログラムとして oxtut2c.ox(Tutorial フォルダにあります）を試してみましょう。Module1 ボタンを押せば実行結果が画面に出てくるはずです。

※インストール後、 autoexec.bat ファイルに OX 関係の PATH を追加しましょう。 PATH には OX の BIN フォルダのパスを追加します。もし OX が C:¥Ox にあるとすれば

```
SET  OX3PATH=C:¥Ox¥include;c:¥Ox
SET  PATH=%PATH%;C:¥Ox¥bin
```

のように追加します。

第 2 章

OX の基礎

2.1 プログラムの基本

ここでは、現在無料で配布されている oxedit というソフトウェアを使います。まず、 oxedit.exe をダブルクリックして起動します。そして **File** メニューから **New** を選びます。そこで次のような内容を入力しましょう。プログラムの基本的なつくりは C 言語と同じです。各種ヘルプファイルは doc フォルダの index.html ファイルを開けるとみられます。最初は Function summary を見るとよいでしょう。

基本的なプログラム

```
#include <oxstd.h>
main(){
 decl a,b,c;
 a=2;b=4;c=a+b;
 print("a=",a," b=",b," a+b=",c);
}
```

はじめの **#include <oxstd.h>** というのは、 **oxstd.h** というヘッダーファイル（拡張子の.h は header の h です）を読み込む、ということを表します。プログラムには、すでに作られた便利な関数 (ライブラリ関数という）がありますが、これを使うためにはまず最初にその関数をリストアップしておく必要があります。ヘッダーファイルの中には使う関数名のリストがあり、これを読み込むことによって毎回関数をリストアップする手間を省くことができます。この **oxstd.h** というファイルは OX の標準的 (standard) なライブラリ関数のヘッダファイルという意味です。中をみてみると log や max といった関数名が並んでいます。特に理由がなければ、この標準的なライブラリ関数は必ず読み込んでおきます。

次に **main(){...}** という部分があります。この部分はプログラムの主要 (main) な部分で、このなかに計算する内容を書いていきます。最初の **decl a,b,c;** ですが、これは「以下では a,b,c を変数として使います」ということを宣言する (declare の decl) ということです。最後のセミコロン**;** はこの宣言文の終わりを示すために用いられます。次の **a=2;** と **b=4;** は変数 a に 2 を、変数 b に 4 を割り当てるということです。そして **c=a+b;** は a+b の結果を c として保存するということです。

最後の print 文では結果を出力させますが” ”でくくられたなかの文字はそのまま文字を、それ以外は変数の数値を出力します。ここで **File** メニューから **Save** を選び、保存しましょう。名前は sample1.ox というように〜.ox というかたちで拡張子を ox としましょう。

さて **Module** メニューから OX を選択すると (あるいはすでに登録しておいた Module 1 ボタンを押すと), Output 画面がでて結果が出力されます。以下がその結果です。

基本的なプログラム (結果)

```
a=2 b=4 a+b=6
```

2.2 行列の計算

さて次に行列での計算を見てみましょう。ここでは

$$a = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad b = \begin{pmatrix} 10 & 20 \\ 30 & 40 \end{pmatrix}, \quad a+b = \begin{pmatrix} 11 & 22 \\ 33 & 44 \end{pmatrix},$$
$$a^{-1} = \begin{pmatrix} \text{-2} & 1 \\ 1.5 & \text{-0.5} \end{pmatrix}, \quad ab = \begin{pmatrix} 70 & 100 \\ 150 & 220 \end{pmatrix},$$

という計算を OX を使って行います。

行列の基本的な演算

```
//  matrix basic calculation
#include <oxstd.h>
main(){
 decl a,b,c;
 a=<1,2;3,4>;b=<10,20;30,40>; c=invert(a);
 print("a=",a,"b=",b,"a+b=",a+b, "a^{-1}=",c,"a*b=", a*b);
}
```

// で始まる部分はプログラムの実行では無視されるのでプログラムの説明、コメントなどを書き込みます。 $\mathbf{a} =< 1, 2; 3, 4 >$ **;** は行列の読み込みで、 $<$ $>$ のなかでは**,** は列の区切りを**;** は行の区切りを意味します。 **invert(a)** は行列 a の逆行列を計算します。以下がアウトプットです。

行列の基本的な演算 (結果)

```
a=
       1.0000       2.0000
       3.0000       4.0000
b=
       10.000       20.000
       30.000       40.000
a+b=
       11.000       22.000
       33.000       44.000
a^{-1}=
      -2.0000       1.0000
       1.5000     -0.50000
a*b=
       70.000       100.00
       150.00       220.00
```

以上が基本的なプログラムの書き方です。以下では、実際のプログラミングで必要となるテクニックに関して説明を加えていきましょう。

2.3 行列の便利なつくりかた

※ 1/a も invert(a) と同じ結果を出します。また 0 だけからなる $m \times n$ 行列 a は a=zeros(m,n); で、1 だけからなる $m \times n$ 行列 a は a=ones(m,n)；で、 m 次元単位行列 a は a=unit(m)；で作成できます。

便利な行列の作り方

```
#include <oxstd.h>
main(){
 decl a,b,c;
 a=zeros(2,1);b=ones(1,3);c=unit(2);
 print("a=",a,"b=",b,"c=",c);
 }
```

出力は以下の通り。

便利な行列の作り方 (結果)

```
a=
      0.00000
      0.00000
b=
       1.0000       1.0000       1.0000
c=
       1.0000      0.00000
      0.00000       1.0000
```

行列の便利な作成方法 2

```
#include <oxstd.h>
main()
{
 decl a,b,c;
 a=<1;2>;b=<3;4>;
 println("a=",a,"b=",b);
 println("a~b=",a~b);
 println("a|b=",a|b);
}
```

行列の便利な作成方法 2(出力)

```
a=
       1.0000
       2.0000
b=
       3.0000
       4.0000
a~b=
       1.0000       3.0000
       2.0000       4.0000
a|b=
       1.0000
       2.0000
       3.0000
       4.0000
```

2.4 いろいろな行列演算

すでにいくつか例をみたように加減乗除は a+b,a-b,a*b,a/b で表記されます。

+	足し算
−	引き算
*	かけ算
/	わり算
^	べき乗
|	縦に行列をつなげる
~	横に行列をつなげる

次の例で見るように、通常の行列の乗除ではなく行列の要素ごとの乗除の場合は、. (dot) を記号の前に用いて.* （要素ごとのかけ算）, ./(要素ごとのわり算) とします。

2.4.1 行列の一部をとりだす

行列の 1 部をとりだす

```
#include <oxstd.h>
main()
{
 decl a,b,c;
 a=<1,2;3,4>;
 print("a",a,"a'=",a');
 print("a[0][0]=",a[0][0],", a[0][1]=",a[0][1],"\n");
 print("a[1][0]=",a[1][0],", a[1][1]=",a[1][1],"\n");
 print("a[0][0:1]=",a[0][0:1],"a[1][0:1]=",a[1][0:1]);
 print("a[0:1][0]=",a[0:1][0],"a[0:1][1]=",a[0:1][1]);
 }
```

行列の 1 部をとりだす (出力)

```
a
       1.0000       2.0000
       3.0000       4.0000
a'=
       1.0000       3.0000
       2.0000       4.0000
a[0][0]=1, a[0][1]=2
a[1][0]=3, a[1][1]=4
a[0][0:1]=
       1.0000       2.0000
a[1][0:1]=
       3.0000       4.0000
a[0:1][0]=
       1.0000
       3.0000
a[0:1][1]=
       2.0000
       4.0000
```

2.4.2 行列のいろいろな計算

行列のいろいろな計算

```
#include <oxstd.h>
main()
{
 decl a,b,c;
 a=b=<1,2;3,4>;c=<10;20>;
 print("a",a,"b=",b,"c=",c);
 print("a^2=",a^2,a*a); //a^2=a*a;
 print("a.^2=",a.^2); // componentwise square
 b+=1; // b=b+1
 print("b=",b);
 print("a/b=", a/b, a*invert(b)); // a*invert(b)
 print("a./b=",a./b); //componentwise
 print("a*b=", a*b); // a*invert(b)
 print("a.*b=",a.*b); //componentwise
 print("a**(c')=",a**(c'));  //Kronecker product
 }
```

行列のいろいろな計算 (出力)

```
a
       1.0000       2.0000
       3.0000       4.0000
b=
       1.0000       2.0000
       3.0000       4.0000
c=
       10.000
       20.000
a^2=
       7.0000       10.000
       15.000       22.000

       7.0000       10.000
       15.000       22.000
a.^2=
       1.0000       4.0000
       9.0000       16.000
b=
       2.0000       3.0000
       4.0000       5.0000
a/b=
       1.5000     -0.50000
      0.50000      0.50000

       1.5000     -0.50000
      0.50000      0.50000
a./b=
      0.50000      0.66667
      0.75000      0.80000
a*b=
       10.000       13.000
       22.000       29.000
a.*b=
       2.0000       6.0000
       12.000       20.000
a**(c')=
       10.000       20.000       20.000       40.000
       30.000       60.000       40.000       80.000
```

2.4.3 便利な加減乗除の記号

しばしば現在の値にある値を加えたり、引いたりして新しい値に置き換えたいということがあります。もちろん

よくある置き換えの加減乗除

```
#include <oxstd.h>
main(){
 decl x1,x2,x3,x4;
 x1=x2=0;x3=x4=1;
 x1=x1+1;x2=x2-1;x3=x3*2;x4=x4/2;
 print("x1=",x1,", x2=",x2,", x3=",x3,", x4=",x4);
}
```

といったように書くこともできますが、変数の名前が長かったりすると面倒です。そのような場合、以下のような表記を行うとプログラムがすっきりして見やすくなります。

便利な加減乗除の記号

```
#include <oxstd.h>
main(){
 decl x1,x2,x3,x4;
 x1=x2=0;x3=x4=1;
 x1+=1;x2-=1;x3*=2;x4/=2;
 print("x1=",x1,", x2=",x2,", x3=",x3,", x4=",x4);
}
```

便利な加減乗除の記号 (出力)

```
x1=1, x2=-1, x3=2, x4=0.5
```

2.5 同じ計算を繰り返す方法 (for, while 文)

同じ計算を繰り返して行う場合は以下のように **for** や **while** を使います。以下では初期値 i=0 から始めて i の値を 1 づつ増やし $i < 3$ である間は続行するということを表す。

for による繰り返し計算の方法

```
#include <oxstd.h>
main()
{
 decl i;
 println("loop: increasing");
 for(i=0; i < 3;++i)
 {
  println("i=",i);
 }
 println("loop: decreasing");
 for(i=3; i > 0; --i)
  {
   println("i=",i);
  }
 }
```

for による繰り返し計算の方法 (結果)

```
loop: increasing
i=0
i=1
i=2
loop: decreasing
i=3
i=2
i=1
```

同じ結果は次の while 文でも得られる。

while による繰り返し計算の方法

```
#include <oxstd.h>
main(){
 decl i;
  i=0;
 println("loop: increasing");
 while(i < 3)
 {
  println("i=",i);
  ++i;
 }
 i=3;
 println("loop: decreasing");
 while(i > 0)
 {
  println("i=",i);
 --i;
 }
```

do-while による繰り返し計算の方法

```
#include <oxstd.h>
main(){
 decl i;
 i=0;
 print("loop: increasing");
 do
 {
  print("i=",i);
  ++i;
 } while(i < 3);

 i=3;
 print("loop: decreasing");
 do
 {
  print("i=",i);
 --i;
 } while(i > 0);
```

do − while 文は while 文とほとんど同じですが、 while のかっこの中の評価がループの前に行われる (while 文) か、後に行われるか (do − while) の違いです。

2.6 条件付きの計算 (if 文)

if 文では条件式を評価して満たされれば次の文を実行し、満たされなければ何もせずに if 文から抜けるという形になっています。具体的には

If 文の基本形

```
if(条件式 A1){プログラム B1;}
```

という形になり、条件式 A1 が満たされれば B1 を実行し、満たされなければ何もせずに次の文へ進むということになります。これはさらに拡張できて

If 文の一般的な形

```
if(条件式 A1){プログラム B1;}
else if(条件式 A2){プログラム B2;}
else(条件式 A3){プログラム B3;}
```

という形になり、条件式 A1 が満たされれば B1 を実行し、満たされなければ else if の条件式 A2 に進み、今度は条件式 A2 が満たされれば B2 を実行し、満たされなければ else を実行するということになります。 else if はいくらでも増やすことができます。また最後の else はあってもなくてもかまいません。以下の例では 1 から 9 までの整数を x として、 3 で割って 1 余るものは y1, 2 余るものは y2, 割り切れるものは y3 へ分割するというプログラムです。 fmod(A,B) は A/B の剰余を計算する関数です。

If 文

```
#include <oxstd.h>
main()
{
    decl i,j1,j2,j3,x,y1,y2,y3;
x=<1;2;3;4;5;6;7;8;9>;
y1=y2=y3=zeros(3,1);
j1=j2=j3=0;
for(i=0;i<9;++i){
 if(fmod(x[i],3)==0){
   y1[j1]=x[i];j1+=1;}
 else if(fmod(x[i],3)==1){
   y2[j2]=x[i];j2+=1;}
 else{
   y3[j3]=x[i];j3+=1;}
}
print("x=",x,"y1=",y1,"y2=",y2,"y3=",y3);
}
```

If 文 (出力)

```
x=
       1.0000
       2.0000
       3.0000
       4.0000
       5.0000
       6.0000
       7.0000
       8.0000
       9.0000
y1=
       3.0000
       6.0000
       9.0000
y2=
       1.0000
       4.0000
       7.0000
y3=
       2.0000
       5.0000
       8.0000
```

If 文の条件式の中でよく使われる記号は以下の通りです。 A と B が実数の場合

記号	意味	
`A == B`	A は B と等しい	$(A = B)$
`A != B`	A は B と等しくない	$(A \neq B)$
`A < B`	A は B より小さい	$(A < B)$
`A <= B`	A は B 以下	$(A \leq B)$
`A > B`	A は B より大きい	$(A > B)$
`A >= B`	A は B 以上	$(A \geq B)$

もし、 A と B が行列の場合に、要素ごとの比較をしたければ.(dot) 記号を用いて

記号	意味	
`A .== B`	A の要素は B の要素と等しい	$(A = B)$
`A .!= B`	A の要素は B の要素と等しくない	$(A \neq B)$
`A .< B`	A の要素は B の要素より小さい	$(A < B)$
`A .<= B`	A の要素は B の要素以下	$(A \leq B)$
`A .> B`	A の要素は B の要素より大きい	$(A > B)$
`A .>= B`	A の要素は B の要素以上	$(A \geq B)$

とします。また論理記号としては

記号	意味
`A && B`	A かつ B
`A || B`	A または B
`A .&& B`	A の要素かつ B の要素
`A .|| B`	A の要素または B の要素

第 3 章

関数のつくりかた

3.1 関数の基本的なつくりかた

OX では、関数を main 部分の上に書きます。まず Hello! という文字を出力する関数を書いてみましょう。関数の名前は fHello とします。名前はどんなものでもいいのですが、大文字から始める名前の前に f を付ける習慣をつけるとプログラムが見やすくなります。関数の名前の後には () が必ずあり、引数があるときに使われます。 main 文のなかで fHello() という呼び出しが行われると main 文の前のほうに fHello という名前の関数を探しに行き、その関数を実行します。実行が終わると再び main 文の、関数呼び出しが行われた直後の部分に戻ってプログラムの実行を続けます。

最も簡単な関数の例

```
#include <oxstd.h>
fHello(){
println("Hello!");
}
//
main(){
fHello();
}
```

出力結果は

最も簡単な関数の例 (出力)

```
Hello!
```

となります。

3.2 引数のある関数

次に引数を用いた関数例を示します。 fTest1 と fTest2 という 2 つの関数では main 文のなかで定義された変数 x を用いて関数を実行します。 fTest1 の関数の中では引数が変数 x であり、変数の値は定数 (const) となっている。従って関数 fTest1 のなかではこの値は変えることはできず, 関数を実行した後も main 文のなかの x の値が変わらない。このことは (1) ∼ (3) で確認できます。出力を見ると x=0 のまま同じ値をとっています。
fTest2 では、関数の引数は変数 x ではなく変数 x のアドレスが指定されています。アドレスとは変数 x の値が格納されている場所を表します。通常、変数 x の値がいろいろな値をとっても格納場所、つまり x のアドレスは同じ場所になります。ここでは x のアドレスを main 文の関数で &x として表現し、引数で使います。また関数の中ではこのアドレスを adX という変数で受けます。この adX という変数を使って adX という場所に格納されている値 (つまり x の値) を表すときに adX[0] とします。従って

```
変数の値                  変数のアドレス
    x       -->           &x
  adX[0]    <--           adX
```

となります。 fTest2 では x のアドレスが引数として関数に渡されて、関数の中でそのアドレスに格納されている値を変更するので、アドレスは同じままですが、アドレスの指す値は変更されてから main 文へ戻ってきます。従って (4) では x=0 だったのが、関数のなかの (5) で x=1 となり、 main 文に戻ったときにも (6) で x=1 に変更されたままになるわけです。

引数のある関数

```
#include <oxstd.h>
fTest1(const x){
decl y;
y=x;println("(2) x=",y);
}
fTest2(const adX){
println("(4) x=",adX[0]);adX[0]=1;println("(5) x=",adX[0]);
}
main(){
decl x;
x=0;println("(1) x=",x);
println("Function without using address");
fTest1(x);println("(3) x=",x);
//
println("Function using address");
fTest2(&x);println("(6) x=",x);
}
```

引数のある関数 (出力結果)

```
(1) x=0
Function without using address
(2) x=0
(3) x=0
Function using address
(4) x=0
(5) x=1
(6) x=1
```

この例で fTest1 関数の中にも変数名 x が (const x のところ）でてきますが、これは main 部分の中の x とは異なるものです。従って fTest1 関数の中の x をすべて z と置き換えても同じ結果になります。このように変数は main 文の変数は main 文のなかだけで有効であり、関数の中で現れる変数はその関数の中だけで有効となります。このような変数は局所的にしか有効ではないのでローカル変数といいます。基本的には変数はローカルであり、これによってプログラムの誤りを見つけやすくなります。ただ場合によってはローカルの枠を超えて、すべての部分で共通な変数を必要とすることがあります。このような変数をグローバル変数といいます。グローバル変数の使い方は後の例で説明することにします。

3.3 返値のある関数

上のように変数を入力したものに、計算結果を代入して戻す場合にはアドレスを使うのが便利ですが、入力した変数はそのままで計算結果は別に保存したい場合には返値を用います。以下の fSquare という関数では二乗した計算結果を return という文で返します。

返値のある関数

```
#include <oxstd.h>
fSquare(const y){
decl y2;y2=y^2;return y2;
}
main(){
decl x,x2;
x=2;x2=fSquare(x);println("x=",x,", x^2=",x2);
}
```

返値のある関数 (出力結果)

```
x=2, x^2=4
```

3.4 関数の最大化への応用

次に応用として関数の最大化問題を考えます。関数最大化を行うライブラリを使うために maximization ライブラリを使います。ここではすでにコンパイルされたコードを使うので **#import < maximize >** という行を加えます。以下ではまず $y = -2x^2 + 4x + 1 = -2(x-1)^2 + 3$ という関数の最大値を求めます。答えは $x = 1, y = 3$ ですが、初期値を $x = 0$ から始めます。最大化する関数は引数として変数（パラメータ）の値、関数のアドレス、1階導関数のアドレス、2階導関数のアドレスをとるように作ります。以下で用いる関数最大化を行う MaxBFGS という組み込み関数では2階導関数は不要ですので、使いません。1階導関数も必ずしも必要ではないので以下の例では用いていません。従って導関数に関しては引数にリストしますが、関数の中ではでてきません。
また MaxBFGS は引数として、最大化する関数の名前、変数（パラメータ）のアドレス、関数のアドレス、ヘッシアン行列の初期値のアドレス（0 とおくと初期値は単位行列）、数値的な 1 階の導関数を使うかどうか (0/FALSE: 使わない、 1/TRUE: 使う）を用います。以下では数値的な 1 階の導関数を用いて最大化を行います。

関数の最大化

```
#include <oxstd.h>
#import <maximize>
// Maximize y=-2*x^2+4*x+1=-2*(x-1)^2+3
// Maximum: (x,y)=(1,3)
fMyfunc(const dX, const adFunc, const adScore, const adHess){
adFunc[0]=-2*dX^2+4*dX+1;  // value of function
return 1; // return value 1 if succeeded
}
main(){
decl x,dfunc;
x=0; //initial value
MaxBFGS(fMyfunc, &x, &dfunc, 0, TRUE);
print("Value of Function=",dfunc,"at x=",x);
}
```

関数の最大化 (出力結果)

```
Value of Function=
       3.0000
at x=
       1.0000
```

一般に、もし解析的な 1 階導関数やヘッシアン行列がわかれば最大化までの収束は早くなります。その方法について以下で説明します。 MaxBFGS に加えて MaxNewton という関数も使ってみます。 adScore[0],adHess[0] にそれぞれ 1 階導関数、 2 階導関数を定義してやります。

関数の最大化 (導関数を使う)

```
#include <oxstd.h>
#import <maximize>
// Maximize y=-2*x^2+4*x+1=-2*(x-1)^2+3
// Maximum: (x,y)=(1,3)
fMyfunc(const dX, const adFunc, const adScore, const adHess){
// dX: parameter, adFunc: function value (address)
// adScore
adFunc[0]=-2*dX^2+4*dX+1;  // value of function
//
 if(adScore)
 // when the last parameter of MaxBFGS = FALSE
 // or when MaxNewton is used.
 {
 adScore[0]=-4*dX+4; // println("adScore=",adScore[0]);
 }
 if(adHess)
 // when the last parameter of MaxNewton = FALSE
 {
 adHess[0]=-4; //  println("adHess=",adHess[0]);
 }

return 1; // return 1 if succeeded
}
main(){
decl x,dfunc,ir;
x=0; //initial value
MaxBFGS(fMyfunc, &x, &dfunc, 0, FALSE);
// quasi-Newton method by Broyden, Fletcher, Goldfarb, Shanno (BFGS)
// We do not need to provide analytical derivatives for MaxBFGS.
// However, if we provide analytical 1st derivatives, convergernce
// will be faster. second derivatives are not used anyway.
print("MaxBFGS: Value of Function=",dfunc,"at x=",x);
x=0;
MaxNewton(fMyfunc, &x, &dfunc, 0, FALSE);
// Newton method
// We need to provide first derivatives for MaxNewton.
// The second derivatives are optional.
print("MaxNewton: Value of Function=",dfunc,"at x=",x);
}
```

関数の最大化 (導関数を使う)(出力結果)

```
MaxBFGS: Value of Function=
       3.0000
at x=
       1.0000
MaxNewton: Value of Function=
       3.0000
at x=
       1.0000
```

最後にグローバル変数を用いた例を示します。グローバル変数は最初に **static decl** という文で宣言します。するとその変数はどこに現れても常に同じ変数と値を表します。前の例で $y = -2x^2 + 4x + 1$ という関数の最大値を求めましたが、グローバル変数を用いることによって $y = ax^2 + bx + c$ という関数の最大値を求めるアルゴリズムに一般化してみます。

グローバル変数

```
#include <oxstd.h>
#import <maximize>
static decl a,b,c;
fMyfunc(const dX, const adFunc, const adScore, const adHess){
adFunc[0]=a*dX^2+b*dX+c;  // value of function
return 1; // return value 1 if succeeded
}
main(){
decl x,dfunc;
a=-2;b=4;c=1;
x=0; //initial value
MaxBFGS(fMyfunc, &x, &dfunc, 0, TRUE);
print("Value of Function=",dfunc,"at x=",x);
}
```

3.5 別のファイルにある関数を読み込む方法

自分でよく使う関数を作った場合には例えば **mylib.h** というようなヘッダファイルに関数名を保存し, 関数の中身は **mylib.ox** としておき、 OX プログラムの include フォルダに保存しておきます。そして

```
#include <mylib.h>
#include <mylib.ox>
```

という文を追加すればプログラミングの手間を省くことができます。

もちろん include フォルダにおかずに、プログラムファイルと同じ場所に置くこともできます。その場合

```
#include "mylib.h"
#include "mylib.ox"
```

とします。

第 4 章

データの読み込みと書き出しかた

データを分析するプログラムを書く場合、データをプログラムファイルに書くのではなく、外部のファイルから読み込むことになります。また分析結果を外部ファイルに書き出したいこともあります。いろいろな方法がありますが、ここでは一部の方法を紹介します。このような場合は次のようにします。

ファイル (test1.txt)

```
1 10
2 20
3 30
```

上のテキストファイルのデータを 3×2 の行列として読み込みます。

ファイルからの入力

```
#include <oxstd.h>
main(){
decl mx,file;
 file = fopen("test1.txt");
 fscan(file,"%#m",3,2,&mx);
 fclose(file);
println(mx);
}
```

fopen で test1.txt ファイルをあけて、 file という名前にします。 fscan ではそのファイルの中味を 3×2 の行列として mx という名前で読み込みます。 `"%#m"`はファイルを行列形式として読むということを表します。また行列の名前 mx はアドレス `&mx` で指定します。読み込みが終わったら fclose で開けたファイルを閉じておきます。

すると以下のような結果が得られます。

ファイルからの入力（結果)

```
        1.0000        10.000
        2.0000        20.000
        3.0000        30.000
```

やや上のような形式で難しい場合にはエクセルで test1.txt の内容を入力し、test1.xls と名前をつけると以下のように簡単に読み込むことができます。

ファイルからの入力 (エクセルファイル)

```
#include <oxstd.h>
main(){
decl mx;
 mx = loadmat("test1.xls");
println(mx);
}
```

以下では、続けて、読み込んだ mx をファイルに書き出すプログラムを書いてみます。

ファイルへの出力

```
file = fopen("test2.txt", "w");
fprint(file,"%15.5f",mx);
fclose(file);
```

%15.5f は出力の表示形式をあらわします。つまり、出力部分に 15 文字分を割り当てて、小数点以下を 5 桁とるという意味です。小数点を表す · が一文字分使うので残りの使える部分は 9 文字分です。もともと一行で使える文字数は 80 に設定されているので増やしたい場合には format(600); などとすると 600 に増やすことができます（ただし 1024 文字が最大限）。

test2.txt

```
        1.00000        10.00000
        2.00000        20.00000
        3.00000        30.00000
```

簡単にエクセルファイルに出力することもできます。

ファイルへの出力 (エクセルファイル)

```
savemat("test2.xls",mx);
```

とすると test2.xls というエクセルファイルに保存されます。ただしエクセルの形式は Ver2.1 となり、なかは以下のようになります。

test2.xls

```
Var1 Var2
1-1 1 10
2-1 2 20
3-1 3 30
```

第 5 章

グラフを書くには

OX の無料バージョンではグラフを画面上で見る機能はありません。しかし、グラフをファイルに出力することはできるので、別のソフトウェアを使ってみることになります。出力できるファイル形式はポストスクリプト (拡張子が.eps や.ps) や GiveWin （拡張子が.gwg)、メタファイル (拡張子が.wmf) などあります。ポストスクリプト形式のファイルを見るソフトウェアには GSView があり、商用目的でなければ無料で提供されているので、インターネット上からダウンロードすることもできます（その方法は省略します）。以下では図の出力の方法だけを紹介します。

図を書くには **oxdraw.h** というヘッダを読み込んでおく必要があります。

図を書きファイルに出力する

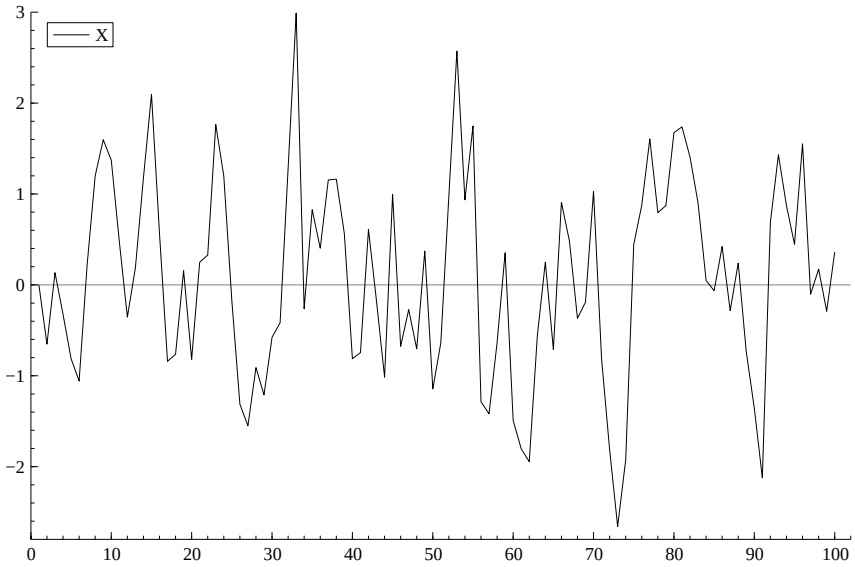
```
#include <oxstd.h>
#include <oxdraw.h>
main(){
decl t,nobs,x;
 nobs=100;
 x=zeros(nobs,1);
 for(t=1;t<nobs;++t){
 x[t]=0.5*x[t-1]+rann(1,1);
 }
// Draw Time Series Plot of X
 SetDrawWindow("draw1");
 DrawTMatrix(0,x',{"X"},1,1,1);
 SaveDrawWindow("x-tsplot.ps");
 CloseDrawWindow();
}
```

`rann(1,1)` は標準正規分布 (平均 0, 分散 1) の乱数です。まず

$$X_t = 0.5X_{t-1} + \epsilon_t, \quad t = 1, \ldots, 99, \quad X_0 = 0,$$

で ϵ_t が標準正規分布 (平均 0, 分散 1) の乱数であるような時系列データ $X_0, X_1, \ldots, X_{99}$ を作ります。その次に X の時系列プロットを x-tsplot.ps という名前のファイルに出力します。

図 5.1: X の時系列プロット (x-tsplot.ps)



第 6 章

きれいな出力をするために

やや余談になりますが、計算結果をきれいに出力するための方法を紹介します。

きれいに出力する

```
#include <oxstd.h>
main(){
decl x,y;
x=rann(100,1);y=ranu(200,1);
println("%r",{"X","Y"},
        "%c",{"Nobs","Mean","StDev","Max","Min"},
(rows(x)|rows(y))~(meanc(x)|meanc(y))~(varc(x)|varc(y))
 ~(max(x)|max(y))~(min(x)|min(y)));
}
```

100 個の標準正規乱数と 200 個の一様乱数を発生させて、個数、平均、標準偏差、最大値、最小値を出力します。`%r` は行 (row) の文字を入力することを表し、`%c` は列 (column) の文字を入力することを表します。

きれいに出力する (結果)

```
              Nobs         Mean        StDev          Max          Min
X           100.00     0.036508      0.89381       2.3465      -2.1607
Y           200.00      0.53112     0.084316      0.99883     0.011271
```

第 7 章

オブジェクト指向プログラミング

7.1 オブジェクト指向プログラミングとは

ここではオブジェクト指向プログラミングの考え方を紹介します。基本的には C++ や Java などの言語と同じ構成なので、それら言語の入門にもなります。

さてオブジェクト指向プログラミングでは「クラス (class)」と「オブジェクト (object)」という概念が基本となります。クラスとオブジェクトの関係は、自動車で例えると「自動車の設計図」と「製造された自動車」という関係になります。クラスはプログラムの基本的な動作に関する規則を決めたもので関数や変数のかたまりです。 main 文のなかでクラス（設計図）が呼び出されるとオブジェクト（製品）として使うことができるようになります。

7.2 クラス (class) とは

クラスとはどんなものか、一番簡単な例を見てみましょう。

まず class と書いて次にクラス名を書きます。ここではクラス名を Nissan とします。その後は { と }; のなかにクラスで使うメンバを宣言します (; を忘れないように気をつけましょう）。その役割はヘッダファイルと同じなのでこの部分だけヘッダファイルとして分離することもあります。

クラスとは

```
#include <oxstd.h>
class Nissan
{
   Nissan();    // constructor
   ~Nissan();   // destructor
   Rental();
   Dealer();
   decl m_price1, m_price2;
};
```

最初の Nissan() はコンストラクタと呼ばれ、クラスが呼ばれたときに変数や関数の初期化を行い, オブ

ジェクトを作る役割を持ちます。コンストラクタの名前はクラス名と同じで、引数をとることもできます。コンストラクタの文を省略すると自動的に Nissan() というコンストラクタになります。同様に~Nissan() はデストラクタといい、オブジェクトを削除する役割を持ちます。デストラクタの名前はコンストラクタの名前に ~ をつけたものになります。オブジェクトを削除する際に、他に何もすることがなければデストラクタの文を省略することもできます。

次に変数と関数という 2 つの種類のメンバを宣言します。次の Rental();, Dearler(); はクラスの中で後に定義される関数名で、メンバ関数（あるいはメソッド、 method) と呼ばれます。また同様に m_price1, m_price2 はクラスの中ならばどこでも使うことができる変数で、データメンバ (data member) といいます。 OX ではこれらのメンバはオブジェクトとして使うことができます。

クラスとは（続き）

```
Nissan::Nissan()
{
println("Welcome to Nissan.");
m_price1=10000;
m_price2=1000000;
}
Nissan::~Nissan()
{
println("Goodbye Nissan.");
}
Nissan::Rental()
{
println("Nissan Rent-A-Car. Price:",m_price1,"Yen.");
}
Nissan::Dealer()
{
println("Nissan for Sale. Price:",m_price2,"Yen.");
}
```

クラスの中の関数はクラス名:: 関数名というように:: でつなげて書き始めます。まずコンストラクタですがコンストラクタ名はクラス名と同じなのでNissan::Nissan となります。オブジェクトを作るときにWelcome to Nissan. というメッセージを流し、二つの変数m_price1, m_price2 をそれぞれ 10000,1000000 に設定します。一方、デストラクタではオブジェクトを削除するときにGoodbye Nissan. というメッセージを流すということを行います。この他の関数についても同様に定義することができRental() やDealer ではそれぞれレンタカーは 10000 円で販売価格は 1000000 円というメッセージを流す関数に定義しています。これでクラスができました。では実際にどのように使うのでしょうか。

クラスとは（続き）

```
main(){
decl car;
 car=new Nissan();
 car.Rental();
 car.Dealer();
 delete car;
}
```

まずオブジェクトのための変数をcar と宣言し、 car=new Nissan(); で Nissan クラスのオブジェクトとして作ります。すると Nissan クラスの関数は、 car.Rental(); やcar.Dealer(); のようにオブジェクト名. クラスの関数名として使うことができます。オブジェクトを使わなくなったらdelete を使ってデストラクタを呼び出して削除します。そうすることによってプログラムの使うメモリが開放されて、速やかな計算が可能になります。

クラス（結果）

```
Welcome to Nissan.
Nissan Rent-A-Car. Price:10000Yen.
Nissan for Sale. Price:1000000Yen.
Goodbye Nissan.
```

まず car=new Nissan(); でコンストラクタNissan() が呼ばれたので変数を初期化するとともに Welcome to Nissan. をプリントします。次にcar.Rental();,car.Dealer(); の関数で指定された内容を印刷します。最後にデストラクタ~Nissan() が呼ばれたのでGoodbye Nissan. と印刷して終わります。
上の例では引数のないクラスでしたが、自由に引数を設定することもできます。以下では簡単な例を示すことにしましょう。以下ではデータの個数, 平均, 標準偏差, 最大値, 最小値を計算するプログラムを書いてみます。最初のdX はデータで、コンストラクタの引数になります。class で始まる文ではメンバ関数として要約統計量を出力するSummary とクラスのバージョンを印刷するVersion, またデータメンバとしてdX を保存するためにm_dx を宣言します。文の終わりにはかならず; を忘れないようにしましょう。コンストラクタは引数dX をとるのでBaseStat(dX) とします。そして他のメンバ関数がデータを使えるようにdX をデータメンバm_dx に代入しておきます。ここでは特にデストラクタを書いていませんが、自動的に~BaseStat() となり、またオブジェクトが削除されるときも削除以外は特に何も実行しないということになります。メンバ関数Summary ではデータメンバを使って要約統計量を出力し、Version ではこれが BaseStat のクラスであることとそのバージョンを印刷します。

クラスの例（その２）

```
#include <oxstd.h>
class BaseStat
{  BaseStat(const dX);    // constructor
   Summary();
   Version();
   decl m_dx;
};
BaseStat::BaseStat(const dX)
{ m_dx=dX;}
BaseStat::Summary()
{ Version();
  println("%c",{"Nobs","Mean","StDev","Max","Min"},
  rows(m_dx)~meanc(m_dx)~varc(m_dx)^0.5~max(m_dx)~min(m_dx));
}
BaseStat::Version()
{println("This is BaseStat Class Version 1.0.");}
```

main 文でのオブジェクトの作り方は同じですが、引数としてデータ x を指定します。ここではデータとして 100 個の標準正規乱数を作ってみました。

クラスの例（その２）続き

```
main(){
decl x,test;
 x=rann(100,1);
 test=new BaseStat(x);
 test.Summary();
 delete test;
}
```

クラスの例（その２）（結果）

```
This is BaseStat Class Version 1.0.
         Nobs          Mean         StDev           Max           Min
       100.00      0.036508       0.94541        2.3465       -2.1607
```

7.3 派生クラス (derived class) とは

すでにあるクラスをもとにして新しいクラスを作ることを継承 (Inheritance) するいいます。もとのクラスを基底クラス (base class) といい、もとのクラスに基づいて作られるクラスをもとのクラスから派

生したクラスということで派生クラス (derived class) といいます。派生クラスでは基底クラスのメンバ関数や変数をすべて使うことができます。上の例を使って BaseStat というクラスを基底クラスとして、MyStat という派生クラスを作ってみましょう。派生クラスは **class 派生クラス名：基底クラス名**と書きます。従って`class MyStat:BaseStat` という文で始まりますが、基底クラスと異なり終わりにセミコロンはいらず、`{ }` のなかにプログラムの文を書いていきます。`class` 文の中では派生クラスのコンストラクタ、メンバを宣言します。次に派生クラスのコンストラクタでは必ず基底クラスのコンストラクタを呼び出しておきます。あとは通常のクラスと同じようにプログラムします。またこの例では派生クラスのメンバ関数として、`MySummary` というデータの個数, 平均, 標準偏差だけを計算する簡略版要約統計量の関数と、派生クラスのバージョンを印刷する`Verion` という関数を定義します。

派生クラスの例

```
class MyStat:BaseStat // derived class
{ MyStat(const dX); // constructor
  Version();MySummary();
}
MyStat::MyStat(const dX)
{ BaseStat(dX);}
MyStat::MySummary()
{ Version();
  println("%c",{"Nobs","Mean","StDev"},
  rows(m_dx)~meanc(m_dx)~varc(m_dx)^0.5);
}
MyStat::Version()
{println("This is MyStat Class Derived from BaseStat.");}
```

main 文では通常のクラスと同じように呼び出すことができます。また基底クラスの関数も呼び出すことができます。

派生クラスの例 (続き)

```
main(){
decl x,test;
 x=rann(100,1);
 test=new MyStat(x);
 test.MySummary();
 test.Summary();
 delete test;
}
```

結果は最初に簡略版、次にオリジナル版がでてきます。

派生クラスの例 (結果)

```
This is MyStat Class Derived from BaseStat.
         Nobs          Mean         StDev
       100.00     0.036508      0.94541
This is BaseStat Class Version 1.0.
         Nobs          Mean         StDev           Max           Min
       100.00     0.036508      0.94541        2.3465       -2.1607
```

7.4 オーバーライド (override)

前節の例で見ましたが、時として基底クラスと派生クラスで同じ名前の関数が出てくることがあります。例ではVersion 関数がこれにあたります。もし、派生クラスで呼び出すときにすべてのVersion 関数は新しく定義した関数で上書きしたい場合はどうすればよいでしょうか？そのためには基底クラスの中のVersion 関数をvirtual Version(); というように仮想関数にすれば可能になります。

オーバーライド

```
      . . . .
class BaseStat
{  BaseStat(const dX);    // constructor
   Summary();
   virtual Version();
   decl m_dx;
};
      . . . .
```

そうすると派生クラスのプログラムの実行結果は

オーバーライドした派生クラスの例 (結果)

```
This is MyStat Class Derived from BaseStat.
         Nobs          Mean         StDev
       100.00     0.036508      0.94541
This is MyStat Class Derived from BaseStat.
         Nobs          Mean         StDev           Max           Min
       100.00     0.036508      0.94541        2.3465       -2.1607
```

となります。このように新しく定義した関数で以前の関数を上書きすることをオーバーライド (override) するといいます。もし、以前の関数も新しい関数も使いたいのであれば, クラス名を含めてBaseStat::Version();, MyStat::Version(); というようにクラス名:: 関数名; としてやれば使うことができます。