Latent Variable Modeling and Statistical Computing
Spring 2016

A Quick Introduction to Ox

Materials are mostly lifted from *../OxMetrics7/ox/doc/OxIntro.pdf*

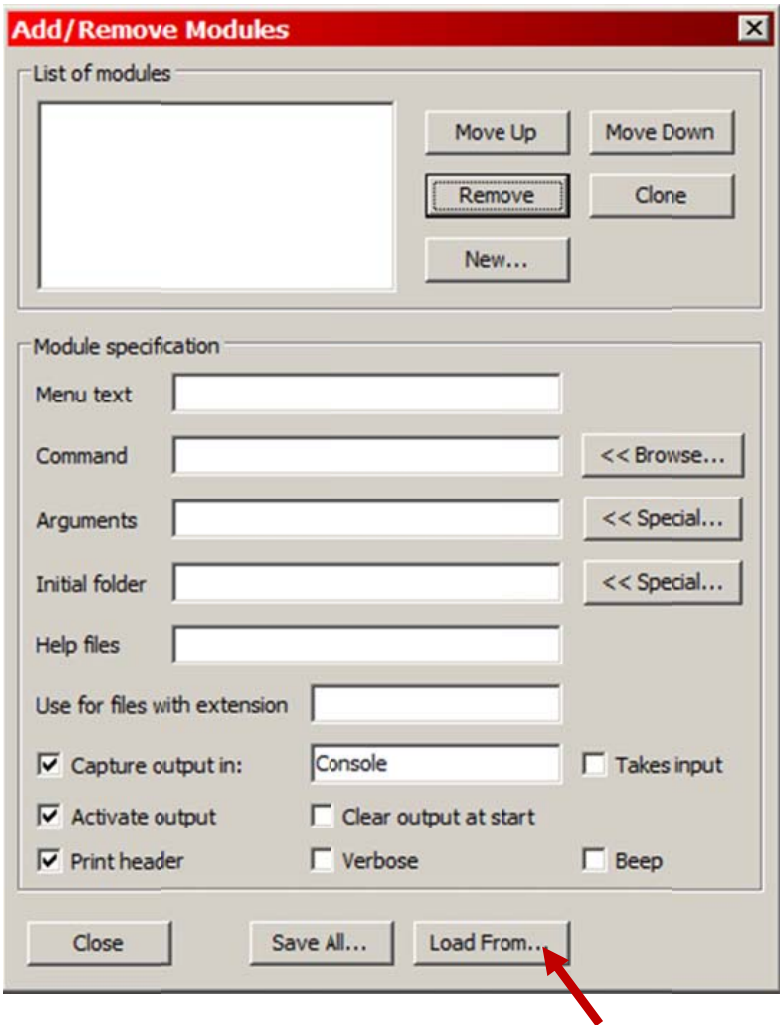
Chapter 1: Ox Environment

Installation

The latest version of Ox console and OxEdit can be downloaded at

`www.doornik.com/download/oxmetrics7/Ox_Console/`.

The latest version (i.e., Ox 7 Console) used to require that you specify certain modules before you can run your code. To do so, choose Add/Remove Modules in the Tools menu.

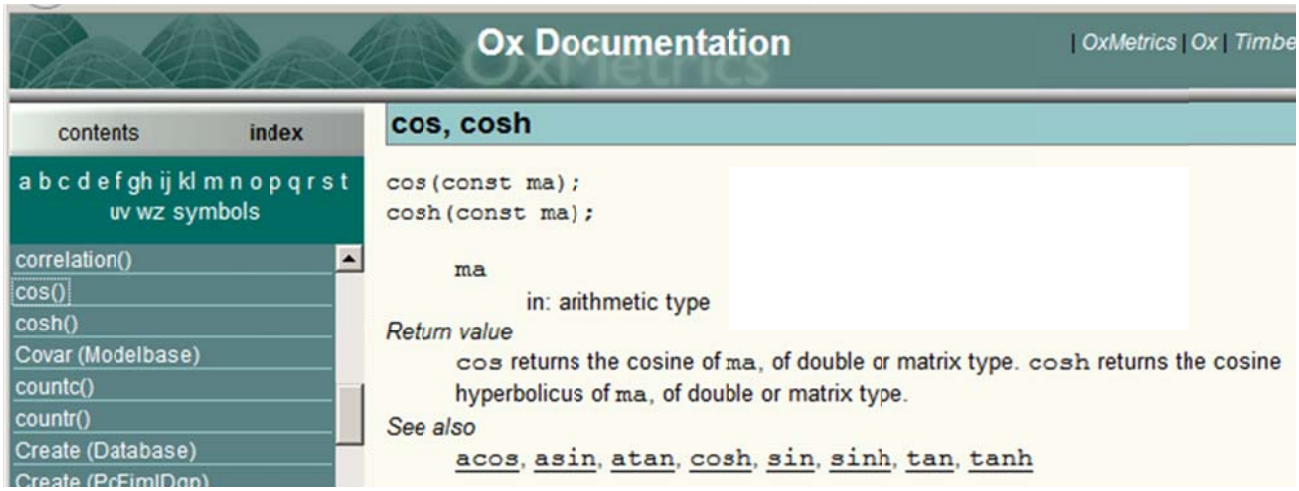


Load the file OxCons.tool, which can be found in *../OxMetrics7/ox/bin/OxEdit*.

Help and Documentation

The help files can be found in the ../ox/doc/index.html. To access the master file, use Ox Help.

The entries at the top give access to the table of contents, and to the index. The capture below shows the help on `cos` - the cosine function.



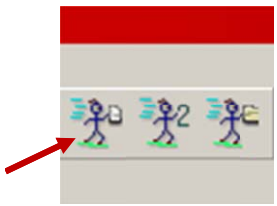
Using OxMetrics and OxRun

Ox Console works with OxMetrics, and offers some very useful features for developing and running Ox programs. The first thing to note when opening ../OxMetrics/ox/samples/myfirst.ox using Open in OxMetricss File menu is the syntax highlighting:

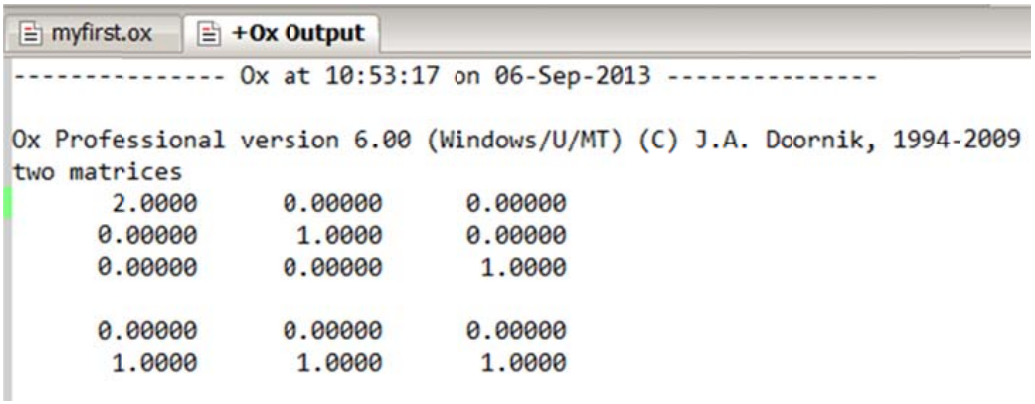
```
1   #include <oxstd.h> // include the Ox standard library header
2
3   main()                  // function main is the starting point
4   {
5       decl m1, m2;            // declare two variables, m1 and m2
6
7       m1 = unit(3);     // assign to m1 a 3 x 3 identity matrix
8       m1[0][0] = 2;                   // set top-left element to 2
9       m2 = <0,0,0;1,1,1>;//m2 is a 2 x 3 matrix, the first row
10                        // consists of zeros, the second of ones
11
12      print("two matrices", m1, m2);      // print the matrices
13  }
```

There are several features to make programming easier. Unmatched parentheses are shown in red (forgetting a closing) or } is quite a common mistake). You can select of block of lines, and then use `Comment In/Comment Out` to make temporary changes (there is unlimited undo/redo as well).

To run a file, you can click on the `Run` button on the top toolbar:



The output will appear in a new window, entitled +Ox Output.



Compilation and Run-Time Errors

Program statements are processed in two steps, resulting in two potential types of errors:

(1) Compilation errors

The statements are scanned in and compiled into some kind of internal code. Errors which occur at this stage are compilation errors. No statements are executed when there is a compilation error. Compilation errors could be caused by undeclared variables, wrong number of function arguments, forgetting a semicolon at the end of a statement (among many other reasons). For example, these two messages are caused by one undeclared variable at line 10 of the program:

```
myfirst.ox (10): y undeclared identifier
myfirst.ox (10): lvalue expected
```

Occasionally, a syntax error leads to a large list of error messages. Then, correcting the first mistake could well solve most of the problem.

(2) Run-time errors

When the code which does not have syntax errors is executed, things can still go wrong, resulting in a run-time error. An example is trying to multiply two matrices which have non matching dimension. Here, this happened at line 10 in the main function:

```
Runtime error: matrix[3][3] * matrix[2][3] bad operand
Runtime error occurred in main(10), call trace:
myfirst.ox (10): main
```

Chapter 2: Syntax

Introduction

This chapter gives a brief overview of the main syntax elements of the Ox language. The most important features of the Ox language are:

- The matrix is a standard type in Ox. You can work directly with matrices, for example adding or multiplying two matrices together. Ox also has the standard scalar types for integers (type int), and real numbers (type double). A vector is a matrix with one column or one row, whereas a 1×1 matrix is often treated as a scalar. Ox will keep track of matrix dimensions for you.
- Variables are implicitly typed. So a variable can start as an integer, then become a 10×1 matrix, then a 2×2 matrix. and so on. As in most languages, variables must be explicitly declared.
- Ox has strings and arrays as built-in types, to allow for higher dimensional matrices, or arrays of strings.
- Ox has an extensive numerical and statistical library.
- Ox allows you to write object-oriented code (this is an optional feature).

The syntax of Ox is modeled on C and C++ (and also Java), so if you are familiar with these languages, you will recognize the format for loops, functions, etc. However, prior knowledge of these language is not assumed.

Comment

Ox has two types of comment: `/* . . .*/` for blocks of comment, and `//` for comment up to the end of a line.

When writing functions, it is useful to add comment to document the function, especially the role of the arguments, and the return value.

Program Layout

Our first complete program is:

```
#include <oxstd.h>

main()
{
    print("Hello world");
}
```

This program does only print a line of text, but is worth discussing anyway:

- The first line includes a header file. The contents of the file oxstd.h are literally inserted at the point of the `#include` statement. The name is between `<` and `>` to indicate that it is a standard header file: it actually came with the Ox system. The purpose of that file is to declare all standard library functions, so that they can be used from then onwards.
- This short program has one function, called `main`. It has no arguments, hence the empty parentheses (these are compulsary). An Ox program starts execution at the `main` function; without `main`, there might be a lot of code, but nothing will happen.
- A block of code (here the *function body* of the `main` function), is enclosed in curly braces.

Most Ox code presented here uses syntax coloring: comment is shown as italic, and reserved words (also called keywords) are bold. Both OxMetrics and OxEdit use syntax coloring. This is a purely visual aide, the actual Ox code is a plain text file.

Statements

Statements are commands to do something, for example, some computation. Important ingredients of statements are variables, to store or access a result, and operators (discussed in the next chapter) to combine existing results into new ones. A statement is terminated with a semicolon (`;`). Please note that, when you are copying a code from a paper, Ox makes a distinction between lower case and upper case letters.

```
#include <oxstd.h>

main()
{
    decl n, sigma, x, beta, eps;
    n = 4; sigma = 0.25;
    x = 1 ~ ranu(n, 2);
    beta = <1; 2; 3>;

    eps = sigma * rann(n, 1);
    print("x", x, "beta", beta, "epsilon", eps);
}
```

Some remarks on this program:

- `decl` is used to declare the variables of this program.
- `n = 4` simply assigns the integer value 4 to the variable n.
- `sigma = 0.25` simply assigns a real value.
- `;` terminates each statement.
- `ranu` and `rann` are library functions for generating uniform and normal random numbers.
- `print` is a standard library function used for printing.
- `*` multiplies two variables.
- `~` concatenates two variables. Here we concatenate an integer with a 4×2 matrix.

Exercise

As a first exercise run the Ox program listed above. This code, **oxtut2c.ox**, can be found in the **ox/tutorial** folder.

Use the help system to discover the meaning of `ranu` and the difference between `print` and `println`.

Add a line for computing $y = X\beta + \epsilon$. Also print the value of y. This requires:
(1) declaring the variable y;
(2) inserting a statement computing $y = X\beta + \epsilon$ and storing it in y; and
(3) adding a statement to print the y variable.

The `rows()` and `sizer()` functions returns the number of rows of a matrix, the `columns()` and `sizec()` functions the number of columns. Add a print statement to report the dimensions of x in the above program.

Here are some of the things which can go wrong in the previous exercises:

(1) Forget a comma. For example `decl a, b c;` needs a comma after the `b`.

(2) Forget a semicolon, as for example in: `n = 4 sigma = 0.25;`

(3) Adding a semicolon after the function header, as in:

```
main();
{
}
```

(4) Omitting the curly braces, as in:

```
main()
{
     print("some text");
```

(5) Forget to declare a variable. The new `y` variable must be declared before it can be used.

(6) Any typo, such as writing `priny` instead of `print`.

(7) In some parts of the world a comma is used as a decimal separator. Ox uses a dot. Perhaps surprisingly at this stage, this is valid code: `sigma = 0,25;`

Identifiers

Identifiers (names of variables and functions) may be up to 60 characters long. They may consist of the characters [A-Z], [a-z], [_], and [0-9], but may not start with a digit.

Style

It is useful to raise the issue of programming style at this early stage. A consistent style makes a program more readable, and easier to extend. Even in the very small programs of these tutorials it helps to be consistent. Often, a program of a few lines grows over time as functionality is added. With experience, a good balance between conciseness and clarity will be found.

Later, we shall introduce a system of name decoration, which will increase the readability of a computer program. For example, we would prefix all global variables with `g_`, such as `g_dMisval` (but we shall do our best to avoid global variables as much as possible).

Matrix Constants

The previous code used various types of constants: 4 is an integer constant, 0.25 is a double constant, and "x" is a string constant. Most interesting is the value assigned to beta, which is a matrix constant. This is a list of numbers inside `<` and `>`, where a comma separates elements within a row, and a semicolon separates rows. Remember that you can only use numbers in a matrix constant, no variables: `<1,2,sigma>` is illegal. In that case use `1~2~sigma`.

Creating a Matrix

There are several ways to create a matrix in Ox, as the following examples show.

(1) Using matrix constants

(2) Reading matrices directly from disk

(3) Using library functions such as `unit`, `zeros`, `ones`:

Some of the relevant library functions are:

- `zeros(r,c)` creates an $r \times c$ matrix of zeros;
- `ones(r,c)` creates an $r \times c$ matrix of ones;
- `unit(r)` creates an $r \times r$ identity matrix;
- `constant(d,r,c)` creates an $r \times c$ matrix filled with ds;
- `range(i,j)` creates an $1 \times j - i + 1$ matrix with $i, i + 1, \ldots, j$;
- `diag(x)` creates an square matrix with with the elements of x in the diagonal.

(4) Using matrix concatenation

Example: `x = (0 ~ 1) | (2 ~ 3);`

Concatenation works as follows: `~` is for horizontal concatenation, and `|` is for vertical concatenation. The above line will create the following matrix:

$$\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}.$$

The code uses parentheses to ensure that the two horizontal concatenations are done first. Ox always does horizontal before vertical concatenation, so the parentheses are redundant. When in doubt it is better to write them anyway.

Chapter 3: Operators

Introduction

A language like Ox needs at least three kinds of operators to work with matrices:

(1) index operators, to access elements in a matrix;
(2) matrix operators, for standard matrix operations such as matrix multiplication;
(3) dot operators, for element by element operations.

There are quite a few additional operators, for example to work with logical expressions these are discussed later in this chapter.

Index Operators

Previously, we learned various ways to create a matrix. The counterpart is the extraction of single elements or specific rows or columns from the matrix. Ox has a flexible indexing syntax to achieve this. But first:

Indexing in Ox starts at zero, not at one!

Initially you might forget this and make a few mistakes, but before too long it will become second nature. Ox has adopted this convention for compatibility with most modern languages, and because it leads to faster programs. There is an option to start at index one, which is explained in the Ox manual (and not really recommended). The available indexing options are:

- Single element indexing

  A matrix usually has two indices: `[i][j]` indexes element (i, j) of a matrix, where `[0][0]` is the first (top left) element.

- Range indexing

  Either `i` or `j` may be replaced by a range, such as `i1:i2`. If the lower value of a range is missing, zero is assumed; if the upper value is missing, the upper bound is assumed.

- Empty index

  The empty index `[]` selects all rows (when it is the first index) or all columns (when it is the second). When there is only one index `[]` the result is a column vector, filled by taking the elements from the indexand row by row.

- Indexing a vector (i.e. a matrix with one row or one column)

  When a matrix is used with one index, it is treated as a vector. In that case it does not matter whether the vector is a row or a column vector.

- Using a matrix as an index

  In this case the matrix specifies which rows (if it is the first index) or columns (for the second index) to select. The elements in the indexing matrix must be integers (if not, they are truncated to integers by removing the fractional part).

Here are some examples:

$$x = \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{pmatrix}, \; y = \begin{pmatrix} 0 & 1 & 2 \end{pmatrix}, \; z = \begin{pmatrix} 0 \\ 3 \\ 6 \end{pmatrix}$$

$$x[0][0] = 0, \; x[][1:] = \begin{pmatrix} 1 & 2 \\ 4 & 5 \end{pmatrix}, \; x[1][y] = \begin{pmatrix} 3 & 4 & 5 \end{pmatrix},$$

$$y[:1] = y[0][:1] = \begin{pmatrix} 0 & 1 \end{pmatrix}, \; z[:1] = z[:1][0] = \begin{pmatrix} 0 \\ 3 \end{pmatrix}.$$

Write a program to verify these examples.

Write a program which creates a 4×4 matrix of random numbers. Then extract the 2×2 leading submatrix using (a) range indexing and (b) matrix indexing.

Matrix Operators

All operators + - * / work as expected when both operands are an integer or a double: when both operands are an integer, the result is an integer, otherwise it will be a double. The exception is division of two integers: this will produce a double, so 1/2 equals 0.5 (C and C++ programmers take note!).

When matrices are involved, things get more interesting. Obviously, when two matrices have the same size we can add them element by element (+), or subtract them element by element (-). Although not a standard matrix algebraic operation, adding a column vector to a row vector works is allowed in Ox (and at times very useful). It works like a table:

$$\begin{pmatrix} x_0 & x_1 \end{pmatrix} + \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_0 + y_0 & x_1 + y_0 \\ x_0 + y_1 & x_1 + y_1 \\ x_0 + y_2 & x_1 + y_2 \end{pmatrix}.$$

For matrix multiplication use *, then element i, j of the result is the inner product of row i (left operand) and column j (right operand).

Division (/), when the right operand is a matrix, corresponds to post multiplication with the inverse.

In some applications, (e.g., creating the constant term for regression), when concatenating an integer (or double) and a matrix, the scalar is automatically replicated to get the same number of rows (~) or columns (|). When concatenating two non-matching matrices, the shortest one is padded with zeros at the end. (So there is a difference between `1 ~ <1;1>` and `<1> ~ <1;1>` ; a warning is printed for the latter.)

A square matrix may be raised to an integer power, using ^, for example `A2` equals `A*A`. To summarize:

Operator	Operation
'	transpose, `X'y` is short for `X'*y`
^	(matrix) power
*	(matrix) multiplication
/	(matrix) division
+	addition
-	subtraction
~	horizontal concatenation
|	vertical concatenation

Some operations are illegal, resulting in an error message. Here is an example:

```
Runtime error: matrix[4][1] * matrix[3][4] bad operand
Runtime error occurred in main(16), call trace:
C:\Program Files\Ox\tutorial\oxtut3a.ox (16): main
```

The first says that we cannot multiply a 4×1 matrix into a 3×4 matrix. The error occurred in the `main` function, at line 16. In OxEdit or OxMetrics you can double click on the line with the error to jump directly to the problematic code.

Dot Operators

Dot operators are element-by-element operators. For adding and subtracting matrices there is only the dot version, already used in the previous section (written as + and -). Element-by-element multiplication is denoted by `.*` and `./` is used for element-by- element division. As with addition and subtraction, dot conformity implies that either operand may be a row (or column) vector. This is then swept through the rows (columns) of the other operand. For example:

$$\begin{pmatrix} x_0 & x_1 \end{pmatrix} .* \begin{pmatrix} y_0 & y_1 \\ y_2 & y_3 \\ y_4 & y_5 \end{pmatrix} = \begin{pmatrix} x_0 y_0 & x_1 y_1 \\ x_0 y_2 & x_1 y_3 \\ x_0 y_4 & x_1 y_5 \end{pmatrix}.$$

To summarize:

Operator	Operation
`.^`	element-by-element power
`.*`	element-by-element multiplication
`./`	element-by-element division
`+`	addition
`-`	subtraction

Relational and Equality Operators

Relational operators compare both operands, and exist in matrix version and in element by element (or dot) version. The first version always returns an integer, even when both arguments are matrices. The return value 0 stands for FALSE, and 1 for TRUE. When comparing a matrix to a scalar, the result is only 1 (TRUE) if it holds for each element of the matrix.

Operator	Operation
`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to
`==`	is equal
`!=`	is not equal

The second form of relational operator is the dotted version: this does an element by element comparison, and returns a *matrix* of 0s and 1s. The dotted versions are:

Operator	Operation
`.<`	element-by-element less than
`.>`	element-by-element greater than
`.<=`	element-by-element less than or equal to
`.>=`	element-by-element greater than or equal to
`.==`	element-by-element is equal
`.!=`	element-by-element is not equal

Often code is more readable when using the predefined constant TRUE and FALSE, instead of the numbers 1 and 0. These are defined in oxstd.h. Relational operators are especially important in conditional expressions and loops, and these are discussed in the next chapter.

Logical Operators

These are closely related to the relational operators, and also have non-dotted and dotted versions. The first evaluate to either zero or one:

Operator	Operation
`&&`	logical-and
`||`	logical-or

If an expression involves several logical operators after each other, evaluation will stop as soon as the final result is known. For example in (`1 || checkval(x)`) the function `checkval` is never called, because the result will be true regardless of its outcome. This is called a *boolean shortcut*.

The dotted versions perform the logical expression for each element when matrices are involved (therefore they cannot have boolean shortcuts):

Operator	Operation
`.&&`	element-by-element logical-and
`.||`	element-by-element logical-or

Some procedures are available for selecting or dropping rows/columns based on a logical decision. These are `selectifr`, `selectifc`, `deleteifr` and `deleteifc`; `vecindex` may be used to translate the 0-1s to indices. A very useful, but slightly more complex operator is the dot-conditional operator.

Here are some examples using these functions:

Expression	Outcome
`u`	`1 0 1 0 2`
`u .> 0`	`1 0 1 0 1`
`vecindex(u)`	`0 2 4`
`vecindex(u .> 1)`	`4`
`selectifc(u, u .> 0)`	`1 1 2`
`selectifc(u, u .> 1)`	`2`

Ox has more operators (e.g., conditional operators). See the Ox book for the full list.

Chapter 4: Input and Output

The program below shows a way of reading elements of an external file and creating an output file.

The output file `first.out` is a 4×1 matrix containing the squared elements of the first column of the 4×2 matrix x;

The output file is read into the vector y.

(To the best of my knowledge) "`%#M`" specifies the matrix to be printed row by row with the default format.

Details about print formatting can be found under `print` of the Ox documentation.

Specifying paths can be avoided by keeping the program and the related files in the same folder.

```
#include <oxstd.h>

main()
{

decl x=rann(4,2);
print("x =", x);

decl file, tmp;

tmp=sprint("first.out");
file=fopen(tmp,"w");
fprint(file,"%#M",x[][0].^2);
fclose(file);

decl y;
file=fopen(tmp);
fscan(file,"%#M",4,1,&y);
fclose(file);

print("y =", y);

}
```

```
#include <oxstd.h>
main()
{
decl x=rann(4,2);
print ("x =x", x);
decl file, tmp;
tmp=sprint("first.out");
file=fopen(tmp,"w");
fprint(file,"%#M",x[][0].^2);
fclose(file);
decl y;
file=fopen(tmp);
fscan(file,"%#M",4,1,&y);
fclose;
print("y =",y);
}
```

Chapter 5: Program Flow and Program Design

Introduction

Ox is a complete programming language, with `if` statements and `for` loops. However, where you need loops in more traditional languages, you can often use the special matrix statements available in Ox. Try to avoid loops whenever you can: the vectorized version will often be very much faster than using loops. On the other hand, youll discover that loops cannot be avoided altogether: some code just doesnt vectorize (or the resulting code might get too complex to maintain).

`for` *Loops*

The `for` loop has the following syntax:

```
for ( initialization ; condition ; incrementation )
{
      statements
}
```

For example:

```
decl i;

for (i = 0; i < 4; i++)
{
     print(`` '', i);
}
```

Prints: `0 1 2 3`.

At the end of the loop, `i` will have the value 4. Since the condition is checked prior to executing the loop, it is possible that the body is not executed at all (then `i` will have the initial value).

It is allowed to have more than one statement in the initialization or incrementation part of the for loop. A comma is then required as a separator.

A loop within a loop is also allow.

Write a function which multiplies two matrices, $A_{3\times2}$ and $B_{2\times4}$, using `for` loops. Implement the multiplication using a) a two-tiered loop and b) a three-tiered loop.

Compare the results with using the matrix multiplication operator.

`while` *Loops*

The example for the `for` loop can also be written using a `while` loop:

```
decl i=0;
```

```
    while (i < 4)
    {
         print(`` '', i);
         i++;
    }
```

In this case, the `for` loop is more readable. But if there is not a clear initialization or incrementation part, the `while` form might be preferred.

Again, the `while` loop is not executed at all when i starts at 4 or above. If a loop must be executed at least once, use the `do while` loop:

`break` *and* `continue`

Two special commands are available inside loops:

```
    break;
```

Terminates the loop in which the command appears, for example:

```
    for (i = 0; i < 4; i++)
    {
         if (i==2) break;
         print(`` '', i);
    }
```

Prints: `0 1`

```
    continue;
```

Starts with the next iteration of the loop, for example:

```
    for (i = 0; i < 4; i++)
    {
         if (i==2) continue;
         print(`` '', i);
    }
```

Prints: `0 1 3`

Conditional Statements

In the previous section we used `if` statements to illustrate the use of `continue` and `break`. The full syntax is:

```
    if (condition)
    {
        statements
    }
    else if (condition)
```

```
{
    statements
}
else
{
    statements
}
```

When using if-else syntax, the conditions are checked sequentially. For example, the condition for `else if` is only examined if the condition for `if` is not satisfied; in the same manner, when the condition for `else if` in not satisfied, `else` statement is automatically implemented.

```
decl x=<1,-2,-1,0,2>;

for (decl i = 0; i < 5; i++)
{
    if (x[i]<0) print (" ","1");
    else if (x[i]==0) print (" ","2");
    else print (" ","3");
}
```

Prints: `3 1 1 2 3`

How would you implement the same program using three `if` statements?

Would this implementation be more efficient? Why or why not?

Vectorization

The program below draws n observations from a standard normal distribution, and computes the mean of the positive numbers ONLY.

The mean of positive numbers are computed two ways: using `for` loop and `if` statement, and using `selectifr`.

The time each procedure takes to compute the mean is kept tracked by `timer()`.

The `meanc(x)` function returns a $1 \times c$ matrix holding the means of the columns of x. The `"meanr"`function operates on the rows.

In the above program, which of the two procedures is more efficient? How much faster is it? You might need to increase the size of n to get more stable estimates.

This example demonstrates the advantage of using vector functions over loops.

```
main()
{
decl n=100000;

decl x=rann(n,1);
decl count,total,time;
count=total=0;

time=timer();//save starting time

for(decl i=0;i<n;i++){
if(x[i]>0){
count=count+1;
total=total+x[i];
}//end if
}//end i

decl t1=timespan(time);

time=timer();//save starting time again

x=selectifr(x,x.>0);
decl t2=timespan(time);

print("Time 1 = ",t1,"\n");
print("Time 2 = ",t2,"\n");
print("Mean =: ",total/count~meanc(x));
}
```

```
main()
{
decl n=100000;
decl x=rann(n,1);
decl count,total,time;
count=total=0;
time=timer();//save starting time
for(decl i=0;i<n;i++){
if(x[i]>0){
count=count+1;
total=total+x[i];
}//end if
}//end i
decl t1=timespan(time);
```

```
time=timer();//save starting time again
x=selectifr(x,x.>0);
decl t2=timespan(time);
print("Time 1 = ",t1,"\n");
print("Time 2 = ",t2,"\n");
print("Mean =: ",total/count~meanc(x));
}
```

Additional Exercises

1. Finding $\sqrt{S}$: The square-root of S can be solved using the so-called Babylonian method. This is an iterative solution where the n^{th} estimate is equal to

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right).$$

a) Write a program that provides an estimate for $n = 10$, where $x_0 = S$. How close is the approximate to the exact value (i.e., $\sqrt{S}$).

b) Write a program that stops when $|x_n - \sqrt{S}| \leq 0.0001$. How many iterations are needed when $S = 2$? $S = 20$? $S = 200$?. (Note that Ox does not have an absolute value function. However, you can recall that $|x| = \sqrt{x^2}$.)

2. The BoxMuller transformation is a method for generating pairs of independent random numbers from $N(0, 1)$, given random numbers from $U(0, 1)$. Given U_1 and U_2 are independent random variables from $U(0, 1)$, if we let

$$Z_1 = \sqrt{-2\log(U_1)}\cos(2\pi U_2)$$

and

$$Z_2 = \sqrt{-2\log(U_1)}\sin(2\pi U_2),$$

then Z_1 and Z_2 are independent random variables from $N(0, 1)$.

a) Generate n random number from $U(0, 1)$, and designate it as U_1; do the same for U_2. Transform U_1 and U_2 into Z_1 and Z_2. (Note that Ox has a predefined constant for π. This constant requires the inclusion of `oxfloat.h`. Also, log is the natural logarithm.)

b) Find the correlation between Z_1 and Z_2 to examine if the two random variables are independent.

c) Let $Z_{2n\times 1}$ be the concatenation of Z_1 and Z_2. Find its mean and variance. Based on these statistics, is it reasonable to assume that $Z \sim N(0, 1)$?

3. By definition, the statistic $Q = \sum_{i=1}^{n} Z_i^2$, where Z_i is identically and independently distributed as $N(0, 1)$ is said to be $Q \sim \chi^2(n)$. The mean and variance of Q are $E(Q) = n$ and $V(Q) = 2n$, respectively.

a) Generate n random number from $N(0,1)$, and compute Q.

b) Repeat the process N times.

c) Find the mean and variance of the N Q statistics. Are they close to what you would expect them to be? Again, N may need to be large for the sample statistics to be stable.

d) Can this be done without using any loop?

Functions - Part I

Using Functions

The function is a fundamental building block when writing Ox programs. Functions allow for splitting complex tasks up in manageable bits. The best ones are those which only interact with the outside via the arguments (the inputs) and the return value (the outputs, if any). Then, when there are no external variables used inside the function, the function can be treated as an isolated piece of code: the only thing which matters is the documentation of the function.

Up to this point, only one function has been used, the main function. Execution of an Ox program starts at main, from which other functions are called; there is no action outside functions. Ox comes with a vast library of functions for your convenience. These are all documented in the help and the Ox book. Whether a function is written in C, and added to the Ox system (as for the standard library), or written in Ox itself (such as the maximization functions and the Database class), does not make any difference to the user of the function.

Simple Functions

The most simple Ox function has no arguments, and returns no value. The syntax is:

```
function_name ()
{
        statements
}
```

For example:

```
#include <oxstd.h>

sometext()
{
    print("Some text\n");
}

main()
{
    sometext();
}
```

```
#include <oxstd.h>
sometext()
{
print("Some text\n");
}
main()
{
sometext();
}
```

We've created the `sometext` function, and call it from the `main` function. When the program is run, it just prints `Some text`. Note that, to call the function, the empty parentheses are required.

Function Argument

A function can take arguments. In the header of the function code, the arguments are listed, separated by commas. This example takes one argument:

```
#include <oxstd.h>

dimensions(const mX)
{
    println("the argument has ", rows(mX), " rows");
}

main()
{
    dimensions( zeros(40, 5) );
}
```

```
#include <oxstd.h>
dimensions(const mX)
{
println("the argument has ", rows(mX), " rows");
}
main()
{
dimensions( zeros(40, 5) );
}
```

The `const` which precedes each argument indicates the function is only accessing the value, not changing it. Although any change made to `mX` is only local to the function (once the function returns, both will have their old values back), it is very useful to use `const` wherever possible: the compiler can then

generate much faster code.

Exercise: Modify the dimensions function to give it two arguments, printing the number of rows in both arguments.

Returning a Value

The `return` statement returns a value from the function, and also *exits the function.* So, when the program flow reaches a `return` statement, control returns to the caller, without executing the remainder of the function. The syntax of the `return` statement is:

`return` *return value* `;`

Or, to exit from a function which does not have a return value:

```
return;
```

For example:

```
MyOls1(const mY, const mX)
{
    return (mX'mX)^-1 * (mX'mY);
}
```

```
MyOls1(const mY, const mX)
{
return (mXmX)^-1 * (mXmY);
}
```

Or, using the library function `olsc` as shown in the next example. This estimates and prints the coefficients of the linear regression model. But first, look up in Ox Help what exactly the `olsc` is supposed to do.

In the example below, the dependent variable is in the $n \times 1$ vector `mY`, and the regressors in the $n \times k$ matrix `mX`. The `&b` part is explained below. Any local variable (here: `b`) must be declared; `b` only exists while the function is active. With `return` the result is returned to the caller, and the function exited.

```
#include <oxstd.h>

MyOls(const mY, const mX)
{
    decl b;

    olsc(mY, mX, &b);
    print("in MyOls(): b=", b);
    return b;
}

main()
{
    decl b;
    // mY argument, mX argument, both just random
    b = MyOls( rann(10, 1), ranu(10, 2) );
    // b now holds the result
    print("in main(): b=", b);
}
```

```
#include <oxstd.h>
MyOls(const mY, const mX)
{
decl b;
olsc(mY, mX, &b);
print("in MyOls(): b=", b);
return b;
}
main()
{
decl b;
// mY argument, mX argument, both just random
b = MyOls( rann(10, 1), ranu(10, 2) );
// b now holds the result
print("in main(): b=", b);
}
```

In MyOls, move the line with the `return` statement to above the `print` statement, and compare the output with the old version.

Test the function using the program given underneath: the task is to use `MyOls()` for the regression. The data are observations on the weight of chickens (y) versus the amount of feed they were given (X).

```
#include <oxstd.h>

MyOls(const mY, const mX)
{
    decl b;
    olsc(mY, mX, &b);
    return b;
}

main()
{
    decl y = <0.58; 1.10; 1.20; 1.30; 1.95;
              2.55; 2.60; 2.90; 3.45; 3.50;
              3.60; 4.10; 4.35; 4.40; 4.50>;
    decl x1 = <1 : 15>'; // note transpose!
    decl mx = 1~ x1~ x1 .^ 2; // design matrix with quadratic effect
    print(y~ mx); // print all data
    // Finish the code: Use MyOls to regress y on mx & print results
}
```

```
#include <oxstd.h>
MyOls(const mY, const mX)
{
decl b;
olsc(mY, mX, &b);
return b;
}
main()
{
decl y = <0.58; 1.10; 1.20; 1.30; 1.95;
2.55; 2.60; 2.90; 3.45; 3.50;
3.60; 4.10; 4.35; 4.40; 4.50>;
decl x1 = <1 : 15>'; // note transpose!
decl mx = 1 ~ x1 ~ x1 .^2; // design matrix with quadratic effect
print(y ~mx); // print all data
// Finish the code: Use MyOls to regress y on mx & print results
}
```

Function Declaration

A function can only be called when the compiler knows about it. In the program below, the `MyOls()` function can be used inside main, because the source code is already known at that stage. If `MyOls()` were to be moved below main it cannot be used any more: the compiler has not yet encountered `MyOls()`. However, there is a way to inform about the existence of `MyOls()`, without yet giving the source code, namely by declaring the function. This amounts to copying the header only, terminated with a semicolon. To illustrate the principle:

```
#include <oxstd.h>

MyOls(const mY, const mX);// forward declaration of MyOls,
                          // so that it can be used in main

main()
{
    // now MyOls may be used here
}

MyOls(const mY, const mX)
{
    // code of MyOls
}
```

The header files (e.g. `oxstd.h`) mainly list all the function declarations together, whereas the source code resides elsewhere.

An option for small bits of code is to write the function to an `.ox` file, and just include the whole file into the source file which needs it.

Save the following code as `MyOls.ox`:

```
MyOls(const mY, const mX)
{
    decl b;
    olsc(mY, mX, &b);
    return b;
}
```

Then adjust your program as follows:

```
#include <oxstd.h>
#include "myols.ox"

main()
{
    // now MyOls may be used here
}
```

Returning Values in an Argument

Often, a function needs to return more than one value. It was pointed out before that a function cannot make a permanent change to an argument. However, this can be changed using the ampersand (`&`). The following program illustrates the principle.

```
#include <oxstd.h>

test1(x) // no const, because x will be changed
{
    x = 1;
    println("in test1: x=", x);
}

test2(const ax)
{
    // Note: indexing starts at 0 in Ox
    ax[0] = 2;
    println("in test2: x=", ax[0]);
}

main()
{
    decl x = 10;
    println("x = ", x);
    test1(x); // pass x
    println("x = ", x);
    test2(&x); // pass reference to x
    println("x = ", x);
}
```

The program prints:

```
x = 10
in test1: x=1
x = 10
in test2: x=2
x = 2
```

This is happening:

- When calling `test2`, it receives in `&x` the address of the variable `x`, not its contents. In other words, we are now working with a reference to `x`, rather than directly with `x`.
- Inside `test2`, the `ax` argument holds this address. To access the contents at that address, we use subscript 0: `ax[0]` is the contents of the address, which we can now change.
- `ax[0] = 2` does precisely that: it changes `x` itself, because `x` resides at that address.

```
#include <oxstd.h>
test1(x) // no const, because x will be changed
{
x = 1;
println("in test1: x=", x);
}
test2(const ax)
{
// Note: indexing starts at 0 in Ox
ax[0] = 2;
println("in test2: x=", ax[0]);
}
main()
{
decl x = 10;
println("x = ", x);
test1(x); // pass x
println("x = ", x);
test2(&x); // pass reference to x
println("x = ", x);
}
```

Consider the variable as a `mailbox`: a location at which a value can be stored. In the first case (`test1`), we just pass the content of the box: the function can read or change the content, but we (the function caller) still have the original version.

In the second case (`test2`), we pass the key to the mailbox (the 'address): this allows the function to put something new in the box, which will then permanently replace the content once the function returns.

Whether the variable is passed by *value* as in `test1`, or by *reference* as in `test2` is determined by the author of the function. The function user must follow the conventions adopted by the function author.

Finally, Ox makes no distinction between functions which **do** return a value, and those that **do not**. If you wish, you may ignore the return value of a function altogether. (But, of course, if a function has no return value, it should not be used in an expression.)

For example:

```
decl b0 = MyOls(my, mx)[0];
MyOls(my, mx);
print(MyOls(my, mx));
```

The first line directly indexes the returned vector. The second line calls the function, but does not use

the returned vector. The third line prints the returned vector.

Modify `MyOls` to print the following information:

```
Number of observations: xx
Coefficients:
    xx
    xx
Error variance:
    xx
```

Modify `MyOls` to compute the estimated error variance. Return this through an argument.

Recall: Error variance is $\hat{\sigma}^2 = \sum e_i^2/(n-p-1)$, where p is the number of predictor.

Additional Exercises

a) Standardizing the Columns of a Matrix

i) Look up in Ox Help how the function `standardize` works? Use the function to create sX, the standardized version the $n \times p$ matrix X. Print the (column) means and covariance of sX?

ii) Write your own function called, `mystandardize`, that turns X into sX.

b) Square Root of a Matrix

A matrix $\mathbf{A}^{1/2}$ is said to be a square root of $\mathbf{A}$ if $\mathbf{A}^{1/2}\mathbf{A}^{1/2} = \mathbf{A}$. (What does this imply about the dimension of $\mathbf{A}^{1/2}$, and, therefore, $\mathbf{A}$?) In general, a matrix can have many square roots (i.e., the square root is not unique).

i) What is the square root of a diagonal matrix $\mathbf{D}$? For real numbers, what constraints are necessary for the elements of $\mathbf{D}$?

One (iterative) method for obtaining $\mathbf{A}^{1/2}$ is the matrix extension of the Babylonian method. Let $\mathbf{X}_0$ be the identity matrix $\mathbf{I}$. In the $t+1$ iteration, the estimate of $\mathbf{A}^{1/2}$ is given by

$$\mathbf{X}_{t+1} = \frac{1}{2}\left(\mathbf{X}_t + \mathbf{A}\mathbf{X}_t^{-1}\right).$$

ii) Write a function that obtains $\mathbf{A}^{1/2}$ using the Babylonian method. In addition to the solution, the function should also indicate how many iterations were needed to arrive at the solution. Use different dimensions and types of matrix (i.e., symmetric, nonsymmetric) to test your function.

Note: Check how the function `invert` is used in Ox.

Another (exact) method is based the eigendecomposition or spectral decomposition. If the $n \times n$ matrix $\mathbf{A}$ can be decomposed as $\mathbf{A} = \mathbf{V}\mathbf{D}\mathbf{V}^{-1}$, where $\mathbf{V}$ is the matrix of eigenvectors and $\mathbf{D}$ is the diagonalized eigenvalues, the the square root of $\mathbf{A}$ is given by

$$\mathbf{A}^{1/2} = \mathbf{V}\mathbf{D}^{1/2}\mathbf{V}^{-1}.$$

iii) Show why this is true.

iv) Look up in Ox Help how the functions associated with eigendecomposition.

v) Write a function that obtains $\mathbf{A}^{1/2}$ using the eigendecomposition. Again, use different dimensions and types of matrix (i.e., symmetric, nonsymmetric) to test your function.

vi) What happens to $\mathbf{A}^{1/2}$ when $\mathbf{A}$ is symmetric? How does $\mathbf{A}^{1/2}\mathbf{A}^{1/2}$ compare with $\mathbf{A}^{1/2}(\mathbf{A}^{1/2})'$?

c) Decomposition of a Symmetric Matrix

In psychometrics, we will need to work a lot with positive-definite matrices (e.g., the covariance matrix $\mathbf{\Sigma}$. By definition, a positive-definite matrix is necessarily symmetric. One important decomposition of a positive-definite matrix $\mathbf{A}$ is the Cholesky (also, Choleski) decomposition, which is of the form

$$\mathbf{A} = \mathbf{L}\mathbf{L}',$$

where $\mathbf{L}$ is a (unique) lower triangular matrix.

i) Look up in Ox Help how the function `choleski` works.

ii) Decompose a (positive-definite) matrix using the Babylonian, eigendecomposition, and Cholesky methods, and compare the results.

d) Affine Transformations

Suppose that $\boldsymbol{X}$ is a random variable with mean $\boldsymbol{\mu}_X$ and covariance $\mathbf{\Sigma}_X$, then the affine transformation

$$\boldsymbol{Y} = \boldsymbol{a} + \mathbf{B}\boldsymbol{X}$$

will result in the random variable $\boldsymbol{Y}$, with mean $\boldsymbol{\mu}_Y = \boldsymbol{a} + \mathbf{B}\boldsymbol{\mu}_X$ and covariance $\mathbf{\Sigma}_Y = \mathbf{B}\mathbf{\Sigma}_X\mathbf{B}'$.

Let the elements of $\boldsymbol{X}_{4\times 1}$ be independently and identically distributed as χ^2_{10}. Also, let

$$\boldsymbol{a} = \begin{pmatrix} 2 \\ 4 \\ 6 \end{pmatrix} \text{ and } \mathbf{B} = \begin{pmatrix} 1 & 1 & -1 & -1 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{pmatrix}.$$

i) What are $\boldsymbol{\mu}_X$ and $\mathbf{\Sigma}_X$?

ii) Let $\boldsymbol{Y} = \boldsymbol{a} + \mathbf{B}\boldsymbol{X}$. What is the dimension of $\boldsymbol{Y}$? Derive $\boldsymbol{\mu}_Y$ and $\mathbf{\Sigma}_Y$. (Hint: You can use Ox to carry out the matrix operations.

We can also derive the answers to i) and ii) using simulation.

iii) Generate $\mathbf{X}_{N\times 4}$, a matrix of random numbers from χ^2_{10}, where N is a very large number. Find the column means and the covariance of $\mathbf{X}$, and compare to i).

Note that the formula for the affine transformation involves **column** vectors. However, $\mathbf{X}$ and $\mathbf{Y}$ are made up of **row** vectors. Consequently, you also need to transpose the formula.

iv) Transform $\mathbf{X}$ to $\mathbf{Y}$. What should be the dimension of $\mathbf{Y}$? Find the column means and the covariance of $\mathbf{Y}$, and compare to ii).

The formulas associated with the affine transformation can be applied to the sample counterparts. As in, $\bar{\boldsymbol{Y}} = \boldsymbol{a} + \mathbf{B}\bar{\boldsymbol{X}}$ and $\mathbf{S}_Y = \mathbf{B}\mathbf{S}_X\mathbf{B}'$.

v) In the following transformation, $\boldsymbol{Y} = \mathbf{S}_X^{-1/2}\left(\boldsymbol{X} - \bar{\boldsymbol{X}}\right) = \mathbf{S}_X^{-1/2}\boldsymbol{X} - \mathbf{S}_X^{-1/2}\bar{\boldsymbol{X}}$, what are $\boldsymbol{a}$ and $\mathbf{B}$?

vi) What would $\bar{\boldsymbol{Y}}$ and $\mathbf{S}_Y$ be? You can use derivation, simulation, or both to answer this question.

e) Generating Random Numbers from Multivariate Normal Distributions

It is important to know that if $\boldsymbol{X}$ is a normal random variable, the random variable $\boldsymbol{Y}$ resulting from an affine transformation is also normal. This result will allow us to generate random numbers from a multivariate normal distribution with specific $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ given random numbers from the standard normal distribution.

i) If $\boldsymbol{X}$ is from $MVN(\mathbf{0}, \mathbf{I})$, solve for $\boldsymbol{a}$ and $\mathbf{B}$ in the following equations:

$$\begin{aligned} E(\boldsymbol{Y}) = \boldsymbol{\mu} &= \boldsymbol{a} + \mathbf{B}\boldsymbol{\mu}_X \\ Var(\boldsymbol{Y}) = \boldsymbol{\Sigma} &= \mathbf{B}\boldsymbol{\Sigma}_X\mathbf{B}' \end{aligned}$$

ii) What are our options for $\mathbf{B}$?

Let `X`, an $N \times p$ matrix of random numbers from $N(0, 1)$, `M`, the desired mean vector, and `Sigma`, the desired covariance matrix, be variables in `main()`.

iii) Write a function, `mvn_gen1`, that involves **passing** `X`, plus other necessary variables, and **returns** multivariate normal random numbers with the desired mean and covariance.

iv) Write a function, `mvn_gen2`, that involves **passing** reference to `X`, plus other necessary variables, and **changes** the value of `X` to multivariate normal random numbers with the desired mean and covariance.

Functions - Part II

Functions as Arguments

A function may be passed as argument to another function, and then called from within that function. To pass a function as argument, just pass the name (without parentheses). The argument is then used as any other function, but there can be no argument checking at compile time, only at run time.

The examples in this section involves maximization of a function of several parameters. Fortunately, maximization code is provided with Ox, and we shall use that to illustrate using functions as arguments.

The library function `MaxBFGS` implements the BFGS (Broyden-Fletcher-Goldfarb-Shanno) method (other available unconstrained maximization methods are Newton's method and the Nelder-Mead simplex method). Further information on all these functions is in the Ox manual and online help.

Details of the procedures are beyond our current objectives, but there is a vast literature on non-linear optimization techniques to consult. Note that many texts on optimization focus on *minimization*, rather than *maximization*, but of course that is just a matter of reversing the sign.

Consider the linear regression model with two predictors:

$$Y_i = \beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i} + \epsilon_i.$$

Using least-squares, estimating the regression coefficients requires minimizing the sum of squared residuals. As in, the goal is to minimize the objective function

$$Q(\boldsymbol{\beta}) = \sum_{i=1}^{N} [Y_i - (\beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i})]^2.$$

In practice, X_1, X_2 and Y are observed, and β_0, β_1 and β_2 are the parameters to be estimated.

For our purposes, we can simulate data that we will analyze using the following code:

```
#include <oxstd.h>
decl X1,X2,mX,Y;
main()
{
decl beta;
decl N=10;
X1 = rann(N, 1);
X2 = rann(N, 1);
mX = 1 ~ X1 ~ X2;
beta = rann(3, 1);
Y = mX * beta + rann(N, 1);
}
```

```
#include <oxstd.h>

decl X1,X2,mX,Y;

main()
{
    decl beta;
    decl N=10;
    X1 = rann(N, 1);
    X2 = rann(N, 1);
    mX = 1 ~ X1 ~ X2;
    beta = rann(3, 1);
    Y = mX * beta + rann(N, 1);
}
```

Once we have the our data, we can obtain estimates of β_0, β_1 and β_2 by minimizing the objective function above. To achieve this goal, several functions for the nonlinear optimization *without constraints* can be considered. Examples of such functions are **MaxBFGS**, and **MaxSimplex**, which implements the Nelder-Mead simplex method.

To use the built-in functions for maximization, the objective function should be defined according to the following format:

```
func(const vP,  const adFunc,  const avScore,  const amHess);
```

obeying the following rules:

- `vP` is a $p \times 1$ matrix of parameter values at which the function is to be evaluated.
- `adFunc` must be the address of a variable on input. On output, the function value at the supplied parameters should be stored at the address.
- `avScore` holds either 0 on input, or the address of the score variable. If it was not 0 on input, the first derivatives of the function (the scores, a $p \times 1$ vector) should be stored at the address.
- We ignore the `amHess` argument.
- `func` should return 1 if it was successful, and 0 if it failed to evaluate the function at the supplied parameter values.

To minimize the objective function above, the following function can be defined:

```
funcReg(const vP, const adFunc, const avScore, const amHessian)
{
adFunc[0] = -1 * sumc ( (Y - mX * vP) .^2 );
return 1;
}
```

Note: `Y` is an $N \times 1$ matrix, `mX` is a $N \times 3$ matrix, and `vP` is a 3×1 matrix, and `sumc ( (Y - mX * vP) .^2 )` is $\sum_i^N [Y_i - (\beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i})]^2$.

Next, in the `main` function, specify the initial value of the parameters and use this function name (i.e., `funcReg`) as an argument in `MaxBFGS` or `MaxSimplex`, as in the code below. To make it run, the header `#import <maximize>` is required.

```
decl vP, func;
vP = rann(3, 1);  //initial values
//If using BFGS
MaxBFGS(funcReg, &vP, &func, 0, TRUE);
//If using Simplex
MaxSimplex(funcReg, &vP, &func, 0);
```

The function `MaxBFGS`,

```
MaxBFGS(const func, const avP, const adFunc, const amInvHess, const fNumDer);
```

has five arguments, where

`func`
in: a function computing the function value, optionally with derivatives

`avP`
in: address of $p \times 1$ matrix with starting values
out: $p \times 1$ matrix with final coefficients

`adFunc`
in: address
out: double, final function value

`amInvHess`
in: 0 or address of $p \times p$ matrix, initial (inverse negative) quasi-Hessian $\mathbf{K}$; a possible starting value is the identity matrix (which is used when 0 is used as argument)
out: if not 0 on input: final $\mathbf{K}$ (not reliable as estimate of actual Hessian)

`fNumDer`
in: 0 - `func` provides analytical first derivatives; 1 - use numerical first derivatives

Refer to Ox Help to know more about the function **MaxSimplex**.

```
#include <oxstd.h>
#import <maximize>

decl X1,X2,mX,Y;

//Optimization for minimizing the residuals of regression
//-------------------Linear Regression----------------------//
//y=b0+b1x1+b2x2+e

funcReg(const vP, const adFunc, const avScore, const amHessian)
{
    adFunc[0] = -1 * sumc ( (Y - mX * vP) .^ 2 );
    return 1; // function value - 1 indicates success
}

main()
{
    decl beta; //regression coefs (True values) <b0, b1, b2>'
    decl N=1000; //sample size
    decl vP, func, est;

    //---------Data simulation----------------//
    X1 = rann(N, 1);
    X2 = rann(N, 1);
    mX = 1 ~ X1 ~ X2; //N x 3
    beta = rann(3, 1); //3 x 1
    Y = mX * beta + rann(N, 1); //N x 1

    //True values
    print("\n","True parameters:",beta');

    //-----Linear Regression Solution---------
    print("\n","Estimates using OLS:");
    print((invert(mX'mX)*mX'Y)');

    //------------Optimization using BFGS-----------------
    vP = rann(3, 1); // starting values
    est = MaxBFGS(funcReg, &vP, &func, 0, TRUE); //MaxBFGS method
    print("\n","Estimates using BFGS:");
    print(vP');

    //---------Optimization using Simplex---------------
    vP = rann(3, 1);
    est = MaxSimplex(funcReg, &vP, &func, 0); //Simplex method
    print("\n","Estimates using Simplex:");
    print(vP');
}
```

```
#include <oxstd.h>
#import <maximize>
decl X1,X2,mX,Y;
//Optimization for minimizing the residuals of regression
//-------------------Linear Regression----------------------//
//y=b0+b1x1+b2x2+e
funcReg(const vP, const adFunc, const avScore, const amHessian)
{
adFunc[0] = -1 * sumc ( (Y - mX * vP) .^ 2 );
return 1; // function value - 1 indicates success
}
main()
{
decl beta; //regression coefs (True values) <b0, b1, b2>'
decl N=1000; //sample size
decl vP, func, est;
//---------Data simulation----------------//
X1 = rann(N, 1);
X2 = rann(N, 1);
mX = 1 ~ X1 ~ X2; //N x 3
beta = rann(3, 1); //3 x 1
Y = mX * beta + rann(N, 1); //N x 1
//True values
print("\n","True parameters:",beta');
//-----Linear Regression Solution---------
print("\n","Estimates using OLS:");
print((invert(mX'mX)*mX'Y)');
//------------Optimization using BFGS-----------------
vP = rann(3, 1); // starting values
est = MaxBFGS(funcReg, &vP, &func, 0, TRUE); //MaxBFGS method
print("\n","Estimates using BFGS:");
print(vP');
//---------Optimization using Simplex---------------
vP = rann(3, 1);
est = MaxSimplex(funcReg, &vP, &func, 0); //Simplex method
print("\n","Estimates using Simplex:");
print(vP');
}
```

In the example above, the optimization uses numerical first derivatives of the objective function. To accelerate the speed, analytical first derivatives can be provided with the objective function, and used for optimization purpose.

The partial derivatives for the objective function

$$Q(\boldsymbol{\beta}) = \sum [Y_i - (\beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i})]^2$$

are:

$$\begin{aligned}
\frac{\partial Q(\boldsymbol{\beta})}{\partial \beta_0} &= -\sum 2[Y_i - (\beta_0 + \beta_1 X_{i1} + \beta_2 X_{i2})], \\
\frac{\partial Q(\boldsymbol{\beta})}{\partial \beta_1} &= -\sum 2X_{i1}[Y_i - (\beta_0 + \beta_1 X_{i1} + \beta_2 X_{i2})], \\
\frac{\partial Q(\boldsymbol{\beta})}{\partial \beta_2} &= -\sum 2X_{i2}[Y_i - (\beta_0 + \beta_1 X_{i1} + \beta_2 X_{i2})].
\end{aligned}$$

We can modify our objective function to include the partial derivatives.

```
funcReg(const vP, const adFunc, const avScore, const amHessian)
{
adFunc[0] = -1 * sumc ( (Y - mX * vP) .^ 2 );
if (avScore) //if !0: compute score
{ //this bit is not needed for numerical derivatives
(avScore[0])[0] = 2 * sumc (Y - mX * vP);
(avScore[0])[1] = 2 * sumc (X1 .* (Y - mX * vP));
(avScore[0])[2] = 2 * sumc (X2 .* (Y - mX * vP));
}
return 1; // function value - 1 indicates success
}
```

What happened to the negative signs?

In the main function, let's make the following modifications:

```
//------------Optimization using BFGS-----------------
vP = rann(3, 1); // starting values
est = MaxBFGS(funcReg, &vP, &func, 0, TRUE); //MaxBFGS method
print("\nMaxBFGS using numerical analysis: \nFunction value is", func);
print("Estimates are:", vP');
vP = <1;2;3>; //starting values
MaxBFGS(funcReg, &vP, &func, 0, FALSE);
print("\nMaxBFGS using analytical derivative analysis:\nFunction value is", func);
print("Estimates are:", vP');
```

It should be noted that the function within which we define the objective function has a return value (i.e., 1 or 0). It can be used to determine whether the function evaluation succeeded (1) or failed (0).

Furthermore, the optimization functions (e.g., `MaxBFGS`, `MaxSimplex`) have their own return values as well. These return values can be used to determine whether the optimization has converged or not.

Include a code that shows the return values of the optimization functions, and explain the output.

In some situations, the objective function needs to be maximized subject to certain constraints (e.g., $0 \leq x \leq 1$).

The following is an example of a nonlinear optimization with *constraints* using the built-in function `MaxSQPF`. It is of the form:

```
MaxSQPF(const func, const avP, const adFunc, const amHessian,
        const fNumDer, const cfunc_gt0, const cfunc_eq0, vLo, vHi);
```

where

`func`
in: a function computing the function value, optionally with derivatives

`avP`
in: address of $p \times 1$ matrix with starting values
out: $p \times 1$ matrix with final coefficients

`adFunc`
in: address
out: double, final function value

`amHessian`
in: 0 or address
out: if not 0 on input: final Hessian approximation matrix $\mathbf{B}$

`fNumDer`
in: 0: func provides analytical first derivatives; 1: use numerical first derivatives

`vLo`
in: $p \times 1$ matrix with lower bounds, or <> if there are no lower bounds

`vHi`
in: $p \times 1$ matrix with upper bounds, or <> if there are no upper bounds

The `cfunc_gt0` argument can be zero, or a function evaluating the nonlinear constraints (which will be constrained to be positive) with the following format:

```
cfunc_gt0(const avF, const vP);
```

`avF`
in: address
out: $m \times 1$ matrix with constraints at `vP`

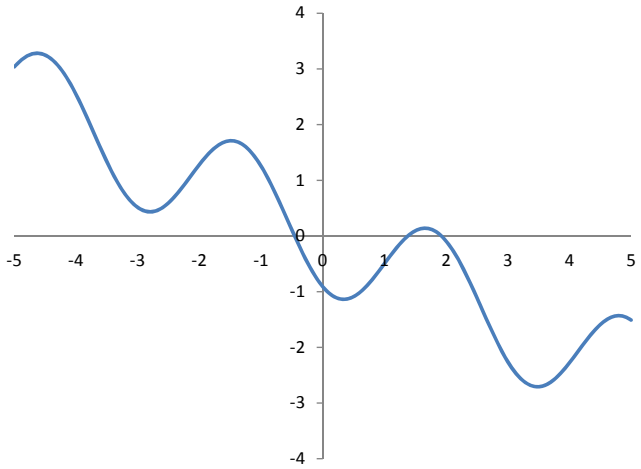
`vP`
in: $p \times 1$ matrix with coefficients

returns
1: successful; 0: constraint evaluation failed

Assume we want to find the maximum of the objective function

$$Q(x) = \sin(2x - 2) - .5x,$$

with the constrain that $x \in [1, +\infty)$. Below is the graph of $Q(x)$.



From the graph, we can see that the global maximum of $Q(x)$ "occurs" at $-\infty$ (i.e., it does not exist). However, a few local maxima exist. We want to find global maximum for $x \geq 1$. Let's see how we can solve this problem using `MaxSQPF`.

First, define the objective function using the following code:

```
func(const vP, const adFunc, const avScore, const amHessian)
{
adFunc[0] =sin(2*vP[0]-2)-.5*vP[0];
return 1;
}
```

Then set the constraints $x \geq 1$ using the following code:

```
cfunc_gt0(const avF, const vP)
{
avF[0] = vP - 1;
return 1;
}
```

Finally, the optimization can be put into action by calling it in `main`.

```
main(){
decl fvalue; //function value
decl vP=2.5; //initial value for x
decl vlow = <>; //lower bound of x, set to null
decl vhigh = <>; //upper bound of x, set to null
MaxSQPF(func, &vP, &fvalue, 0, TRUE, cfunc_gt0, 0, vlow, vhigh);
println("\nMaximum function value is ",fvalue,"\nand it can be found at x = ", vP);
}
```

To make it work, the following headers are required:

```
#include <oxstd.h>
#import <maxsqp>
```

This is how the Ox final program should look like.

```
#include <oxstd.h>
#import <maxsqp>

func(const vP, const adFunc, const avScore, const amHessian)
{
    adFunc[0] =sin(2*vP[0]-2)-.5*vP[0];
    return 1;
}

cfunc_gt0(const avF, const vP)
{
    avF[0] = vP - 1;
    return 1;
}

main(){
    decl fvalue; //function value
    decl vP=2.5; //initial value for x
    decl vlow = <>; //lower bound of x, set to null
    decl vhigh = <>; //upper bound of x, set to null

    MaxSQPF(func, &vP, &fvalue, 0, TRUE, cfunc_gt0, 0, vlow, vhigh);

    println("\nMaximum function value is ",fvalue);
    println("and it can be found at x = ", vP);
}
```

Consider the following questions:

(1) Can we do this optimization by setting lower bound or upper bound of x instead of the constraint function?

(2) How do we set the constraint $x^2 \leq 4$?

(3) How do we use MaxSQPF for unconstrained optimization?

The following is an example to estimate the coefficients in logistic regression using optimization techniques.

Let the regression model be

$$\text{logit}P(Y_i = 1|\boldsymbol{\beta}) = \beta_0 + \beta_1 X_{1i},$$

or

$$P(Y_i = 1|\boldsymbol{\beta}) = \frac{\exp(\beta_0 + \beta_1 X_{1i})}{1 + \exp(\beta_0 + \beta_1 X_{1i})}.$$

To simplify the notation, we can denote $P(Y_i = 1|\boldsymbol{\beta})$ as P_i. Then, the objective function (in this case, the log-likelihood) can be written as

$$Q(\boldsymbol{\beta}) = \sum_{i=1}^{N} Y_i \log P_i + (1 - Y_i) \log(1 - P_i).$$

We can optimize $Q(\boldsymbol{\beta})$ using numerical first derivative or analytical first derivative. To use the latter, the first derivatives should be provided with the objective function. It can be easily shown that the first derivatives with respect to β_0, β_1 and β_2 can be shown as

$$\begin{aligned} \frac{\partial Q(\boldsymbol{\beta})}{\partial \beta_0} &= \sum (Y_i - P_i), \\ \frac{\partial Q(\boldsymbol{\beta})}{\partial \beta_1} &= \sum X_{1i} \times (Y_i - P_i), \end{aligned}$$

Based on the objective function and the first derivatives above, we can define the objective function as follows:

```
LogistReg(const vP, const adFunc, const avScore, const amHessian)
{
    decl ex=exp(mX * vP);
    decl P=ex./(1 + ex);
    adFunc[0] = sumc(Y.*log(P)+(1-Y).*log(1-P));
    if (avScore)
        {
        (avScore[0])[0] = sumc (Y - P);
        (avScore[0])[1] = sumc (X1 .* (Y - P));
        }
    return 1;
}
```

```
LogistReg(const vP, const adFunc, const avScore, const amHessian)
{
decl ex=exp(mX * vP);
decl P=ex./(1 + ex);
adFunc[0] = sumc(Y.*log(P)+(1-Y).*log(1-P));
if (avScore)
{
(avScore[0])[0] = sumc (Y - P);
(avScore[0])[1] = sumc (X1 .* (Y - P));
}
return 1;
}
```

The following code in the `main` can be used for data generation.

```
decl beta;
decl N=500;
decl vP, func, est;
//---------Begin Data simulation----------------
X1 = rann(N, 1);
mX = 1 ~ X1;
beta = <-.5; 2>;
decl p = exp(mX * beta) ./ ( 1 + exp(mX * beta) );
Y = p .> ranu(N, 1);
//---------Data simulation End----------------
```

The following code in the `main` can be used to obtain estimates of β with analytical and numerical derivatives.

```
//---MaxBFGS using analytical derivatives
vP=<1, 2>';
est=MaxBFGS(LogistReg, &vP, &func, 0, FALSE);
println("\nMaxBFGS with analytical derivatives");
println(" Convergence: ",MaxConvergenceMsg(est));
println(" Function value = ", func," Parameter estimates = ", vP');
//---MaxBFGS using numerical derivatives
vP=<1, 2>';
est=MaxBFGS(LogistReg, &vP, &func, 0, TRUE);
println("\nMaxBFGS with numerical derivatives");
println(" Convergence: ",MaxConvergenceMsg(est));
println(" Function value = ", func," Parameter estimates = ", vP');
```

The following code in the `main` involves two ways of maximization with constraints - through a constraint functions or through bounds.

```
//---This is one way to maximize with a constraint
// - using constraint function
vP=<1, 2>';
decl vlow = <>;
decl vhigh = <>;
est=MaxSQPF(LogistReg, &vP, &func, 0, TRUE, cfunc_gt0, 0, vlow, vhigh);
println("\nMaximization with a constraint function");
println(" Convergence: ",MaxConvergenceMsg(est));
println(" Function value =\n     ", func,"\n Parameter estimates = ", vP');
//---This is another way to maximize with a constraint
//- using lower and upper bounds
vP=<1, 2>';
vlow = <M_INF_NEG, 0>';
vhigh = <>;
est=MaxSQPF(LogistReg, &vP, &func, 0, TRUE, 0, 0, vlow, vhigh);
println("\nMaximization with bounds");
println(" Convergence: ",MaxConvergenceMsg(est));
println(" Function value =\n     ", func,"\n Parameter estimates = ", vP');
```

Note that for the maximization with constraint function to work, we need to define a constraint function outside `main`. The function below constrain the second parameter to be positive.

```
cfunc_gt0(const avF, const vP)
{
avF[0] = vP[1];
return 1;
}
```

Lastly, make sure to include and import the necessary headers, and declare the appropriate global variables:

```
#include <oxstd.h>
#include <oxfloat.h>
#import <maximize>
#import <maxsqp>
decl X1, mX, Y;
```

1) Create a program from these codes, and run it.

2) Compare the results from the different maximization algorithms.

3) Compare the results from what you would get in SPSS.

Model Summary

Step	-2 Log likelihood	Cox & Snell R Square	Nagelkerke R Square
1	483.857[a]	.312	.422

a. Estimation terminated at iteration number 5 because parameter estimates changed by less than .001.

Variables in the Equation

		B	S.E.	Wald	df	Sig.	Exp(B)
Step 1[a]	X1	1.761	.170	107.465	1	.000	5.817
	Constant	-.611	.116	27.967	1	.000	.543

a. Variable(s) entered on step 1: X1.

In addition to the estimates, we are typically also interested in the standard errors. We can calculate the standard errors using the function `Num2Derivative`. This function is used to numerically calculate the second derivatives for objective function at user-specified points. In our case, the standard errors of the coefficients using numerical second derivatives of the log-likelihood at the converged parameter values can be estimated. Because the variance-covariance matrix is minus the inverse of the second derivatives:

$$\mathbf{Var}(\hat{\boldsymbol{\beta}}) \approx -\mathbf{H}^{-1}(\hat{\boldsymbol{\beta}}) \quad \text{where} \quad \mathbf{H} = \left\{ \frac{\partial^2 l}{\partial \beta_i \partial \beta_{i'}} \right\}.$$

The standard errors are the square root of the diagonal elements of the variance-covariance matrix.

We can now add the following code in the `main` function to calculate the standard errors of the estimates. Note that the standard errors will only be calculated when the return value of the optimization functions are `MAX_CONV` or `MAX_WEAK_CONV`, which means the estimates have converged strongly or weakly.

```
// if estimation converges, compute the standard errors
decl mhess;
if (est == MAX_CONV || est == MAX_WEAK_CONV)
{
if (Num2Derivative(LogistReg, vP, &mhess))
{
mhess = -invert(mhess);
print("\n Standard errors:", sqrt(diagonal(mhess)'));
}
}
```

Add this code to the `main` function, and compare the results to the standard errors provided by SPSS.

Additional Exercises

1. Refer to Ox Help to check how the functions `MaxSimplex` and `MaxNewton` work. Modify the program to obtain the estimates and standard errors of the logistic regression coefficients using these two functions.

2. The following are responses of five examinees on 10 items. These responses were generated based on a 2PL model.

```
decl X=<
1,1,0,0,0,1,0,0,0,0;
1,1,0,1,0,1,0,1,0,0;
1,0,1,1,0,1,1,0,0,0;
0,0,1,0,0,1,1,1,0,0;
1,1,1,0,0,1,1,0,0,0>;
```

The true item parameters a and b are

```
decl b=<-2; -1; 0; 1; 2; -2; -1; 0; 1; 2>;
decl a=<0.81; 1.44; 0.70; 0.57;
0.61; 0.67; 0.81; 0.72; 0.71; 0.95>;
```

Recall that the 2PL is

$$P_{ij} = P(X = 1|\theta_i, a_j, b_j) = \frac{\exp[1.7a_j(\theta_i - b_j)]}{1 + \exp[1.7a_j(\theta_i - b_j)]},$$

and the corresponding likelihood is

$$L = \prod_{i=1}^{N} \prod_{j=1}^{J} P_{ij}^{X_{ij}} (1 - P_{ij})^{(1-X_{ij})}$$

where N is the number of examinees, and J is the number of items. Accordingly, the log-likelihood is

$$l = \log L = \sum_{i=1}^{N} \sum_{j}^{J=1} [X_{ij} \log P_{ij} + (1 - X_{ij}) \times \log(1 - P_{ij})]$$

a) Based on these equations, estimate the abilities using optimization methods with numerical derivatives. You may want to constrain the person abilities to between -3 and 3.

b) Estimate person abilities using appropriate optimization method with the analytical derivative with respect to θ given below:

$$\frac{\partial l}{\partial \theta_i} = \sum_{j=1}^{J} [1.7a_j(X_{ij} - P_{ij})].$$

c) Estimate the standard errors of the estimates.

3. Two attributes are required for a item. Therefore, examinees can be classified into one of the four groups, 00, 10, 01 and 11, depending on the attribute mastery patterns. Assume we have the following information about the expected number of examinees N_g, and the expected number of examinees answering correctly R_g in group g.

$\boldsymbol{\alpha}_g$	N_g	R_g
00	50.23	10.08
10	69.32	15.47
01	31.01	4.89
11	152.44	124.66

The likelihood function is

$$L = \prod_{g=1}^{4} P(X = 1|\boldsymbol{\alpha}_g)^{R_g}(1 - P(X = 1|\boldsymbol{\alpha}_g))^{N_g - R_g},$$

where $P(X = 1|\boldsymbol{\alpha}_g)$ is the probability of answering correctly the item given the g^{th} attribute pattern α_g.

If we want to fit a DINA model, $P(\boldsymbol{\alpha}_1) = P(\boldsymbol{\alpha}_2) = P(\boldsymbol{\alpha}_3) = \varrho$ and $P(\boldsymbol{\alpha}_4) = 1 - \varsigma$, where ϱ and ς are the guessing and slip parameters, respectively.

a) Show that likelihood in terms of ϱ and ς is

$$L = \varrho^{\sum_{g=1}^{3} R_g} \times (1 - \varrho)^{\sum_{g=1}^{3}(N_g - R_g)} \times (1 - \varsigma)^{R_4} \times (\varsigma)^{N_4 - R_4}$$

b) Derive the log-likelihood function, and estimate ϱ and ς using an optimization method without a constraint.

c) This time, include the following in your optimization function as constraints: $s, g \geq 0$ and $1 - s > g$.

d) The following are the first derivatives of the log-likelihood function with respect to ϱ and ς:

$$\begin{aligned} \frac{\partial l}{\partial \varrho} &= \frac{\sum_{g=1}^{3} R_g}{\varrho} - \frac{\sum_{g=1}^{3}(N_g - R_g)}{1 - \varrho} \\ \frac{\partial l}{\partial \varsigma} &= \frac{N_4 - R_4}{\varsigma} - \frac{R_4}{1 - \varsigma}. \end{aligned}$$

Estimate ϱ and ς with the the analytical derivatives in your optimization.

e) The maximum likelihood estimates of ϱ and ς can be written in closed form, as in:

$$\begin{aligned} \hat{\varrho} &= \frac{\sum_{g=1}^{3} R_g}{\sum_{g=1}^{3} N_g} \\ \hat{\varsigma} &= 1 - \frac{R_4}{N_4} \end{aligned}$$

Compute the MLEs using these formulas, and compare the estimates with the optimization results.

f) Find the standard errors of the estimates.

g) Lastly, find the DINO model parameter estimates and standard errors for the same data.

Data Generation and Analysis

Data Generation

In psychometrics, (Monte Carlo) simulation study is one of the primary tools, if not the primary tool, used by researchers to investigate the viability of newly developed models and procedures. Conducting simulation studies requires generating lots and lots of data. The reliability of the findings depends on how well the data have been generated from the underlying models. In addition to standard distributions, simulation studies in psychometric research also involve sampling from nonstandard distributions. We will discuss procedures for sampling from distributions commonly encountered in psychometric literature. We will also discuss general techniques that can be used for sampling from standard and nonstandard distributions.

Generating Continuous Latent Variable

Univariate

Typically, when the latent variable is univariate, it is assumed to have a standard normal distribution. Most programming languages have a function that can generate random numbers from ordinary distributions such as $N(0,1)$. If not, techniques, such as the Box-Muller transformation, can be used. As you already know, sampling from the more general $N(\mu, \sigma^2)$ involves a simple linear transformation.

Occasionally, we may need to work with nonnormal distributions to manipulate the kurtosis or skewness of the sample. For example, we may want to work with a distribution that has heavier tails. Instead of the normal distribution, we can use the t-distribution. In Ox, this can be done using `rant(r, c, df)`. Alternatively, we can also use `rann(r,c)./sqrt(ranchi(r,c,df)/df)`.

For skewed distribution, the skew-normal distribution can be used. Its density function is given by

$$f(x) = \frac{2}{\omega}\phi\left(\frac{x-\xi}{\omega}\right)\Phi\left[\alpha\left(\frac{x-\xi}{\omega}\right)\right],$$

where ξ, ω, and α are the location, scale, and shape parameters, respectively,

$$\phi(x) = \frac{1}{\sqrt{2\pi}}\exp\left(-\frac{x^2}{2}\right),$$

and

$$\Phi(x) = \int_{-\infty}^{x}\phi(t)dt.$$

Its mean and variance are:

$$\mu = \xi + \omega\delta\sqrt{\frac{2}{\pi}}$$

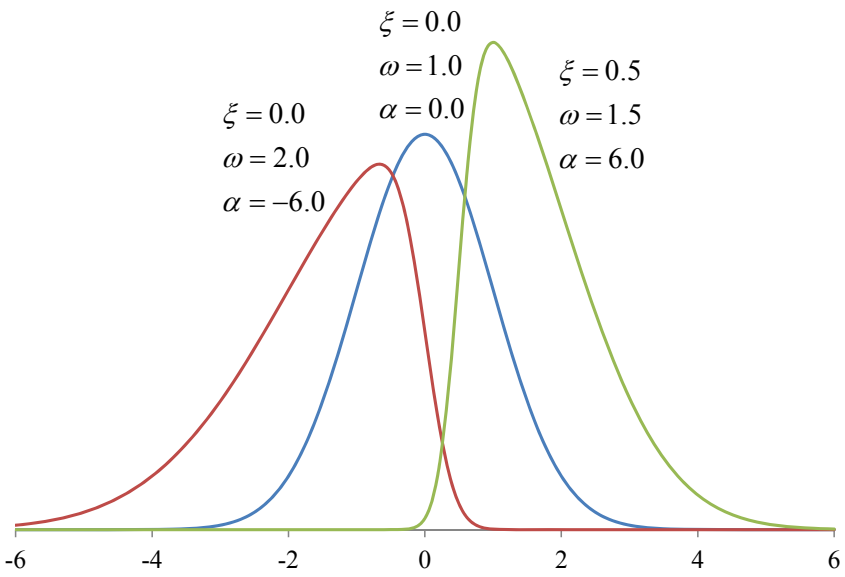
and

$$\sigma^2 = \omega^2\left(1 - \frac{2\delta^2}{\pi}\right),$$

where

$$\delta = \frac{\alpha}{\sqrt{1+\alpha^2}}.$$

Below is a graph showing three skew-normal distributions for different value of ξ, ω, and α. When $\xi = 0$, $\omega = 1$, and $\alpha = 0$, we get the standard normal distribution.



A Detour: Rejection Sampling (From *Bayesian Data Analysis*, pages 284-285)

Of the many procedures for sampling from a target distribution, *rejection sampling* or *acceptance-rejection method* is possibly the simplest and more general method of doing so. It can also be used in conjunction with other more complex procedures (i.e., Markov chain Monte Carlo).

Let $f(x)$ be the target density. To perform rejection sampling, we need a positive function $g(x)$ defined for all x for which $f(x) > 0$ that has the following properties:

- We need to be able to sample from the probability density $h(x)$ proportional to $g(x)$, as in, $g(x) = M \times h(x)$, for $M > 1$.
- The importance ratio $f(x)/g(x)$ must have a known bound. That is, there must be a known constant M for which $f(x)/h(x) \leq M$ for all x.

To use rejection sampling to obtain a single draw from $f(x)$, the procedure is carried out in two steps:

1. Sample X at random from $h(x)$, the probability density proportional to $g(x)$.
2. Accept the sample with probability $f(x)/g(x)$. If sample is rejected, repeat the previous step.

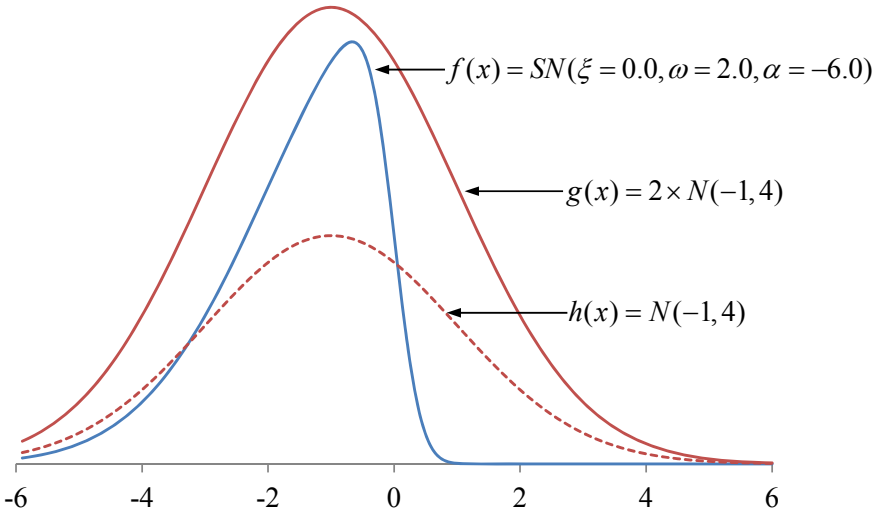
A good approximate density $h(x)$ for rejection sampling should be roughly proportional to $f(x)$. This can lead to most of the draws being accepted. If $h(x)$ is quite different from $f(x)$, we may need to chose a large M to ensure that $f(x) \leq g(x)$. This can lead to an inefficient sampling process (i.e., many of the draws will be rejected).

End of Detour

Suppose we are interested in sampling from a skew-normal distribution with parameters $\xi = 0$, $\omega = 2$, and $\alpha = -6$. Because we do not know a *direct* sampling method to obtain observations from this

distribution, we can use rejection sampling.

Below is a graph showing the density curve of $f(x) = SN(\xi = 0, \omega = 2, \alpha = -6)$. We will sample from a known distribution, $h(x) = N(-1, 4)$. It can be verified that $f(x)/h(x) \leq 2$. So, we set $g(x) = 2 \times h(x)$.



Below is a function for drawing from a skew-normal distribution. Instead of a single draw, the function samples `n` draws at a time from $h(x)$ to be more efficient. The M for this function is appropriate for the current $f(x)$ and $h(x)$ parameters. This means than a different M might be needed if $f(x)$ and $h(x)$ are altered.

```
decl N=10000,n=100;
decl mu=-1,sigma=2;
decl xi=0,omega=2,alpha=-6;
decl M=2;
drawSN(decl n)
{
decl h=rann(n,1)*sigma+mu;
decl g=densn((h-mu)/sigma);
decl f=2/omega*densn((h-xi)/omega).*probn(alpha*(h-xi)/omega);
decl prob=f./(M*g);
return selectifr(h,prob.>ranu(n,1));
}
```

We will need the code below in the *main*. The `while` loop keeps track of the sampled draws, number of attempts, and number of accepted draws. We also compare μ and σ^2 with $\bar{x}$ and s^2.

```
decl X=<>;
```

```
decl count0=0,count1=0; //count0->total attempts;count1->number of accepts
decl time=timer();
while(count1<N)
{
decl tmp=drawSN(n);
decl acc=rows(tmp);
count0=count0+n;
count1=count1+acc;
X=X|tmp;
}//end while
print("\n  Elapsed Time:",timespan(time),"\n");
print("\n  # of Attempts vs. # of Accepts:", count0~count1);
X=X[0:N-1];
decl delta=alpha/sqrt(1+alpha^2);
print("\n Theoretical Mean and Variance:");
print(xi+omega*delta*sqrt(2/M_PI )~omega^2*(1-2*delta^2/M_PI ));
print("\n Sample Mean and Variance:", meanc(X)~varc(X));
```

1. Use different values for `n`, and examine how it affects the a) time and b) number of attempts needed to reach the sample size N.

2. Examine the impact of changing $M = 2$ to $M = 10$ on the number of attempts.

Another way of generating skewed (or more generally, nonnormal) distribution is to use the Fleishman's cubic transformation method. The random variable Y can be derived from $X \sim N(0, 1)$ using the following transformation:

$$Y = a + bX + cX^2 + dX^3.$$

The coefficients a, b, c, and d can be chosen so that the distribution has specific mean (μ_Y), variance (σ_Y^2), skewness (γ_{1Y}), and kurtosis (γ_{2Y}). For example, for $\mu_Y = 0$ and $\sigma_Y^2 = 1$, $\gamma_{1Y} = 0$, and $\gamma_{2Y} = 0$ (i.e., standard normal distribution), we set $a = c = d = 0$ and $b = 1$; for $\mu_Y = 0$ and $\sigma_Y^2 = 1$, $\gamma_{1Y} = 0.5$, and $\gamma_{2Y} = 0$ (i.e., a skewed distribution), we set $a = -c \approx -0.0926$, $b \approx 1.0399$, and $d \approx -0.0166$.

Although we know the moments of Y, we do not know its exact distribution.

It should be noted that for latent variable models to be identified, we typically constrain the latent variable to have a mean of $\mu = 0$ and variance $\sigma^2 = 1$.

Multivariate

Many of the models in the psychometric literature call for multivariate latent variables. More often than not, they are also multivariate normal.

As discussed in Lecture 2, we can generate $Y_{p\times 1} \sim N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ by sampling $X_{p\times 1}$ from MVN$(\mathbf{0}, \mathbf{I})$, and performing the affine transformation

$$\boldsymbol{Y} = \boldsymbol{\mu} + \boldsymbol{\Sigma}^{1/2}\boldsymbol{X}.$$

Again, for identifiability purposes, we typically set $\boldsymbol{\mu} = \mathbf{0}$ and $\boldsymbol{\Sigma} = \mathbf{R}$.

Incorporating Covariates

Increasingly, covariates are being included as part of latent variable models. In addition to the response data, covariates can be a source of ancillary information to improve estimation.

Let $\boldsymbol{\theta}_{D\times 1}$ be the latent variable, and $\boldsymbol{Y}_{p\times 1}$ the vector of covariates. The objective is to generate $\boldsymbol{\theta}$ that has the following structures:

- (1) the marginal covariance of $\boldsymbol{\theta}$ is a correlation matrix, and
- (2) the proportion of variance accounted for by the covariates in regressing $\boldsymbol{\theta}$ on $\boldsymbol{Y}$ is ψ.

Let

$$\begin{bmatrix} \boldsymbol{\theta} \\ \boldsymbol{Y} \end{bmatrix} \sim \text{MVN}\left(\begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \begin{bmatrix} \boldsymbol{\Sigma}_{\theta\theta} & \boldsymbol{\Sigma}_{\theta Y} \\ \boldsymbol{\Sigma}_{Y\theta} & \boldsymbol{\Sigma}_{YY} \end{bmatrix} \right).$$

By design, we can let $\boldsymbol{\Sigma}_{YY}$ be equal to $\mathbf{I}$. Using the properties of conditional distributions of a MVN distribution (Result 4.6 of Johnson, and Wichern, 2007; Theorem 3.2.4 of Mardia, Kent, & Bibby, 1979), the conditional distribution of $\boldsymbol{\theta}$ given $\boldsymbol{Y} = \boldsymbol{y}$ is

$$\boldsymbol{\theta}|\boldsymbol{Y} = \boldsymbol{y} \sim \text{MVN}\left(\boldsymbol{\Sigma}_{\theta Y}\boldsymbol{y},\ \boldsymbol{\Sigma}_{\theta\theta} - \boldsymbol{\Sigma}_{\theta Y}\boldsymbol{\Sigma}_{Y\theta}\right).$$

The two structures given above imply the following.

$$\begin{aligned} \{\boldsymbol{\Sigma}_{\theta\theta}\}_{dd} &= 1; \\ \{\boldsymbol{\Sigma}_{\theta Y}\boldsymbol{\Sigma}_{Y\theta}\}_{dd} \,/\, \{\boldsymbol{\Sigma}_{\theta\theta}\}_{dd} = \{\boldsymbol{\Sigma}_{\theta Y}\boldsymbol{\Sigma}_{Y\theta}\}_{dd} &= \psi; \end{aligned}$$

where $\{\boldsymbol{\Sigma}\}_{rc}$ is the element in row r and column c, of the matrix $\boldsymbol{\Sigma}$.

It can be assumed without loss of generality that the proportions of variance accounted for by the covariates are the same for all dimensions, and that each covariate has the same predictive power. It follows that

$$\begin{aligned} \{\boldsymbol{\Sigma}_{\theta\theta} - \boldsymbol{\Sigma}_{\theta Y}\boldsymbol{\Sigma}_{Y\theta}\}_{dd} &= 1 - \psi \\ \{\boldsymbol{\Sigma}_{\theta\theta} - \boldsymbol{\Sigma}_{\theta Y}\boldsymbol{\Sigma}_{Y\theta}\}_{dd'} &= \rho(1 - \psi) \\ \{\boldsymbol{\Sigma}_{\theta\theta}\}_{dd'} &= \rho + \psi - \rho\psi \\ \{\boldsymbol{\Sigma}_{\theta Y}\boldsymbol{\Sigma}_{Y\theta}\}_{dd} = \{\boldsymbol{\Sigma}_{\theta Y}\boldsymbol{\Sigma}_{Y\theta}\}_{dd'} &= \psi \\ \{\boldsymbol{\Sigma}_{\theta Y}\} &= \sqrt{\psi/p} \end{aligned}$$

Following are the steps in generating N $\boldsymbol{\theta}$ observations with a correlation of ρ between the dimensions, and where the proportion variance accounted for by the covariates is equal to ψ.

1. Generate a $\text{Y}_{N\times p}$ matrix of covariates, as in, N observations from $\text{MVN}_p(\mathbf{0}, \mathbf{I})$.
2. Compute $\mathbf{YB}$, the matrix *conditional* mean vector for the N examinees, where $\mathbf{B}$ is $p \times D$ and $\{\text{B}\} = \sqrt{\psi/p}$.
3. Generate observations from $\text{MVN}_D(\{\text{YB}\}_i', \boldsymbol{\Sigma}_{\theta|Y})$, $i = 1, 2, \ldots, N$, where $\{\mathbf{YB}\}_i$ is the i^{th} row of $\mathbf{YB}$, and $\boldsymbol{\Sigma}_{\theta|Y} = (1 - \psi)[(1 - \rho)\boldsymbol{I}_{D\times D} + \rho]$.

Using this algorithm, the *marginal* distribution of $\boldsymbol{\theta}$ is a multivariate normal distribution with the parameters

$$\begin{aligned} \boldsymbol{\mu}_\theta &= \mathbf{0} \\ \boldsymbol{\Sigma}_{\theta\theta} &= (1-\rho-\psi+\rho\psi)\boldsymbol{I}_{D\times D} + (\rho+\psi-\rho\psi) \end{aligned}$$

Approximate normal distributions can be obtained when using discrete covariates by adding an additional step to the algorithm. For example for a dichotomous variable, the additional step, Step 1.5, is to code the covariate as -1 if the observation is below the median, or 1 otherwise. This modification preserves the mean and covariance structure of the covariates. The rest of the steps still apply.

Higher-Order Models

In some applications, the structure of the covariance $\boldsymbol{\Sigma}$, typically, expressed as $\mathbf{R}$, of the latent variable $\boldsymbol{\theta}_{D\times 1}$ can be accounted for by positing a higher-order latent variable $\boldsymbol{\omega}_{p\times 1}$. For our purposes, it would suffice to treat $\boldsymbol{\omega}$ as univariate (i.e., $\boldsymbol{\omega} = \omega$).

Specifically, we can express the the d^{th} element of $\boldsymbol{\theta}$ as a linear function of ω, as in

$$\theta_{di} = \lambda_d \times \omega_i + \epsilon_{di},$$

where λ_d is the latent coefficient in regressing the θ_d on ω, ϵ_{di} is the error term that is independently and identically distributed as $N(0, 1-\lambda_d^2)$, and $|\lambda_d| \leq 1$.

This formulation implies the following properties:

$$\begin{aligned} \rho(\theta_d, \theta_{d'}|\omega) &= 0 \\ \rho(\theta_d, \omega) &= \lambda_d \\ \rho(\theta_d, \theta_{d'}) &= \lambda_d \times \lambda_{d'} \end{aligned}$$

Show that

(i) $\theta_{di}|\omega \sim N(\lambda_d \times \omega_i, 1-\lambda_d^2)$, and

(ii) if $\omega \sim N(0,1)$, the marginal distribution of θ_d is also $N(0,1)$.

To generate $\boldsymbol{\theta} \sim \text{MVN}(0, \mathbf{R})$ using the higher-order model, where $\mathbf{R}$ is compound symmetric (i.e., $\{\mathbf{R}\}_{dd'} = \rho$):

1. Generate ω_i from $N(0,1)$.
2. Then, given ω_i, generate θ_{di} from $N(\lambda_d \times \omega_i, 1-\lambda_d^2)$, where $\lambda_d = \sqrt{\rho}$.

It should be noted that when $D = 2$, the estimates of the regression parameters will not be unique because different sets of λ_1 and λ_2 resulting in the same ρ can be found. When $D = 3$, the higher-order model perfectly fits the structure of $\boldsymbol{\Sigma}$. Why?. When $D \geq 4$, the true correlational structure of $\boldsymbol{\theta}$ may be more complex than what a linear model can fit. Again, why?

Generating Discrete Latent Variable

In this section, we will focus on a specific type of discrete latent variable - (multivariate) binary latent variables $\boldsymbol{\alpha}_{K\times 1}$ (i.e., attribute vectors). Of course, $\boldsymbol{\alpha}$ need not by binary/dichotomous, as in, α_k need not assume the value 0 or 1 only. However, only a few works involving polytomous attributes are currently available so it may not be worthwhile to delve into this type of attribute in the time being. In addition, extending procedures for dichotomous to polytomous attributes should be relatively straightforward. Below is a discussion of how to generate attributes with different structures.

Saturated Structure

A saturated attribute structure is the most general formulation of $\boldsymbol{\alpha}$ because no *prior* constraints are imposed on the attributes. The prevalence of an attribute pattern, $P(\boldsymbol{\alpha}_l)$, for $l = 1, \ldots, 2^K$, is unrestricted, and is only subject to the constraint that $\sum_{\forall l} P(\boldsymbol{\alpha}_l) = 1$. It should be noted, however, that attributes based on a saturated formulation are not necessarily unstructured - they may still follow a particular structure, albeit implicitly. For example, although $P(\boldsymbol{\alpha}_l)$ may be assigned without specific restrictions, the resulting attributes may still end up following a particular structure, say, a higher-order structure.

To sample $\boldsymbol{\alpha}$ from a saturated model or structure, we the $P(\boldsymbol{\alpha}_l)$. Given the probabilities, we can sample $\boldsymbol{\alpha}$ using a multinomial distribution. In Ox, we can use `ranmultinomial(const n, const vp)`, where `n` is the number of trials (i.e., N), and `vp` is a vector of probabilities of length L, with probabilities summing to one. This function will return $\boldsymbol{n}_{1\times L} = (n_1, \ldots, n_L)$, a vector of nonnegative integers, where $\sum_{\forall l} n_l = N$.

Given $\boldsymbol{n}$, we can create $\mathbf{A}_{N\times K}$, which contains n_1 rows of $\boldsymbol{\alpha_1}$, n_2 rows of $\boldsymbol{\alpha_2}$, ..., and n_L rows of $\boldsymbol{\alpha_L}$.

Below are codes we can used to sample sample $\boldsymbol{\alpha}$ from a saturated model.

The first code starts with the declaration of a few global variables. Note the difference between `alpha` and `alpha0`.

The function `pattern` will create `alpha`, a $2^K \times K$ matrix containing all the possible attribute patterns. The attributes patterns are ordered as follows:

1. First, increasing with respect to $\sum_{\forall k} \alpha_{lk}$, the number of mastered attributes.
2. Then, for a fixed number of mastered attributes, decreasing with respect to $\sum_{\forall k} \alpha_{lk} 10^{K-k+1}$, the "numerical value" of the attribute pattern, which can be interpreted as "attributes with smaller k are mastered earlier."

In this particular ordering, it will be easy to recreate `alpha` manually. More importantly, this particular ordering makes it easier to locate specific attribute patterns.

```
decl N=20, K=3; //N->sample size; K->number of attributes
decl alpha; //alpha-> 2^KxK possible attribute patterns
decl alpha0; //alpha0-> NxK sampled attribute patterns
```

```
pattern(){ //create a matrix of all the possible attribute patterns
alpha=<>;
for(decl k=1;k<=K;k++){
decl tmp=<>;
decl base=zeros(2^(K-k),1)|ones(2^(K-k),1);
for(decl l=0;l<2^(k-1);l++){
tmp=tmp|base;}
alpha=alpha~tmp;}
alpha=alpha~sumr(alpha);
decl tmp=zeros(2^K,1);
for(decl k=0;k<K;k++){
tmp=tmp-10^(K-k-1)*alpha[][k];
}
alpha=alpha~tmp;
alpha=sortbyc(alpha,K~(K+1));
alpha=alpha[][0:K-1];
}
```

The code below should appear in `main`. It starts by calling the function `pattern`. In lieu of specifying the 2^K probabilities associated with the 2^K attribute patterns, we have a general code for making all the attribute patterns equally likely (i.e., the attribute patterns are uniformly distributed).

Note that, originally, `ex` is used to hold $(P(\boldsymbol{\alpha}_1), \ldots, P(\boldsymbol{\alpha}_{2^K}))'$; eventually, the same variable is used to hold $(n_1, \ldots, n_{2^K})$. In doing so, we have one less variable to declare and keep tract of.

After declaring `alpha0` as a null matrix, we use a loop to convert $\boldsymbol{n}$ to $\mathbf{A}$.

```
ranseed(130930);
pattern();
decl ex=ones(2^K,1)./(2^K); //attribute patterns are equiprobable
ex=ranmultinomial(N,ex);
alpha0=<>;
for(decl l=0;l<2^K;l++){
alpha0=alpha0|(ones(ex[l],1)*alpha[l][]);
}
print(ex'~alpha);
print(alpha0);
```

Assemble the codes, and examine the output. Which attribute pattern has the highest (lowest) frequency?

Higher-Order Structure

The attribute patterns are said to have a higher-order structure if they follow the model presented by de la Torre and Douglas (2004) where an examinees overall (or higher-order) ability θ determines the probability that the examinee masters each of the attributes. (In general, the higher-order ability can be multivariate. However, estimating $\boldsymbol{\theta}$ accurately may be challenging.)

Specifically, θ_i and α_{ik} are related as follows:

$$P(\alpha_{ik} = 1|\theta_i) = \frac{\exp[\lambda_{1k}(\theta_i - \lambda_{0k})]}{1 + \exp[\lambda_{1k}(\theta_i - \lambda_{0k})]}.$$

where λ_{0k} and λ_{1k} are the difficulty and discrimination parameters of attribute k. It is not unusual to use a common discrimination parameter across the attributes (i.e., $\lambda_{1k} = \lambda_1$).

In this particular model, it is assumed that $\theta_i \sim N(0, 1)$. In addition, the attributes are also assumed to be conditionally independent given θ, as in,

$$P(\boldsymbol{\alpha_i}|\theta_i) = \prod_{\forall k} P(\alpha_{ik}|\theta_i).$$

The latter assumption allow us to generate $\boldsymbol{\alpha_i}$ by generating θ_i from $N(0, 1)$, and then α_{ik} given θ_i, $k = 1, \ldots, K$, individually. Compared to sampling from a multinomial distribution, the higher-order model allows us to sample $\boldsymbol{\alpha}$ without needing to know the corresponding $P(\boldsymbol{\alpha})$.

Sampling $P(\alpha_{ik} = 1|\theta_i)$ is identical to sampling $P(X_{ij} = 1|\theta_i)$, which will be discussed below.

Hierarchical Structure

If some the attributes are related such that the mastery of one attribute is a prerequisite to the mastery of another (i.e., attribute mastery is sequential), the structure of the attribute vectors is said to be hierarchical. An important implication of the model is that some attribute patterns are precluded by the structure. For example, if $K = 3$, and $\alpha_1 \rightarrow \alpha_2, \alpha_3$ (read, mastery of α_1 is a prerequisite to the mastery of either α_2 or α_3, then the attribute patterns 010, 001, and 011 are deemed impossible. Thus, instead of eight, we only need to deal with five attribute patterns.

To generate $\boldsymbol{\alpha}$ from a hierarchical structure, we can employ the same procedure used in the saturated attribute structure. However, we need to make sure that impossible attribute patterns are assigned a probability of 0.

The code below can be used to replace the original definition of `ex`.

```
decl ex=<1,1,0,0,1,1,0,1>'; //three attribute patterns are deemed impossible
ex= ex/sumc(ex); // normalization to ensure the probabilities add to 1
```

Examine the output for the code, and determine which of the possible attribute patterns has the highest (lowest) frequency.

Note that, in this code, the possible attribute patterns are equiprobable. However, this need not always be the case. For example, if we have reasons to believe that 000 is twice as likely as the remaining

attribute patterns, we can declare `ex=<2,1,0,0,1,1,0,1>'` instead.

Generating Binary Item Response Data with Underlying Continuous Latent Variable

The most typical models for binary item response data involve a unidimensional continuous latent variable. These models are the 1PL, 2PL and 3PL, which are under the logistic item response model family.

Recall that the 3PL is of the form:

$$P(X_{ij} = 1|\theta_i) = \gamma_j + (1 - \gamma_j)\frac{\exp[1.7\alpha_j(\theta_i - \beta_j)]}{1 + \exp[1.7\alpha_j(\theta_i - \beta_j)]},$$

and constraining the appropriate item parameters will yield the 1PL and the 2PL.

The Ox function below can be used to generate items from the 3PL. Note that including several variables as arguments allows us to minimize our number of global variables to be declared.

```
decl N=10, J=5;
gen3PL(decl t, a, b, c)
{
decl tmp=exp(1.7*(t*a'-ones(N,1)*(a.*b)'));
tmp=c'+ones(N,1)*(1-c)'.*tmp./(1+tmp);
return tmp.>ranu(N,J);
}
```

The function needs to be called from the `main`.

```
ranseed(130930);
decl theta, alpha, beta, gamma;
decl X;
theta=rann(N,1);
theta=sortc(theta); //sorting the abilities from lowest to highest
alpha=.75+ranu(J,1);
beta=.90*rann(J,1);
beta=sortc(beta);
gamma=.25*ranu(J,1);
print("\nItem Parameters:",alpha~beta~gamma);
X=gen3PL(theta, alpha, beta, gamma);
print("\nTheta & Data:",theta~X);
```

Modify the item parameters to generate 1PL and 2PL data.

For multidimensional IRT models, we only need to modify how the original `tmp` is computed. Of course, MIRT models also involve $\boldsymbol{\alpha}_j$ and $\boldsymbol{\theta}_i$, and may require using δ_j instead of β_j.

Generating Polytomous Item Response Data with Underlying Continuous Latent Variable

In addition to dichotomous response, we also have to deal with polytomous response (i.e., $X = 0, 1, \ldots, m-1$). Polytomous response can be classified as either ordered or unordered. *Generalized partial credit* and *graded response* models are example of IRT models for ordered polytomous response, whereas the *nominal* and *multiple-choice* models are examples of IRT models for unordered polytomous response.

Although the different polytomous models calculate $P(X_{ij} = x|\theta)$ differently, the same process can be used to generate polytomous data regardless of how $P(X_{ij} = x|\theta)$ has been calculated. For this reason, we will focus only on one model, the GPCM.

Recall that, in the GPCM,

$$P(X_{ij} = x|\theta_i) = \frac{\exp\left[\sum_{k=0}^{x} 1.7\alpha_j(\theta_i - \beta_{jk})\right]}{\sum_{l=0}^{m-1} \exp\left[\sum_{k=0}^{l} 1.7\alpha_j(\theta_i - \beta_{jk})\right]}.$$

For identifiability, we can constraint $1.7\alpha_j(\theta_i - \beta_{j0}) = 0$.

Another Detour: The Inverse CDF Method

Inverse CDF (also, *inverse transform*, among many other names) sampling is a basic method for sampling numbers at random from any probability distribution given its cumulative distribution function (CDF). Although this procedure is applicable to both continuous and discrete distributions, we will employ the procedure only for the latter.

The basic idea is to sample u from $U(0, 1)$, and then return the largest number x from the domain of the distribution f_X such that $P(X \leq x) \leq u$.

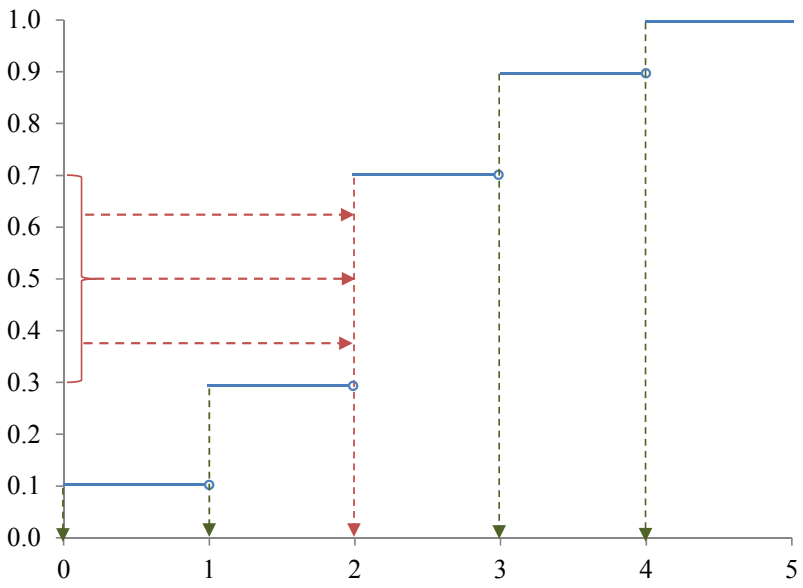
We illustrate this method using the random variable X, with the following probability and cumulative probability distribution functions.

X	$P(X = x)$	$P(X \leq x)$
0	0.1	0.1
1	0.2	0.3
2	0.4	0.7
3	0.2	0.9
4	0.1	1.0

If our sampling procedure is correct, 10% of the time, we should get a "0" or a "4", 20% of the time, a "1" or a "3", and 40% of the time, a "2".

If we sample u from $U(0, 1)$. 40% of the time, $0.3 \leq u \leq 0.7$. The graph below shows that, for $X = 0, 1, 2$, $P(X \leq x) \leq u$. Accordingly, we choose $X = 2$ whenever $0.3 \leq u \leq 0.7$.

A roundabout way to implement this procedure is to use several `if-else` statements. The following code, where $Px = P(X \leq x)$, can sample correctly from this distribution.




```
if(u<P0) X=0;
else if(u<P1) X=1;
else if(u<P2) X=2;
else if(u<P3) X=3;
else X=4;
```

However, this code is inefficient, particularly if we need to generate an $N \times J$ matrix of responses as it will require several `for` loops. Below is an equivalent, but more efficient code that can be used with matrices.

```
X=(u>P0)+(u>P1)+(u>P2)+(u>P3);
```

End of Detour

We can use the Inverse CDF procedure to efficiently sample polytomous responses. The following code generates the response category probabilities using the GPCM for $m = 5$.

```
decl N=10, J=5, m=5;
decl p0, p1, p2, p3, p4;
```

```
probGPC(decl t, a, b) //specifically for m=5
{
decl pp, pp0, pp1, pp2, pp3, pp4;
pp0=ones(N,J);
pp1=exp(1.7*(t*a'-ones(N,1)*(a.*b[][0])'));
pp2=pp1.*exp(1.7*(t*a'-ones(N,1)*(a.*b[][1])'));
pp3=pp2.*exp(1.7*(t*a'-ones(N,1)*(a.*b[][2])'));
pp4=pp3.*exp(1.7*(t*a'-ones(N,1)*(a.*b[][3])'));
pp=pp0+pp1+pp2+pp3+pp4;
p0=pp0./pp;
p1=pp1./pp;
p2=pp2./pp;
p3=pp3./pp;
p4=pp4./pp;
}
```

Given the computed probabilities, the code below converts the probabilities into item responses.

```
genGPC(decl aX) //specifically for m=5
{
decl P0, P1, P2, P3;
P0=p0;
P1=P0+p1;
P2=P1+p2;
P3=P2+p3;
decl u=ranu(N,J);
aX[0]=(u.>P0)+(u.>P1)+(u.>P2)+(u.>P3);
}
```

Finally, to use these functions, we need to call them in `main`. Before we can do that, we also need to declare and initialize a few variables.

```
ranseed(130930);
decl theta, alpha, beta;
decl X;
theta=rann(N,1);
theta=sortc(theta); //sorting the abilities from lowest to highest
alpha=.75+ranu(J,1);
beta=.90*rann(J,m-1);
```

```
beta=sortr(beta); // sorting the category difficulties from easiest to hardest
beta=sortbyc(beta,0); // sorting the item from easiest to hardest
print("\nItem Parameters:",alpha~beta);
probGPC(theta, alpha, beta);
genGPC(&X);
print("\nTheta & Data:",theta~X);
```

Generating Binary Item Response Data with Underlying Discrete Latent Variable

In this section, we will consider data generation procedures for two types of CDMs: reduced and saturated. For the former, we will use the DINA model, and for the latter, the G-DINA model. We assume that the matrix $\mathbf{A}_{N\times K}$ of attribute vectors has already been generated.

DINA Model

This function `alphaeta` below will require a few of the global variables. This function simply requires the $N \times K$ matrix `alpha0` as input and converts it to the $N \times J$ matrix `eta` based on the attribute specification in the $J \times K$ matrix `Q`.

```
decl N=10, J=6, K=3;
decl alpha; //all possible attribute patterns
decl Q;
decl alpha0, eta; //alpha0->generated attribute vectors; eta->converted from alpha0
alphaeta(decl a)
{
eta=a*Q';
eta=eta.==(ones(N,1)*sumc(Q'));
}
```

In addition, we need the following in `main`: initializing the Q-matrix, and the guessing and slip parameters. Once `alphaeta` has been called, we can create the matrix of probabilities, and then convert it to the matrix of item responses.

```
Q=<1,0,0; 0,1,0; 0,0,1; 1,1,0; 1,0,1; 0,1,1>;
decl g=.05+.25*ranu(J,1);
decl s=.05+.25*ranu(J,1);
alphaeta(alpha0);
decl prob=(1-eta).*(ones(N,1)*g') + eta.*(ones(N,1)*(1-s'));
decl X=prob.>ranu(N,J);
print(alpha0, eta);
print(prob,X);
```

Modify `alphaeta` to generate responses for the DINO model.

G-DINA Model

Because the G-DINA model is more general, we can expect the code to generate the item responses to also be more complex, particularly if the code has to have sufficient generality.

In the G-DINA model, Item j can have as many as 2^{K_j} unique parameters, where $K_j = \sum_{\forall k} q_{jk}$. In this dicussion, we will use $P(X_j = 1|\boldsymbol{\alpha}_{lj})$ as our parameters, where $\boldsymbol{\alpha}_{lj}$ denotes the reduced attribute vector for Item j.

Let $K_{j(max)} = \max(K_1, ..., K_J)$. We can collect all the item parameters in the $J \times 2^{K_{j(max)}}$ matrix $\mathbf{B}$. For $K_j < K_{j(max)}$, we only need entries for the first 2^{K_j} columns.

The importance of clarifying the meaning of the columns cannot be overstated. For $K_j = 1$, the relevant columns, from left to right, represent the reduced attribute vectors 0 and 1; for $K_j = 2$, 00, 10, 01 and 11; for $K_j = 3$, 000, 100, 010, 001, 110, 101, 011 and 111; and so forth.

Determining $P(X_{ij} = 1|\boldsymbol{\alpha}_i)$ is a matter of determining the location of $\boldsymbol{\alpha}_{ij}$, and assigning the corresponding item parameter (i.e., probability). To implement this in Ox, we need we additional functions.

First is the function `patternj`. This function is identical to the earlier function `pattern`, except that instead of fixed K, this function uses K_j to create all the possible *reduced* attribute patterns.

```
patternj(decl Kjj)
{
decl eta=<>;
for(decl k=1;k<=Kjj;k++){
decl tmp=<>;
decl base=zeros(2^(Kjj-k),1)|ones(2^(Kjj-k),1);
for(decl l=0;l<2^(k-1);l++){
tmp=tmp|base;}
eta=eta~tmp;}
eta=eta~sumr(eta);
decl tmp=zeros(2^Kjj,1);
for(decl k=0;k<Kjj;k++){
tmp=tmp-10^(Kjj-k-1)*eta[][k];
}
eta=eta~tmp;
eta=sortbyc(eta,Kjj~(Kjj+1));
return(eta[][0:Kjj-1]);
}
```

The other function is `probj`, which determines the probability of correct response to Item j for all the N examinees.

```
probj(decl jj)
{
decl alpha_j=selectifc(alpha0,Q[jj][].==1);
decl Kj=sumr(Q[jj][]);
decl eta_j=patternj(Kj);
decl prob_j=zeros(N,1);
for(decl i=0;i<N;i++){
decl tmp=vecindex(prodr((ones(2^Kj,1)*alpha_j[i][]).==eta_j).==1);
prob_j[i]=beta[jj][tmp];
}//end i
return(prob_j);
}
```

In `main`, we need to define the item parameter matrix `beta`, which is declared as a global variable. Currently, `beta` contains the parameters for the DINA model. However, by changing the entries of `beta`, we can generate item responses for other CDMs.

Note that the probabilities are determined one item at a item, so this program is not as efficient as the program for a specific reduced model, such as for the DINA model. As alluded to earlier, there is always a tradeoff between efficiency and generality.

```
beta=<
.2,.8, 0, 0;
.2,.8, 0, 0;
.2,.8, 0, 0;
.2,.2,.2,.8;
.2,.2,.2,.8;
.2,.2,.2,.8>;
decl X=<>;
for(decl j=0;j<J;j++)
{
X=X~probj(j);
}
print(X);
X=X.>ranu(N,J);
```

Modify `beta` to generate responses for the DINO model.

Reading and Writing Files

Although simulation studies can be done one fell swoop, it is always advisable to break them down into smaller steps. For example, for a simulation study that has the following three components, data generation, analysis of data, and summarizing the results, it would be a good idea to save the data after they have been generated, perform the analysis as a separate step, and carry out the summarization

as another step. By doing it step-by-step, it would be easier to redo or extend the simulation study when the situation calls for it.

In Ox, reading and writing files can be done in many ways. However, knowing a few basic templates would suffice. We can illustrate by solving the following problem.

We are interested in examining the behavior of the sample mean as a function of sample size ($N = 25, 100$), and the underlying distribution ($N(0,1)$ and $U(-3,3)$. Specifically, we want to examine the mean and variance of the sample mean across $R = 10$ replications.

Even though the problem is simple enough, we will carry out the data generation as one step, and the remaining procedure as another step.

The program is short enough so we can declare a few global variables.

```
decl N=<25,100>;
decl dist={"Normal","Uniform"};
decl R=10;
```

The following code in `main` will generate $2 \times 2 \times 10 = 40$ data sets that will written to files. The files can be uniquely identified by the name of the distribution, sample size, and replications number.

```
decl X, file, tmp;
for(decl i=0; i<2; i++){
for(decl j=0; j<2; j++){
for(decl r=0; r<R; r++){
if (j==0) X=rann(N[i],1);
else X=-3+6*ranu(N[i],1);
tmp=sprint(dist[j],"_",N[i],"_",r,".dat");
file=fopen(tmp,"w");
fprint(file,"%#M",X);
fclose(file);
}}}//end r, j, i
```

Most of the code for generating the data can be recycled to read the files. As we read each file, we are also computing the corresponding mean and saving it in the right cell of the $R \times 4$ matrix `mean`.

The variables `r` and `count` allows us to identify the correct rows and columns, respectively.

Finally, we print the summary as a screen output.

Incidentally, the default number of columns of a matrix that the screen output can accommodate is six. To increase the number of characters, hence, columns, use `format(int)`, where `int` is the number

of characters. For example, `format(1000)`.

```
decl mean=zeros(R,4);
decl X, file, tmp;
decl count=0;
for(decl i=0; i<2; i++){
for(decl j=0; j<2; j++){
for(decl r=0; r<R; r++){
tmp=sprint(dist[j],"_",N[i],"_",r,".dat");
file=fopen(tmp);
fscan(file,"%#M",N[i],1,&X);
fclose(file);
mean[r][count]=meanc(X);
}//end r
count=count+1;
}}//end j, i
print("\n    Normal25     Uniform25    Normal100    Uniform100");
print("\nMn: ", meanc(mean),"Var",varc(mean));
```

Based on results in sampling theory, we know that $E(\bar{x}) = \mu_x$, and $\sigma^2_{\bar{x}} = \sigma^2_x/N$, where N is the sample size. By the way, in case you do not remember, for the uniform distribution, $\sigma^2_x = (b-a)^2/12$, where a and b are the lower and upper limits, respectively.

How do the sample statistics compare with their expected values? Try increasing R to see if it has any impact on the results.

Additional Exercises

1. Generate $R = 100$ data sets for $N = 500$ examiness and $J = 25$ items that conform to the 3PL. Fix $\alpha_j = 1.0$ and $\gamma_j = 0.2$, but, for each replication, sample θ_j and β_j from $N(0, 1)$.

2. Find the MLE ($\hat{\theta}_i$) and EAP ($\tilde{\theta}_i$) of θ_i.

You may use last week's optimization function to find the MLE. For the EAP, recall that the expected a posteriori is

$$E(\theta|\boldsymbol{X}_i) = \int \theta \cdot p(\theta|\boldsymbol{X}_i) d\theta,$$

where

$$p(\theta|\boldsymbol{X}_i) = \frac{L(\boldsymbol{X}_i|\theta) \cdot p(\theta)}{\int L(\boldsymbol{X}_i|\theta) \cdot p(\theta) d\theta}.$$

More often than not, we assume $p(\theta) \sim N(0, 1)$.

But as we have learned in the past, we can approximate any continuous distribution with a discrete distribution, and replace integration with summation. The posterior distribution can be approximated

as

$$p(A_q|\boldsymbol{X}_i) = c \cdot L(\boldsymbol{X}_i|A_q) \cdot W(A_q),$$

where A_q is a quadrature node, $W(A_q)$ is its corresponding weight, and

$$c^{-1} = \sum_{q=1}^{Q} L(\boldsymbol{X}_i|A_q) \cdot W(A_q).$$

Once we have the discrete version of the posterior distribution, we can easily approximate $E(\theta|\boldsymbol{X}_i)$ as

$$\sum_{q=1}^{Q} A_q \cdot p(A_q|\boldsymbol{X}_i).$$

3. For each replication, find the correlation $Cor(\breve{\theta}_i, \theta_i)$, bias, $\sum_{\forall i}(\breve{\theta}_i - \theta_i)/\text{N}$, and MSE, $\sum_{\forall i}(\breve{\theta}_i - \theta_i)^2/N$, where $\breve{\theta} = \hat{\theta}, \tilde{\theta}$.

4. Average the correlation, bias, and MSE across the replications. Determine which is the better estimate, MLE or EAP? Is the result consistent across the different criteria?

Optional Exercises

1. In the higher-order attribute structure model, we can obtain the *marginal* probability of $\boldsymbol{\alpha}_l$ from the *joint* probability $P(\boldsymbol{\alpha}_l, \theta)$ by integrating out θ, as in,

$$P(\boldsymbol{\alpha}_l) = \int P(\boldsymbol{\alpha}_l, \theta) d\theta = \int P(\boldsymbol{\alpha}_l|\theta) p(\theta) d\theta \approx \sum_{\forall q} P(\boldsymbol{\alpha}_l|A_q) W(A_q).$$

Recall that $P(\boldsymbol{\alpha}_l|\theta) = \prod_{\forall k} P(\alpha_{lk}|\theta)$. Note that $P(\alpha_{lk}|\theta)$ is a shorthand of writing $P(\alpha_{lk} = a|\theta)$, $a = 0, 1$. For $a = 1$, $P(\alpha_{lk}|\theta) = P(\alpha_k = 1|\theta)$; for $a = 0$, $P(\alpha_{lk}|\theta) = 1 - P(\alpha_k = 1|\theta)$.

For example, $P(101|\theta)$ can be written as $P(\alpha_1 = 1|\theta) \times [1 - P(\alpha_2 = 1|\theta)] \times P(\alpha_k = 1|\theta)$.

More generally, we can write the $P(\boldsymbol{\alpha}_l|\theta)$ as

$$P(\alpha_1 = 1|\theta)^{\alpha_{l1}} \times [1 - P(\alpha_1 = 1|\theta)]^{1-\alpha_{l1}} \times \ldots \times P(\alpha_K = 1|\theta)^{\alpha_{lK}} \times [1 - P(\alpha_K = 1|\theta)]^{1-\alpha_{lK}}.$$

For $K = 5$, $\lambda_{1k} = 2$, $\lambda_{0k} = (-1.0, -0.5, 0.0, 0.5, 1.0)'$, and $\theta \sim N(0, 1)$, find $P(\boldsymbol{\alpha}_l)$, for $l = 1, \ldots, 32$.

As an alternative to using the integration formula above, we can find the marginal probabilities computationally by generating an extremely large number of attribute vectors, and recording their relative frequencies.

It can be noted that, by using these probabilities to generate the attribute vectors using the multinomial distribution, the attribute vectors will have the same (more or less, subject to Monte Carlo errors) relative frequencies. Hence, given the relative frequencies alone, we cannot be certain that the true generating model is the saturated or the higher-order model.

For $\lambda_{1k} = 0$, show that $P(\boldsymbol{\alpha}_l) = 1/2^K$. Note that $P(\alpha_k = 1|\theta) = P(\alpha_k = 1) = 0.5$.

This mean that, when the attributes are unrelated to the higher-order trait, the higher-order structure is equivalent to the *uniform attribute structure*. Note that the exact values of the difficulty parameters are immaterial.

2. For greater flexibility, the higher-order model can be written in the slope-intercept form, as in,

$$P(\alpha_{ik} = 1|\theta_i) = \frac{\exp(\gamma_{0k} + \gamma_{1k}\theta_i)}{1 + \exp(\gamma_{0k} + \gamma_{1k}\theta_i)}.$$

This allows us to set the value of γ_{1k} independent of γ_{0k}. For example, even if $\gamma_{1k} = 0$, the value of γ_{0k} still matters.

When $\gamma_{1k} = 0$, the higher-order model in the slope-intercept form reduces to

$$P(\alpha_{ik} = 1|\theta_i) = P(\alpha_{ik} = 1) = p_k = \frac{\exp(\gamma_{0k})}{1 + \exp(\gamma_{0k})},$$

where p_k is the proportion of individuals in the population who mastered attribute k.

Thus, $P(\boldsymbol{\alpha}_l)$ simplifies to

$$p_1^{\alpha_{l1}} \times q_1^{1-\alpha_{l1}} \times \ldots \times p_K^{\alpha_{lK}} \times q_K^{1-\alpha_{lK}},$$

for $q_k = 1 - p_k$.

In addition to being able to compute the marginally probability of an attribute more easily (e.g., $P(101) = p_1 q_2 p_3$, without any integration), we can also generate the attribute vectors more easily. As in, α_k is sampled independently from $Ber(p_k)$.

This is the *independence model* (i.e., the attributes are independent of each other).

When both $\gamma_{0k} = \gamma_{1k} = 0$, the independence model produces what attribute structure?

For $K = 5$, $\gamma_{1k} = 0$, and $\gamma_{0k} = (-2.0, -1.0, 0.0, 1.0, 2.0)'$, find p_k, for $k = 1, \ldots, 5$, and $P(\boldsymbol{\alpha}_l)$, for $l = 1, \ldots, 32$. Solve this problem in two ways: analytically and computationally.

Latent Variable Modeling and Statistical Computing
Spring 2016

MMLE & EM Algorithms - Part I

Some Background: Structural vs. Incidental Parameters

In latent variable modeling, we need to distinguish between structural and incidental parameters. For example, in item response theory and cognitive diagnosis models, the structural parameters are the item (e.g., α_j, β_j, g_j, s_j) parameters; the incidental parameters are the person-specific parameters (e.g., θ_i, $\boldsymbol{\alpha}_i$).

Another way of viewing the distinction between structural and incidental parameters is to take different samples, and note which parameters should remain the same (up to some random errors), and which should not. The former are the structural parameters, the latter the incidental parameters.

With small enough data and powerful enough optimization tool, we might be tempted to maximize, say, the likelihood function to estimate the structural and incidental parameters simultaneously via joint maximum likelihood estimation. Theoretically, this is not advisable. Neyman & Scott (1948) have shown that in the presence of incidental parameters, the maximum likelihood estimates of structural parameters may not be consistent.

Marginalized Maximum Likelihood Estimation

Recall that $P(A) = \int P(A,B)dB = \int P(A|B) \times P(B)dB$. In the same way, instead of maximizing the likelihood function, which contains both structural and incidental parameters, we can work with the marginalized maximum likelihood function, where the incidental parameters are integrated out. That is, we will maximize the marginalized likelihood

$$L(\boldsymbol{X}_i) \quad = \quad \int L(\boldsymbol{X}_i|\theta) \times g(\theta)d\theta.$$

The integral can be approximated by using summation and quadrature nodes, as in,

$$L(\boldsymbol{X}_i) \quad \approx \quad \sum_{q=1}^{Q} L(\boldsymbol{X}_i|A_q) \times w(A_q).$$

For N individuals, the (marginalized) likelihood is

$$L(\mathbf{X}) \quad = \quad \prod_{i=1}^{N} L(\boldsymbol{X}_i).$$

Instead of the likelihood, it would be more convenient to work with the loglikelihood

$$l(\mathbf{X}) \quad = \quad \log L(\mathbf{X}) = \sum_{i=1}^{N} \log L(\boldsymbol{X}_i).$$

Let ξ_j be the parameter of interest. To obtain an estimate of ξ_j, we need to find

$$\frac{\partial l(\mathbf{X})}{\partial \xi_j} \quad = \quad \sum_{i=1}^{N} \frac{1}{L(\boldsymbol{X}_i)} \times \frac{\partial L(\boldsymbol{X}_i)}{\partial \xi_j}.$$

The derivative on the right-hand side is

$$\frac{\partial L(\boldsymbol{X}_i)}{\partial \xi_j} = \sum_{q=1}^{Q} w(A_q) \times \frac{\partial L(\boldsymbol{X}_i|A_q)}{\partial \xi_j}.$$

For dichotomous response (i.e., $X_{ij} = 0, 1$),

$$L(\boldsymbol{X}_i|A_q) = \prod_{j=1}^{J} P_{jq}^{X_{ij}} \times (1 - P_{jq})^{1-X_{ij}},$$

where $P_{jq} = P(X_j = 1|A_q)$. So

$$\begin{aligned}
\frac{\partial L(\boldsymbol{X}_i|A_q)}{\partial \xi_j} &= \prod_{j' \neq j} \left[P_{j'q}^{X_{ij'}} \times (1 - P_{j'q})^{1-X_{ij'}} \right] \times \frac{\partial \left[P_{jq}^{X_{ij}} \times (1 - P_{jq})^{1-X_{ij}} \right]}{\partial \xi_j} \\
&= \prod_{j' \neq j} \left[P_{j'q}^{X_{ij'}} \times (1 - P_{j'q})^{1-X_{ij'}} \right] \times P_{jq}^{X_{ij}} \times (1 - P_{jq})^{1-X_{ij}} \left[\frac{X_{ij} - P_{jq}}{P_{jq}(1 - P_{jq})} \right] \frac{\partial P_{jq}}{\partial \xi_j} \\
&= \prod_{j'=1}^{J} \left[P_{j'q}^{X_{ij'}} \times (1 - P_{j'q})^{1-X_{ij'}} \right] \left[\frac{X_{ij} - P_{jq}}{P_{jq}(1 - P_{jq})} \right] \frac{\partial P_{jq}}{\partial \xi_j} \\
&= L(\boldsymbol{X}_i|A_q) \left[\frac{X_{ij} - P_{jq}}{P_{jq}(1 - P_{jq})} \right] \frac{\partial P_{jq}}{\partial \xi_j}
\end{aligned}$$

Therefore,

$$\frac{\partial L(\boldsymbol{X}_i)}{\partial \xi_j} = \sum_{q=1}^{Q} w(A_q) \times L(\boldsymbol{X}_i|A_q) \left[\frac{X_{ij} - P_{jq}}{P_{jq}(1 - P_{jq})} \right] \frac{\partial P_{jq}}{\partial \xi_j}.$$

It follows that

$$\begin{aligned}
\frac{\partial l(\mathbf{X})}{\partial \xi_j} &= \sum_{i=1}^{N} \frac{1}{L(\boldsymbol{X}_i)} \sum_{q=1}^{Q} w(A_q) \times L(\boldsymbol{X}_i|A_q) \left[\frac{X_{ij} - P_{jq}}{P_{jq}(1 - P_{jq})} \right] \frac{\partial P_{jq}}{\partial \xi_j} \\
&= \sum_{i=1}^{N} \sum_{q=1}^{Q} \frac{1}{L(\boldsymbol{X}_i)} L(\boldsymbol{X}_i|A_q) \times w(A_q) \left[\frac{X_{ij} - P_{jq}}{P_{jq}(1 - P_{jq})} \right] \frac{\partial P_{jq}}{\partial \xi_j} \\
&= \sum_{i=1}^{N} \sum_{q=1}^{Q} P(A_q|\boldsymbol{X}_i) \left[\frac{X_{ij} - P_{jq}}{P_{jq}(1 - P_{jq})} \right] \frac{\partial P_{jq}}{\partial \xi_j}
\end{aligned}$$

Let P_{jq} be the 2PL, as in,

$$P_{jq} = \frac{\exp\left[1.7\alpha_j(A_q - \beta_j)\right]}{1 + \exp\left[1.7\alpha_j(A_q - \beta_j)\right]}.$$

This means we have a vector of parameters: $\boldsymbol{\xi}_j = (\alpha_j, \beta_j)$. To maximize the loglikelihood, we need to set the gradient $g(\boldsymbol{\xi})$ to $\mathbf{0}$, which is equivalent to

$$\begin{pmatrix} \frac{\partial l(\mathbf{X})}{\partial \alpha_j} \\ \frac{\partial l(\mathbf{X})}{\partial \beta_j} \end{pmatrix} = \mathbf{0}.$$

To solve these equations, we first show that

$$\begin{aligned}\frac{\partial P_{jq}}{\partial \alpha_j} &= 1.7 \times P_{jq} \times (1 - P_{jq}) \times (A_q - \beta_j) \\ \frac{\partial P_{jq}}{\partial \beta_j} &= -1.7 \times P_{jq} \times (1 - P_{jq}) \times \alpha_j\end{aligned}$$

It follows that

$$\begin{aligned}\frac{\partial l(\mathbf{X})}{\partial \alpha_j} &= \sum_{i=1}^{N}\sum_{q=1}^{Q} P(A_q|\boldsymbol{X}_i)\left[\frac{X_{ij} - P_{jq}}{P_{jq}(1 - P_{jq})}\right][1.7 \times P_{jq} \times (1 - P_{jq}) \times (A_q - \beta_j)] \\ &= 1.7\sum_{i=1}^{N}\sum_{q=1}^{Q} P(A_q|\boldsymbol{X}_i) \times (X_{ij} - P_{jq}) \times (A_q - \beta_j) \\ \frac{\partial l(\mathbf{X})}{\partial \beta_j} &= \sum_{i=1}^{N}\sum_{q=1}^{Q} P(A_q|\boldsymbol{X}_i)\left[\frac{X_{ij} - P_{jq}}{P_{jq}(1 - P_{jq})}\right][-1.7 \times P_{jq} \times (1 - P_{jq}) \times \alpha_j] \\ &= -1.7\alpha_j\sum_{i=1}^{N}\sum_{q=1}^{Q} P(A_q|\boldsymbol{X}_i) \times (X_{ij} - P_{jq})\end{aligned}$$

An aside for now: These derivatives can be simplified as

$$\begin{aligned}\frac{\partial l(\mathbf{X})}{\partial \alpha_j} &= 1.7\sum_{q=1}^{Q}(A_q - \beta_j)\left[\sum_{i=1}^{N} P(A_q|\boldsymbol{X}_i) \times X_{ij} - P_{jq}\sum_{i=1}^{N} P(A_q|\boldsymbol{X}_i)\right] \\ &= 1.7\sum_{q=1}^{Q}(A_q - \beta_j) \times [R_{jq} - P_{jq} \times N_q] \\ \frac{\partial l(\mathbf{X})}{\partial \beta_j} &= 1.7\alpha_j\sum_{q=1}^{Q}\left[\sum_{i=1}^{N} P(A_q|\boldsymbol{X}_i) \times X_{ij} - P_{jq}\sum_{i=1}^{N} P(A_q|\boldsymbol{X}_i)\right] \\ &= -1.7\alpha_j\sum_{q=1}^{Q}[R_{jq} - P_{jq} \times N_q]\end{aligned}$$

where R_{jq} is the *expected* number of correct response on item j for quadrature node A_q, and N_q is the expected number of examinees in quadrature node A_q.

Because we are solving the zero of the function, we can use Newton's method, which is a zero-finder. To implement Newton's method, we need $\nabla g(\boldsymbol{\xi}_j)$, the Jacobian matrix of $g(\boldsymbol{\xi}_j)$ at $\boldsymbol{\xi}_j$, which is, in this case, a 2×2 matrix of second derivatives.

The following are the entries of $\nabla g(\boldsymbol{\xi}_j)$:

$$\begin{aligned}
\frac{\partial^2 l^*(\mathbf{X})}{\partial \alpha_j^2} &= -1.7^2 \sum_{i=1}^{N} \sum_{q=1}^{Q} P(A_q|\boldsymbol{X}_i) \times (A_q - \beta_j)^2 \times P_{jq} \times (1 - P_{jq}) \\
\frac{\partial^2 l^*(\mathbf{X})}{\partial \beta_j^2} &= -1.7^2 \sum_{i=1}^{N} \sum_{q=1}^{Q} P(A_q|\boldsymbol{X}_i) \times \alpha_j^2 \times P_{jq} \times (1 - P_{jq}) \\
\frac{\partial^2 l^*(\mathbf{X})}{\partial \alpha_j \partial \beta_j} &= -1.7 \sum_{i=1}^{N} \sum_{q=1}^{Q} P(A_q|\boldsymbol{X}_i) \times [(X_{ij} - P_{jq}) - 1.7\alpha_j \times (A_q - \beta_j) \times P_{jq} \times (1 - P_{jq})]
\end{aligned}$$

At cycle $c+1$, the updated estimate of $\boldsymbol{\xi}_j$ is given by

$$\boldsymbol{\xi}_j^{c+1} = \boldsymbol{\xi}_j^{c} - [\nabla g(\boldsymbol{\xi}_j^{c})]^{-1} g(\boldsymbol{\xi}_j^{c}).$$

As can be seen from the above equation, Newton's method is an iterative solution. However, it should be noted that the Newton cycles are nested within iteration t. We can denote the nested relationship as follows:

$$\boldsymbol{\xi}_j^{t(c+1)} = \boldsymbol{\xi}_j^{t(c)} - [\nabla g(\boldsymbol{\xi}_j^{t(c)})]^{-1} g(\boldsymbol{\xi}_j^{t(c)}).$$

Once the Newton step has converged, we can suppress the superscript for the Newton cycle, and write the item parameter estimate at iteration t as $\boldsymbol{\xi}_j^t$.

Before we can proceed with iteration $t+1$, we need to update the posterior distributions using the most current estimates (i.e., $\boldsymbol{\xi}_j^t$).

This process is repeated until no substantial improvement (i.e., parameter estimates or likelihood remains *essentially* the same) between two successive iterations can be observed.

Below is an rough sketch of how to implement the algorithm.

1. At iteration $t = 0$, assign initial values to the item parameters, as in, $\boldsymbol{\xi}_1^0, \ldots, \boldsymbol{\xi}_J^0$.
2. Compute the posterior distributions across the N examinees.
3. Find the updated estimates of the item parameters.
4. Repeat steps 2 and 3 until convergence.

To implement the MMLE algorithm, we will need to define the following $N \times Q$ matrices.

- $\mathbf{P} = \{P(A_q|\boldsymbol{X}_i)\}$ contains the posterior probabilities; each entry is potentially unique
- $\mathbf{R} = \{X_{ij} - P_{jq}\}$ contains the differences between observed and model-based responses; each entry is potentially unique, and computed as $< X_{1j}; \ldots; X_{Nj} > - < P_{j1}; \ldots; P_{jQ} >'$
- $\mathbf{D} = \{A_q - \beta_j\}$ contains the differences between quadrature node and the difficulty parameter; the rows are repeated, as in, $\mathbf{1}_{N\times 1} \times < A_1; \ldots; A_Q >' - \beta_j$;

Approximate Standard Errors of MLEs

The information matrix of the estimator of $\boldsymbol{\xi}$ is $\boldsymbol{I}(\boldsymbol{\xi}) = -E\{\partial^2 l(\boldsymbol{X})/\partial \boldsymbol{\xi}^2\}$. The second derivative of the marginalized loglikelihood for item j with respect to the parameters ξ_{jk} and $\xi_{jk'}$ is given by

$$\frac{\partial^2 l(\boldsymbol{X})}{\partial \xi_{jk} \partial \xi_{jk'}} = \sum_{i=1}^{N} \left[\frac{1}{L(\boldsymbol{X_i})} \times \frac{\partial L^2(\boldsymbol{X}_i)}{\partial \xi_{jk} \partial \xi_{jk'}} - \frac{1}{L^2(\boldsymbol{X_i})} \times \frac{\partial L(\boldsymbol{X}_i)}{\partial \xi_{jk}} \times \frac{\partial L(\boldsymbol{X}_i)}{\partial \xi_{jk'}} \right].$$

The first term will vanish when the expectation is taken. Hence,

$$\begin{aligned} \frac{\partial^2 l(\boldsymbol{X})}{\partial \xi_{jk} \partial \xi_{jk'}} &= \sum_{i=1}^{N} \left[- \left(\sum_{q=1}^{Q} p(A_q) \frac{\partial L(\mathbf{X}_i | A_q)}{\partial \xi_{jk}} \right) \times \frac{1}{L^2(\boldsymbol{X_i})} \times \frac{\partial L(\boldsymbol{X}_i)}{\partial \xi_{jk'}} \right] \\ &= -\sum_{i=1}^{N} \frac{1}{L^2(\boldsymbol{X_i})} \times \left[\sum_{q=1}^{Q} p(A_q) \frac{\partial L(\mathbf{X}_i | A_q)}{\partial \xi_{jk}} \right] \times \frac{\partial L(\boldsymbol{X}_i)}{\partial \xi_{jk'}}. \end{aligned}$$

Taking the derivatives will give us

$$\begin{aligned} -\sum_{i=1}^{N} \frac{1}{L^2(\boldsymbol{X_i})} &\left\{ \sum_{q=1}^{Q} p(A_q) \times L(\mathbf{X}_i | A_q) \times \left[\frac{X_{ij} - P_{jq}}{P_{jq}(1 - P_{jq})} \right] \times \frac{\partial P_{jq}}{\partial \xi_{jk}} \right\} \times \\ &\left\{ \sum_{q=1}^{Q} p(A_q) \times L(\mathbf{X}_i | A_q) \times \left[\frac{X_{ij} - P_{jq}}{P_{jq}(1 - P_{jq})} \right] \times \frac{\partial P_{jq}}{\partial \xi_{jk'}} \right\} \end{aligned}$$

We can distribute one $L^{-1}(\boldsymbol{X}_i)$ to each of the two factors in the braces to simplify them. For example, the first factor becomes

$$\begin{aligned} &\sum_{q=1}^{Q} \frac{p(A_q) L(\mathbf{X}_i | A_q)}{L(\boldsymbol{X}_i)} \times \left[\frac{X_{ij} - P_{jq}}{P_{jq}(1 - P_{jq})} \right] \times \frac{\partial P_{jq}}{\partial \xi_{j\eta}} \\ = &\sum_{q=1}^{Q} p(A_q | \boldsymbol{X}_i) \times \left[\frac{X_{ij} - P_{jq}}{P_{jq}(1 - P_{jq})} \right] \times \frac{\partial P_{jq}}{\partial \xi_{j\eta}}. \end{aligned}$$

Therefore,

$$\begin{aligned} \frac{\partial^2 l(\boldsymbol{X})}{\partial \xi_{jk} \partial \xi_{jk'}} = -\sum_{i=1}^{N} &\left\{ \sum_{q=1}^{Q} p(A_q | \boldsymbol{X}_i) \times \left[\frac{X_{ij} - P_{jq}}{P_{jq}(1 - P_{jq})} \right] \times \frac{\partial P_{jq}}{\partial \xi_{jk}} \right\} \times \\ &\left\{ \sum_{q=1}^{Q} p(A_q | \boldsymbol{X}_i) \times \left[\frac{X_{ij} - P_{jq}}{P_{jq}(1 - P_{jq})} \right] \times \frac{\partial P_{jq}}{\partial \xi_{jk'}} \right\}, \end{aligned}$$

which represents the sum of the cross-products of expected values based on the examinees' posterior distributions. Instead of computing the expectation, the information matrix can be approximated by evaluating $\partial^2 l(\boldsymbol{X})/(\partial \xi_{jk} \partial \xi_{jk'})$ at $\hat{\boldsymbol{\xi}}$ using the observed $\boldsymbol{X}$ resulting in $\boldsymbol{I}(\hat{\boldsymbol{\xi}})$. Finally, $\boldsymbol{I}^{-1}(\hat{\boldsymbol{\xi}})$ provides an approximation of $\text{Cov}(\hat{\boldsymbol{\xi}})$, and the root of its diagonal elements represents the $\text{SE}(\hat{\boldsymbol{\xi}})$.

Although the two second derivatives appear to be notationally the same, the second derivatives to obtain the standard errors do not invoke the use of the posterior distributions until *after* the necessary

derivatives have been taken.

For the 2PL, we will have the following second derivatives:

$$\begin{aligned}
\frac{\partial^2 l(\boldsymbol{X})}{\partial \alpha_j^2} &= -\sum_{i=1}^{N}\left\{\sum_{q=1}^{Q} p(A_q|\boldsymbol{X}_i) \times \left[\frac{X_{ij}-P_{jq}}{P_{jq}(1-P_{jq})}\right] \times \frac{\partial P_{jq}}{\partial \alpha_j}\right\}^2 \\
&= -\sum_{i=1}^{N}\left\{\sum_{q=1}^{Q} p(A_q|\boldsymbol{X}_i) \times \left[\frac{X_{ij}-P_{jq}}{P_{jq}(1-P_{jq})}\right] \times [1.7 \times P_{jq}(1-P_{jq}) \times (A_q-\beta_j)]\right\}^2 \\
&= -1.7^2 \sum_{i=1}^{N}\left\{\sum_{q=1}^{Q} p(A_q|\boldsymbol{X}_i) \times (X_{ij}-P_{jq}) \times (A_q-\beta_j)\right\}^2
\end{aligned}$$

$$\begin{aligned}
\frac{\partial^2 l(\boldsymbol{X})}{\partial \beta_j^2} &= -\sum_{i=1}^{N}\left\{\sum_{q=1}^{Q} p(A_q|\boldsymbol{X}_i) \times \left[\frac{X_{ij}-P_{jq}}{P_{jq}(1-P_{jq})}\right] \times \frac{\partial P_{jq}}{\partial \beta_j}\right\}^2 \\
&= -\sum_{i=1}^{N}\left\{\sum_{q=1}^{Q} p(A_q|\boldsymbol{X}_i) \times \left[\frac{X_{ij}-P_{jq}}{P_{jq}(1-P_{jq})}\right] \times [-1.7 \times P_{jq}(1-P_{jq}) \times \alpha_j]\right\}^2 \\
&= -1.7^2 \alpha_j^2 \sum_{i=1}^{N}\left\{\sum_{q=1}^{Q} p(A_q|\boldsymbol{X}_i) \times (X_{ij}-P_{jq})\right\}^2
\end{aligned}$$

$$\begin{aligned}
\frac{\partial^2 l(\boldsymbol{X})}{\partial \alpha_j \partial \beta_j} &= -\sum_{i=1}^{N}\left\{\sum_{q=1}^{Q} p(A_q|\boldsymbol{X}_i) \times \left[\frac{X_{ij}-P_{jq}}{P_{jq}(1-P_{jq})}\right] \times \frac{\partial P_{jq}}{\partial \alpha_j}\right\} \times \\
&\qquad \left\{\sum_{q=1}^{Q} p(A_q|\boldsymbol{X}_i) \times \left[\frac{X_{ij}-P_{jq}}{P_{jq}(1-P_{jq})}\right] \times \frac{\partial P_{jq}}{\partial \beta_j}\right\} \\
&= -\sum_{i=1}^{N}\left\{\sum_{q=1}^{Q} p(A_q|\boldsymbol{X}_i) \times \left[\frac{X_{ij}-P_{jq}}{P_{jq}(1-P_{jq})}\right] \times [1.7 \times P_{jq}(1-P_{jq}) \times (A_q-\beta_j)]\right\} \times \\
&\qquad \left\{\sum_{q=1}^{Q} p(A_q|\boldsymbol{X}_i) \times \left[\frac{X_{ij}-P_{jq}}{P_{jq}(1-P_{jq})}\right] \times [-1.7 \times P_{jq}(1-P_{jq}) \times \alpha_j]\right\} \\
&= -1.7^2 \alpha_j \sum_{i=1}^{N}\left\{\sum_{q=1}^{Q} p(A_q|\boldsymbol{X}_i) \times (X_{ij}-P_{jq}) \times (A_q-\beta_j)\right\}\left\{\sum_{q=1}^{Q} p(A_q|\boldsymbol{X}_i) \times (X_{ij}-P_{jq})\right\}
\end{aligned}$$

An Implementation of MMLE for 2PL in Ox: `2PL_MMLE_I.ox`

This program requires including only one header file: `#include <oxstd.h>`.

```
decl X,P,R,D,N=2000;
decl MaxIt=100;
decl MaxC=50;
decl MaxDiff=.001;
decl A,Q,W;
decl a,b;
decl J=15;
decl Par0,SE;
```

`X` is the $N \times J$ matrix of response data, `P` is the $N \times Q$ matrix of posterior probabilities, `R` is the $N \times Q$ matrix of $X_{iq} - P(X_j = 1|A_q)$, and `D` is the $N \times Q$ matrix of $A_q - \beta_j$.

`MaxIt`, `MaxC`, and `MaxDiff` are the maximum number of iterations, number of Newton cycles, and allowable change in item parameter estimates between two consecutive iterations or cycles, respectively.

`A`, `Q`, and `W` are the quadrature nodes, number of quadrature nodes, and weight of the quadrature nodes, respectively.

`a`, `b`, and `J` are discrimination parameters, difficulty parameters, and number of items, respectively.

`Par0` contains the parameter estimates, whereas `SE` the standard errors.

```
initiate(){
A=<-3.5:[.25]3.5>';   //quadrature nodes
Q=rows(A);
W=densn(A);          //quadrature weights
a=ones(J,1);         //specifying initial a parameters
b=.025+.95*meanc(X);  //generating initial b parameters based on the responses
b=-quann(b');
Par0=a~b;
SE=zeros(J,2);
}//end initiate
```

The first function we will discuss is `initiate`. This function gives initial values to the variables that will be used in the later functions.

At $t = 0$, `A`, `Q`, and `W` are initiated. All the discrimination parameters are assigned an initial value of 1. The difficulty parameter of item j is determined using the proportion correct p_j and its normal

quantile equivalent (after a correction to ensure that $0.025 \leq p_j \leq .975$).

```
update_post(){
P=<>;                 // posterior matrix: empty --> N x Q
for (decl i=0;i<N;i++){
decl ex=exp(1.7*(ones(Q,1)*Par0[][0]').*(A*ones(1,J)-ones(Q,1)*Par0[][1]'));//Q x J
decl p=ex./(1+ex);          //Q by J
decl Xi=ones(Q,1)*X[i][];  //Q by J
decl post=exp(sumr(Xi.*log(p)+(1-Xi).*log(1-p))).*W; //Q x 1
post=post./sumc(post);    // Q by 1
P=P|post';                  // posterior matrix: N by Q
}}// end i; end update_post
```

The next function is `update_post`, which is designed to update the posterior distributions at the end of each iteration.

For this functions, we start with a null `P`. This variable is built up one row at a time.

To find the posterior distribution for examinee i, we need the $\boldsymbol{X}_i$. However, it should be noted that A_q is used in place of θ_i.

Note also that although it is more stable to work with the loglikelihood, the posterior requires the actual likelihood, hence, the use of `exp()`.

To ensure that proper posteriors are calculated, a normalization step is required.

The next function, `newton`, is the most critical function of this program. This function estimates the parameters one item at a time, thus, requires the input `j`.

The first seven lines declare and initiate the variables that will used inside the function. Note that it is important to **reset** the `diff`, the largest difference in the parameter estimates between two cycles, and `it`, the number of Newton cycles, when a new item is involved.

It should be noted that although the function is called `newton`, it also contains steps for computing `R` and `D`. The computation of `P` is done in `update_post`.

As mentioned earlier, once we have `P`, `R` and `D`, finding the values of the gradient and Jacobian is a matter of plugging in the right variables.

`Parjtmp` represents the updated estimates of the parameters of item j at this particular cycle. The maximization continues for item j until `diff` remains large AND the number of cycles is below `MaxC`.

The last few lines update the initial values for next cycle, and add 1 to the cycle counter.

```
newton(const j){
decl Xj=X[][j]*ones(1,Q);    //N by Q
decl diff=0.1;
decl it=0;
decl aj=Par0[j][0]; // starts at the estimate from prior iteration
decl bj=Par0[j][1]; // starts at the estimate from prior iteration
decl Parjtmp;

while (diff>MaxDiff && it<MaxC){  //Newton cycle
decl d1=ones(2,1); // gradient vector
decl d2=ones(2,2); // Jacobian matrix
decl ex=exp(1.7*aj*(ones(N,1)*A'-bj'ones(N,Q))); //N x Q
decl Prob=ex./(1+ex);                  //N x Q
decl R=Xj-Prob; //N x Q
decl D=ones(N,1)*A'-bj; //N x Q
//Gradient
d1[0][0]=sumr(sumc(1.7*P.*R.*D));
d1[1][0]=sumr(sumc(-1.7*aj*P.*R));
//Jacobian (Hessian)
d2[0][0]=-sumr(sumc(1.7^2*P.*Prob.*(1-Prob).*(D.^2)));
d2[1][1]=-sumr(sumc(1.7^2*aj^2*P.*Prob.*(1-Prob)));
d2[1][0]=sumr(sumc(-1.7*P.*(R-1.7*aj*D.*Prob.*(1-Prob))));
d2[0][1]=d2[1][0];
Parjtmp=(aj|bj)-invert(d2)*d1;
diff=max(sqrt((Parjtmp-(aj|bj)).^2));
aj=Parjtmp[0];
bj=Parjtmp[1];
it++;
}//end Newton cycle
return Parjtmp;
}//end newton
```

As the name indicates, `findSE` is the function called to compute the standard errors *analytically* after the item parameters have been estimated.

It starts by making sure that the posteriors are computed using the most recent item parameters estimates. It then finds the standard errors one item at a time.

Except for `SE`, the variables in this function are identical to those in `newton`.

```
findSE(){
update_post();
for (decl j=0; j<J; j++){
decl Xj=X[][j]*ones(1,Q);
decl aj=Par0[j][0];
decl bj=Par0[j][1];

decl d2=ones(2,2);
decl ex=exp(1.7*aj*(ones(N,1)*A'-bj'ones(N,Q)));
decl Prob=ex./(1+ex);
decl R=Xj-Prob;
decl D=ones(N,1)*A'-bj;
d2[0][0]=-1.7^2*sumc(sumr(P.*R.*D).^2);
d2[1][1]=-1.7^2*sumc(sumr(aj*P.*R).^2);
d2[1][0]=-1.7^2*sumc(sumr(P.*R.*D).*sumr(aj*P.*R));
d2[0][1]=d2[1][0];
SE[j][]=sqrt(diagonal(-invert(d2)));
}//end j
}//end findSE
```

```
iterate(){
decl diff=0.1;
decl it=0;
while(diff>MaxDiff && it<MaxIt){
update_post();
decl Par1=zeros(J,2);
for (decl j=0;j<J;j++){
Par1[j][]=newton(j)';
}// end j
diff=max(sqrt((Par1-Par0).^2));
Par0=Par1;
it++;
}//end while
}//end iterate
```

The `iterate` function governs the movement from one iteration to another.

This function starts with declaring and initiating `diff` and `it`, which determine when the iterative process should be terminated.

`Par1` temporarily holds updated parameter estimates, which is obtained one item at a time. For each item, `newton` is called to do the heavy lifting.

After going through all the items, constraints are examined to determine whether another iteration is necessary. If it is, the updated estimates serve as the starting values and the iteration counter is increased by 1.

```
main(){
format(10000);
decl file=sprint("twopl.dat");
file=fopen(file);
fscan(file,"%#M",N,J,&X);
fclose(file);

initiate();
iterate();
findSE();
print("\n    The following results are based on MMLE_I algorithm \n");
print("\n          a           b          SE(a)        SE(b)");
print(Par0~SE);
}//end main
```

The `main` function reads the data, calls `initiate`, `iterate`, and `findSE`. Lastly, the estimates and standard errors are printed as screen output.

```
    The following results are based on MMLE_I algorithm

          a           b          SE(a)        SE(b)
      1.8298     -0.96470      0.12453     0.033606
      1.4379     -0.71420     0.085263     0.032320
      1.3799     -0.54251     0.076313     0.030351
      1.2632     -0.39560     0.070886     0.029683
      1.3484     -0.22642     0.070125     0.027642
      1.7499    -0.027444      0.10337     0.023287
      1.7306     0.053287     0.097310     0.023628
     0.77912      0.11880     0.044111     0.039933
      1.2035      0.33102     0.064823     0.030640
      1.2760      0.37469     0.069572     0.029967
      1.1632      0.71873     0.065910     0.037458
      1.6739      0.87003      0.10027     0.033419
      1.5440       1.0198     0.098429     0.037284
     0.79812       1.1151     0.050936     0.061815
      1.7701       1.7574      0.19376     0.067353
```

MMLE Estimation when Only the Gradient is Available: `2PL_MMLE_II.ox`

As you witness above, deriving the gradient and Hessian can be time consuming. In addition, if you are not really careful, it is also easy to make minor mistakes that can have major implications. This next program can be used to obtained the (marginalized) maximum likelihood estimate of the item parameters when you only have the time and patience to derive the gradient.

Without computing the second the second derivatives, we will obtain the standard errors *numerically* in conjunction with the gradient-based item parameter estimates. We will introduce this later.

For this program, in addition to including `oxstd.h`, we also need to import `maxsqp` and `solvenle` for this program.

The variables declared globally are the same as those in `2PL_MMLE_I`, except for `j` and `fL`. The former is the item identifier, and the latter will contain the full likelihood (i.e., the likelihoods based all the J items and the estimates from the previous iteration).

Both need to be declared as global variables for the functions in this program.

The function below, `TwoPL_NLE`, contains `avF[0]`, which holds the system of nonlinear equations (i.e., the gradient) for which the zeros need to be found.

To define the elements of `avF[0]`, we will used the same variables defined earlier. The only difference lies in using the reference to the item parameters of item j, rather than the values of the item parameters per se.

```
TwoPL_NLE(const avF, const vP){ //vP is <aj;bj>
decl ex=exp(1.7*vP[0].*(ones(N,1)*A'-vP[1]));
decl Prob=ex./(1+ex);
decl Xj=X[][j]*ones(1,Q);
decl R=Xj-Prob;
decl D=ones(N,1)*A'-vP[1]*ones(N,Q);
avF[0]=sumr(sumc(1.7*P.*R.*D))|sumr(sumc(-1.7*vP[0]*P.*R));
return 1;
}//end TwoPL_NLE
```

The function `update_post` is the same function in `2PL_MMLE_I`.

The `iterate` function for this program is largely the same as the previous `iterate` functions. The major difference is in the use of `SolveNLE`, a zero-finder function. As input, it takes the function where the system of equations is defined and the address of the initial values for the parameters of item j. It returns the updated estimates to the same address.

```
iterate(){
decl diff=0.1;
decl it=0;
while(diff>MaxDiff && it<MaxIt){
update_post();
decl Par1=zeros(J,2);
for (j=0;j<J;j++){
decl vP=Par0[j][]';
SolveNLE(TwoPL_NLE,&vP);
Par1[j][]=vP';
}
diff=max(sqrt((Par1-Par0).^2));
Par0=Par1;
it++;
} //end while
} //end iterate
```

To estimate standard errors, we need the function `TwoPL_DL_obj`, `fullLike` and `findSE_DL` (not shown here). All of them will be explained later.

The structure of the `main` function of this program is identical to that of `2PL_MMLE_I`.

```
The following results are based on MMLE_II algorithm

      a           b        DL SE(a)     DL SE(b)
 1.8298     -0.96470      0.12650     0.033550
 1.4379     -0.71420     0.083727     0.032176
 1.3799     -0.54251     0.076882     0.030212
 1.2632     -0.39560     0.068563     0.030135
 1.3484     -0.22642     0.072029     0.027664
 1.7499    -0.027444     0.098138     0.023397
 1.7306     0.053287     0.096376     0.023590
0.77912      0.11880     0.044347     0.039954
 1.2035      0.33102     0.063779     0.030675
 1.2760      0.37469     0.068003     0.029953
 1.1632      0.71873     0.064365     0.036824
 1.6739      0.87003      0.10232     0.032290
 1.5440       1.0198     0.096554     0.037094
0.79812       1.1151     0.049487     0.060885
 1.7701       1.7574      0.17177     0.064979
```

The EM Algorithm

It is possible to bypass the calculation of (some of the) derivatives when estimating item parameters. One technique is the EM Algorithm. Generally speaking, the EM is a data-augmentation algorithm that augments the observed data with latent data to simplify calculations, and make the subsequent maximization easier.

In traditional maximum likelihood estimation, the estimate is obtained

$$\hat{\boldsymbol{\xi}} = \arg\max_{\boldsymbol{\xi}} L(\mathbf{X}|\boldsymbol{\xi}) = \arg\max_{\boldsymbol{\xi}} l(\mathbf{X}|\boldsymbol{\xi})$$

In latent variable modeling context, this is can be difficult to do.

To simplify the problem, $\mathbf{X}$ is treated as *incomplete* data, and $\mathbf{Z} = (\mathbf{X}, \boldsymbol{\theta})$ as the *complete* data, where $\mathbf{X}$ is observed, whereas $\boldsymbol{\theta}$ is missing. We assume that $\mathbf{X}$ and $\boldsymbol{\theta}$ are jointly distributed as:

$$p(\mathbf{Z}|\boldsymbol{\xi}) = p(\mathbf{X}, \boldsymbol{\theta}|\boldsymbol{\xi}) = p(\boldsymbol{\theta}|\mathbf{X}, \boldsymbol{\xi})p(\mathbf{X}|\boldsymbol{\xi}).$$

$L(\boldsymbol{\xi}|\mathbf{Z}) = L(\boldsymbol{\xi}|\mathbf{X}, \boldsymbol{\theta}) = p(\mathbf{X}, \boldsymbol{\theta}|\boldsymbol{\xi})$, defined as the *complete-data* likelihood function, is considered a random variable because $\boldsymbol{\theta}$ is *missing*, and assumed to be a random variable.

In contrast, we call $L(\boldsymbol{\xi}|\mathbf{X})$ as the *incomplete-data* likelihood function.

In the EM algorithm, the E-step involves finding the expected value of the complete data log-likelihood with respect to $\boldsymbol{\theta}$ given the observed data $\mathbf{X}$ and the current estimate of $\boldsymbol{\xi}$, denoted by $\boldsymbol{\xi}^t$. Specifically, this expectation is defined as:

$$\begin{aligned}
Q(\boldsymbol{\xi}, \boldsymbol{\xi}^t) &= E\left[l(\mathbf{X}, \boldsymbol{\theta}|\boldsymbol{\xi}^t)\right] \\
&= \int_{\boldsymbol{\theta}} \log p(\mathbf{X}, \boldsymbol{\theta}|\boldsymbol{\xi}) \times p(\boldsymbol{\theta}|\mathbf{X}, \boldsymbol{\xi}^t) d\boldsymbol{\theta} \\
&= \int_{\boldsymbol{\theta}} \sum_i \log p(\boldsymbol{X}_i, \boldsymbol{\theta}|\boldsymbol{\xi}) \times p(\boldsymbol{\theta}|\mathbf{X}, \boldsymbol{\xi}^t) d\boldsymbol{\theta} \\
&= \sum_i \int_{\boldsymbol{\theta}} \log p(\boldsymbol{X}_i, \boldsymbol{\theta}|\boldsymbol{\xi}) \times p(\boldsymbol{\theta}|\mathbf{X}, \boldsymbol{\xi}^t) d\boldsymbol{\theta} \\
&= \sum_i \int_{\theta} \log p(\boldsymbol{X}_i, \theta|\boldsymbol{\xi}) \times p(\theta|\boldsymbol{X}_i, \boldsymbol{\xi}^t) d\theta
\end{aligned}$$

where $p(\theta|\boldsymbol{X}_i, \boldsymbol{\xi}^t)$ is the posterior distribution of $\boldsymbol{\theta}$ given the response vector $\boldsymbol{X}_i$ and the most current estimates of the item parameters. We can approximate this expectation using quadrature nodes as follows:

$$Q(\boldsymbol{\xi}, \boldsymbol{\xi}^t) \approx \sum_i \sum_q \log p(\boldsymbol{X}_i, A_q|\boldsymbol{\xi}) \times p(A_q|\boldsymbol{X}_i, \boldsymbol{\xi}^t).$$

Using our previous notation, we can rewrite this as

$$Q(\boldsymbol{\xi}, \boldsymbol{\xi}^t) \approx \sum_{i=1}^{N} \sum_{q=1}^{Q} \log L(\boldsymbol{X}_i|A_q) \times P(A_q|\boldsymbol{X}_i).$$

The M-step involves maximizing the expectation to find

$$\boldsymbol{\xi}^{t+1} = \arg\max_{\boldsymbol{\xi}} Q(\boldsymbol{\xi}, \boldsymbol{\xi}^t).$$

This we can do one item at a time. To implement the M-step, first find:

$$\begin{aligned}
\frac{\partial Q(\boldsymbol{\xi}, \boldsymbol{\xi}^t)}{\partial \boldsymbol{\xi}_j} &= \sum_{i=1}^{N}\sum_{q=1}^{Q} P(A_q|\boldsymbol{X}_i)\frac{\partial \log L(\boldsymbol{X}_i|A_q)}{\partial \boldsymbol{\xi}_j} \\
&= \sum_{i=1}^{N}\sum_{q=1}^{Q} P(A_q|\boldsymbol{X}_i)\frac{1}{L(\boldsymbol{X}_i|A_q)} \times \frac{\partial L(\boldsymbol{X}_i|A_q)}{\partial \boldsymbol{\xi}_j} \\
&= \sum_{i=1}^{N}\sum_{q=1}^{Q} P(A_q|\boldsymbol{X}_i)\frac{1}{L(\boldsymbol{X}_i|A_q)} \times L(\boldsymbol{X}_i|A_q)\left[\frac{X_{ij} - P_{jq}}{P_{jq}(1-P_{jq})}\right]\frac{\partial P_{jq}}{\partial \boldsymbol{\xi}_j} \\
&= \sum_{i=1}^{N}\sum_{q=1}^{Q} P(A_q|\boldsymbol{X}_i)\left[\frac{X_{ij} - P_{jq}}{P_{jq}(1-P_{jq})}\right]\frac{\partial P_{jq}}{\partial \boldsymbol{\xi}_j}
\end{aligned}$$

Note that this derivative is equal to the derivative given in the last equation on page 68, which indicates that the MMLE results can be obtained via EM algorithm.

The remaining steps of the EM algorithm follows those of MMLE. For example, we can use numerical algorithm to obtain $\boldsymbol{\xi}^{t+1}$.

The Bock-Aitken Solution

Bock and Aitken (1981) showed that the expected marginalized log-likelihood given the most recent item parameter estimates can be written as

$$E\left[l(\mathbf{X})|\boldsymbol{\xi}^t\right] = \sum_{q=1}^{Q}\sum_{j=1}^{J} R_{jq}\log P_{jq} + (N_q - R_{jq}) \times \log(1 - P_{jq}) + constant,$$

where (again) R_{jq} is the expected number of correct response on item j for quadrature node A_q and N_q is the expected number of examinees in quadrature node A_q based on the posterior distribution.

With respect to $\boldsymbol{\xi}_j$, the objective function based on the above equation can be written as

$$O_{EM} = \sum_{q=1}^{Q}\left[R_{jq} \times \log P_{jq} + (N_q - R_{jq}) \times \log(1 - P_{jq})\right].$$

To estimate α_j and β_j, we need to find

$$\begin{pmatrix} \frac{\partial O_{EM}}{\partial \alpha_j} \\ \frac{\partial O_{EM}}{\partial \beta_j} \end{pmatrix} = \mathbf{0}.$$

$$\begin{aligned}
\frac{\partial O_{EM}}{\partial \alpha_j} &= \sum_{q=1}^{Q}\left[\frac{R_{jq}}{P_{jq}} \times \frac{\partial P_{jq}}{\partial \alpha_j} - \frac{N_q - R_{jq}}{1 - P_{jq}} \times \frac{\partial P_{jq}}{\partial \alpha_j}\right] \\
&= \sum_{q=1}^{Q}\left[\frac{R_{jq}}{P_{jq}} - \frac{N_q - R_{jq}}{1 - P_{jq}}\right] \times \frac{\partial P_{jq}}{\partial \alpha_j} \\
&= \sum_{q=1}^{Q}\left[\frac{R_{jq} - P_{jq} \times N_q}{P_{jq} \times (1 - P_{jq})}\right] \times [1.7 \times P_{jq} \times (1 - P_{jq}) \times (A_q - \beta_j)] \\
&= 1.7 \sum_{q=1}^{Q}(A_q - \beta_j) \times [R_{jq} - P_{jq} \times N_q]
\end{aligned}$$

Similarly,

$$\frac{\partial O_{EM}}{\partial \beta_j} = -1.7\alpha_j \sum_{q=1}^{Q}[R_{jq} - P_{jq} \times N_q]$$

Comparing these derivatives to the derivatives we set aside on page 69, we can see that

$$\begin{pmatrix} \frac{\partial l(\mathbf{X})}{\partial \alpha_j} \\ \frac{\partial l(\mathbf{X})}{\partial \beta_j} \end{pmatrix} = \begin{pmatrix} \frac{\partial O_{EM}}{\partial \alpha_j} \\ \frac{\partial O_{EM}}{\partial \beta_j} \end{pmatrix}.$$

This shows that optimizing the $l(\mathbf{X})$ is equivalent to optimizing O_{EM}. To avoid involving any derivatives, we can optimize O_{EM} numerically.

It should be noted that the numerical optimization also involves the Hessian matrix, which is also derived numerically. However, this Hessian matrix cannot be used to compute the standard errors - these standard errors are underestimates because the objective function involves a single item rather than the entire likelihood.

Estimation using Objective Function based on the Bock-Aitken Soluation: `2PL_MMLE_EM`

For this program, in addition to including `oxstd.h`, we also need to import `maxsqp`.

The variables declared globally are the same as those in `2PL_MMLE_II`, except for `Ng`, `Rg` and `SE_EM`. The former two contain N_q and R_{jq}, respectively, and the last contains the standard errors estimated using numerical second derivatives of O_{EM}, which, as stated before, underestimates the standard errors.

The function below, `TwoPL_MMLE_EM`, contains `adFunc[0]`, which holds O_{EM}, the objective function we aim to maximize.

```
TwoPL_MMLE_EM(const vP, const adFunc, const avScore, const amHessian)
{//vP is <aj;bj>
decl ex=exp(1.7*vP[0].*(A'-vP[1]));        //1 x Q
decl Prob=ex./(1+ex);      //1 x Q
adFunc[0] = sumr(Rg[j][].*log(Prob)+(Ng[j][]-Rg[j][]).*log(1-Prob));
return 1;
}//end TwoPL_MMLE_EM
```

To define the elements of `adFunc[0]`, we will used the same variables defined earlier, except `Rg` and `Ng`. Note that `Rg` and `Ng` are $J \times Q$ matrices. `Prob` is a $1 \times Q$ matrix of probabilities and the O_{EM} is calculated for item j.

The function `update_post` is the same as that in `2PL_MMLE_I`.

Based on the posteriors calculated from `update_post`, N_q and R_{jq} are calculated using the function `NgRg` below. Recall that $N_q = \sum_i^N P(A_q|X_i)$ and $R_{jq} = \sum_i^N P(A_q|X_i) \times X_i$.

```
NgRg()
{
Ng=ones(J,1)*sumc(P);      //J x Q
Rg=zeros(J,Q); //J x Q

for (decl j=0;j<J;j++){ //j loop: for each item
decl Xjq=X[][j]*ones(1,Q);   //N x Q
Rg[j][]=sumc(Xjq.*P);   //1 x Q
}//end j
}  //end NgRg
```

```
findSE_MMLE_EM(){
decl  mhess;
for (j=0;j<J;j++){
decl vP=Par0[j][]';
if (Num2Derivative(TwoPL_MMLE_EM,  vP,  &mhess)){
SE_EM[j][]=sqrt(diagonal(-invert(mhess)));
}//end if
}//end j
}//end findSE_MMLE_EM
```

The function `findSE_MMLE_EM` can be used to estimate the standard errors based on the Hessian matrix of O_{EM}. However, this is for pedagogical purposes only to show that the estimates of standard errors are inaccurate when comparing with those obtained in `2PL_MMLE_I`.

The following `iterate` function is largely similar to the previous `iterate` function except two differences. First, in the E step, N_q and R_{jq} are calculated by calling `NgRg` function; and second, `MaxSQPF` is used to maximize O_{EM}. The use of `decl low=<0,-10>';` and `decl hi=<5,10>';` specifies the bounds of the item parameters. That is, $\hat{\alpha}_j \in (0, 5)$ and $\hat{\beta}_j \in (-10, 10)$. The declared `f` stores the value of O_{EM}.

```
iterate(){
decl diff=0.1;
decl it=0;

while(diff>MaxDiff && it<MaxIt){
update_post();
NgRg();
decl Par1=zeros(J,2);
for (j=0;j<J;j++){
decl f;
decl low=<0,-10>';
decl hi=<5,10>';
decl vP=Par0[j][]';
MaxSQPF(TwoPL_MMLE_EM, &vP, &f,0, 1, 0, 0, low, hi);
Par1[j][]=vP';
}
diff=max(sqrt((Par1-Par0).^2));
Par0=Par1;
it++;
} //end while
} //end iterate
```

The other three functions, `TwoPL_DL_obj`, `fullLike`, and `findSE_DL` (not shown here), are the same as those in `2PL_MMLE_II`, and can be used to correctly estimate standard errors, and will be explained later.

```
        The following results are based on EM algorithm

     a          b        DL SE(a)     DL SE(b)     EM SE(a)     EM SE(b)
1.8299    -0.96467      0.12650     0.033548     0.094765     0.028293
1.4379    -0.71418     0.083729     0.032175     0.068832     0.029378
1.3800    -0.54247     0.076880     0.030210     0.064348     0.028219
1.2633    -0.39563     0.068569     0.030134     0.058327     0.028672
```

```
 1.3485    -0.22640    0.072031    0.027662    0.061115    0.026457
 1.7501   -0.027453    0.098147    0.023396    0.078708    0.021970
 1.7307    0.053281    0.096382    0.023589    0.077795    0.022177
0.77913     0.11879    0.044348    0.039953    0.040129    0.039298
 1.2035     0.33101    0.063782    0.030674    0.055490    0.029480
 1.2761     0.37464    0.068004    0.029951    0.058627    0.028619
 1.1632     0.71867    0.064362    0.036822    0.056304    0.034710
 1.6740     0.87003     0.10233    0.032292    0.083445    0.028971
 1.5440      1.0197    0.096542    0.037088    0.079434    0.033140
0.79815      1.1151    0.049485    0.060877    0.044878    0.057427
 1.7701      1.7573     0.17177    0.064982     0.12737    0.050932
```

Estimation using only the Likelihood as the Objective function - The Jiffy Discounted Likelihood Technique: `2PL_MMLE_DL.ox`

This is another method for finding parameter estimates without using analytical derivatives. It provides correct item parameter estimates, as well as correct standard errors.

Additionally, it offers a way to use the item parameter estimates to numerically find the correct standard errors (Note that we have not introduced how to obtain the correct standard errors for `2PL_MMLE_II` and `2PL_MMLE_EM`). However, be forewarned: This approach is computationally slow!.

Recall that the objective function in MMLE is

$$l(\mathbf{X}) = \sum_{i=1}^{N} \log \sum_{q=1}^{Q} L(\boldsymbol{X}_i|A_q) \times w(A_q).$$

As it is, it already involves two summations (i.e., across N examinees and across Q quadrature nodes).

Hidden in $L(\boldsymbol{X}_i|A_q)$, the likelihood of the response vector $\boldsymbol{X}_i$ given A_q, is the product across the J items. Thus, this problem involves an $N \times Q \times J$ array, which can be unwieldy to handle.

To be able to implement marginalized maximum likelihood estimation using only the (marginalized) likelihood as objective function, we need to employ a trick. This trick requires us to recognize (i) that we are estimating the parameters one item at a time, and (ii) that

$$L(\boldsymbol{X}_i|A_q) = \left[\prod_{j' \neq j} P_{j'q}^{X_{ij'}}(1-P_{j'q})^{1-X_{ij'}}\right] \times P_{jq}^{X_{ij}}(1-P_{jq})^{1-X_{ij}}.$$

Thus, when working with item j,

$$L^{-j}(\boldsymbol{X}_i|A_q) = \prod_{j' \neq j} P_{j'q}^{X_{ij'}}(1-P_{j'q})^{1-X_{ij'}} = \frac{L(\boldsymbol{X}_i|A_q)}{P_{jq}^{X_{ij}}(1-P_{jq})^{1-X_{ij}}}$$

is considered to be a constant with respect to $\boldsymbol{\xi}_j$.

At iteration t, we can compute $L^{(t-1)}(\boldsymbol{X}_i|A_q)$, the $L(\boldsymbol{X}_i|A_q)$ based on $\boldsymbol{\xi}_1^{(t-1)}, \ldots, \boldsymbol{\xi}_J^{(t-1)}$, and hold this scaler constant across the J items.

At the same time, we also need to discount the impact of item j by dividing $L^{(t-1)}(\boldsymbol{X}_i|A_q)$ by $P_{jq}^{(t-1)X_{ij}} \left(1 - P_{jq}^{(t-1)}\right)^{1-X_{ij}}$, where $P_{jq}^{(t-1)}$ is the probability of success on item j given A_k and $\boldsymbol{\xi}_j^{(t-1)}$.

Hence, at iteration t, the item parameters of item j can be estimated by maximizing the following modified objective function:

$$
\begin{aligned}
l_j(\mathbf{X}) &= \sum_{i=1}^{N} \log \sum_{q=1}^{Q} \frac{L^{(t-1)}(\boldsymbol{X}_i|A_q)}{P_{jq}^{(t-1)X_{ij}} \left(1 - P_{jq}^{(t-1)}\right)^{1-X_{ij}}} \times \left[P_{jq}^{X_{ij}}(1 - P_{jq})^{1-X_{ij}}\right] \times w(A_q) \\
&= \sum_{i=1}^{N} \log \sum_{q=1}^{Q} L^{-j(t-1)}(\boldsymbol{X}_i|A_q) \times \left[P_{jq}^{X_{ij}}(1 - P_{jq})^{1-X_{ij}}\right] \times w(A_q) \\
&= \sum_{i=1}^{N} \log \sum_{q=1}^{Q} \left[P_{jq}^{X_{ij}}(1 - P_{jq})^{1-X_{ij}}\right] \times \frac{P(A_q|\boldsymbol{X}_i)}{P_{jq}^{(t-1)X_{ij}} \left(1 - P_{jq}^{(t-1)}\right)^{1-X_{ij}}}
\end{aligned}
$$

where candidate values for $\boldsymbol{\xi}_j$ affect only P_{jq}.

It is worth noting that all the factors involved in this objective function are scalar (because $L^{(t-1)}(\boldsymbol{X}_i|A_q)$ or $P(A_q|\boldsymbol{X}_i)$ can be computed elsewhere), allowing us to carry our maximization over a two-dimensional array (i.e., an $N \times Q$ matrix).

Using the discounted likelihood technique (DLT), we can obtain the MMLE estimate using only the likelihood. In essence, we can implement MMLE in a *jiffy* (i.e., quickly).

However, it should be noted that the DLT is only quick with respect our ability to implement MMLE, not the actual implementation time.

Below are some of the functions in this program.

```
TwoPL_DL_obj(const vP, const adFunc, const avScore, const amHessian)//vP is <aj;bj>
{
decl ex0=exp(1.7*Par0[j][0]*(ones(N,1)*A'-Par0[j][1])); //N x Q
decl Prob0=ex0./(1+ex0);   //N x Q
decl Xj=X[][j]*ones(1,Q);                          //N X Q
decl lj0=(Prob0.^Xj).*(1-Prob0).^(1-Xj);
decl fL_j=fL./lj0;
decl ex=exp(1.7*vP[0]*(ones(N,1)*A'-vP[1])); //Q x 1
decl Prob=ex./(1+ex);                                //Q x J
adFunc[0] = sumc(log(sumr(fL_j.*(Prob.^Xj).*((1-Prob).^(1-Xj)).*(ones(N,1)*W'))));
return 1;
}//end TwoPL_DL
```

The function `TwoPL_DL_obj` contains the objective function to be maximized.

The variable `lj0` is the likelihood for item j than needs to be discounted from the full likelihood `fL`. The "discounted likelihood" is given by `fL_j`.

Note that two types of probabilities are computed for item j: one based on parameter estimates from the the previous iteration, and another based on values to be estimated at this iteration.

The objective function in `adFunc[0]` is an implementation of $l_j(\mathbf{X})$.

```
fullLike(){
fL=<>;                                                          //empty -> N x Q
for(decl i=0;i<N;i++){
decl ex=exp(1.7*(ones(Q,1)*Par0[][0]').*(A*ones(1,J)-ones(Q,1)*Par0[][1]'));//Q x J
decl Prob=ex./(1+ex);                                                  //Q x J
decl Xi=ones(Q,1)*X[i][];                                              //Q x J
decl tmp=sumr(Xi.*log(Prob)+(1-Xi).*log(1-Prob));
fL=fL|exp(tmp');
}//end i
}//end fullLike
```

The function `fullLike` finds `fL`. Note that it loops across the N examinees, which represents the N rows. `Prob` is a $Q \times J$ matrix of probabilities, but reduces to $Q \times 1$ after taking the row sums. By exponentiating, row sum is equivalent to row product; and by transposing, we get a $1 \times Q$ of likelihoods for each examinee.

The function `findSE_DL` estimates the standard errors using the numerical second derivatives of the "discounted likelihood".

```
findSE_DL(){
fullLike();
decl  mhess;
for (j=0;j<J;j++){
decl vP=Par0[j][]';
if (Num2Derivative(TwoPL_DL_obj,  vP,  &mhess)){
SE[j][]=sqrt(diagonal(-invert(mhess)));
}//end if
}//end j
}//end findSE_DL
```

Again, `iterate` for this program is highly similar to the previous `iterate` functions.

```
iterate(){
decl diff=0.1;
decl it=0;
while(diff>MaxDiff && it<MaxIt){
fullLike();
decl Par1=zeros(J,2);
for (j=0;j<J;j++){
decl f;
decl vP=Par0[j][]';
decl low=<0,-10>';
decl hi=<5,10>';
MaxSQPF(TwoPL_DL_obj, &vP, &f,0, 1, 0, 0, low, hi);
Par1[j][]=vP';
}
diff=max(sqrt((Par1-Par0).^2));
Par0=Par1;
it++;
} //end while
} //end iterate
```

Lastly, the `main` function has the same structure as the previous `main` functions.

```
The following results are based on MMLE_DL algorithm

        a          b       DL SE(a)      DL SE(b)
   1.8305    -0.96407      0.12653     0.033533
   1.4384    -0.71361     0.083746     0.032160
   1.3804    -0.54208     0.076912     0.030202
   1.2637    -0.39517     0.068585     0.030124
   1.3489    -0.22608     0.072056     0.027655
   1.7506   -0.027098     0.098183     0.023389
   1.7313    0.053604     0.096420     0.023583
  0.77936     0.11895     0.044360     0.039941
   1.2039     0.33117     0.063802     0.030665
   1.2765     0.37484     0.068028     0.029943
   1.1635     0.71876     0.064384     0.036812
   1.6744     0.87002      0.10235     0.032281
   1.5443      1.0197     0.096568     0.037082
  0.79836      1.1150     0.049498     0.060860
   1.7703      1.7573      0.17177     0.064969
```

Numerical Calculation of the Standard Error using

The DLT finds an estimate of the Hessian matrix in the process of estimating the item parameters. As a result, the DLT finds the item parameters and the standard errors in one step, without requiring any derivatives.

At the same time, the DLT can also be used to estimate standard errors using estimated item parameters. Particularly, it can be incoporated in `2PL_MMLE_II` and `2PL_MMLE_EM`.

As alluded to earlier, the functions `TwoPL_DL_obj`, `fullLike` and `findSE_DL` were already used in `2PL_MMLE_II` and `2PL_MMLE_EM` to estimate standard errors.

When using the DLT in conjunction with the estimated item parameters, the estimates are obtained by solving a system of equations using `SolveNLE` in `2PL_MMLE_II` or maximizing O_{EM} in `2PL_MMLE_EM`.

After which, the `findSE_DL` is called, where the Hessian of the "discounted likelihood" is evaluated at the estimates. With the Hessian matrix, the standard errors can be estimated.

1. Run the four programs (MMLE_I, MMLE_II, MMLE_EM, MMLE_DL) and compare the different item parameter estimates and standard errors.

2. Note if there are discernable differences in the computational times.

3. What happens when you increase the sample size N?

Additional Exercises: Estimating the 1PL

1. To derive the MMLE algorithm for 1PLM, we may start by finding the appropriate derivatives. Fill in the missing steps.

Recall that the item response function for 1PLM is given by

$$P(X_{ij} = 1|\theta_i, \beta_j) = \frac{\exp(1.7 \times [\theta_i - \beta_j)]}{1 + \exp[1.7 \times (\theta_i - \beta_j)]}.$$

Then, the marginalized likelihood is given as

$$\begin{aligned} L(\boldsymbol{X}_i) &= \int_\theta L(\boldsymbol{X}_i|\theta) \times g(\theta) d\theta \\ &\approx \sum_{q=1}^{Q} L(\boldsymbol{X}_i|A_q) \times w(A_q). \end{aligned}$$

For N individuals, the marginalized loglikelihood can be written as

$$\begin{aligned} l(\boldsymbol{X}) &= \log \prod_{i=1}^{N} L(\boldsymbol{X}_i) \\ &= \sum_{i=1}^{N} \log L(\boldsymbol{X}_i). \end{aligned}$$

1 The first derivative of the marginalized loglikelihood is

$$\begin{aligned}\frac{\partial l(\boldsymbol{X})}{\partial \beta_j} &= \frac{\partial\left[\sum_{i=1}^{N}\log L(\boldsymbol{X}_i)\right]}{\partial \beta_j} \\ &= (\quad)\times\frac{\partial l(\boldsymbol{X}_i)}{\partial \beta_j},\end{aligned}$$

2 where

$$\begin{aligned}\frac{\partial l(\boldsymbol{X}_i)}{\partial \beta_j} &= \frac{\partial\left[\sum_{q=1}^{Q} L(\boldsymbol{X}_i|A_q)\right]}{\partial \beta_j} \\ &= \sum_{q=1}^{Q} w(A_q)\times\frac{\partial\left[L(\boldsymbol{X}_i|A_q)\right]}{\partial \beta_j},\end{aligned}$$

3 and

$$\begin{aligned}\frac{\partial L(\boldsymbol{X}_i|A_q)}{\partial \beta_j} &= \frac{\partial}{\partial \beta_j}\prod_{j=1}^{J}(\quad) \\ &= \prod_{j'\neq j}^{J} P_{j'q}^{X_{ij}}\times(1-P_{j'q})^{(1-X_{ij})}\times\frac{\partial(\quad)}{\partial \beta_j} \\ &= \prod_{j=1}^{J} P_{jq}^{X_{ij}}\times(1-P_{jq})^{(1-X_{ij})}\times\frac{\partial P_{jq}}{\partial \beta_j}\times(\quad) \\ &= \prod_{j=1}^{J}\frac{\partial P_{jq}}{\partial \beta_j}\times(\quad)\end{aligned}$$

4 Finally,

$$\begin{aligned}\frac{\partial l(\boldsymbol{X})}{\partial \beta_j} &= \sum_{i=1}^{N}\frac{1}{L(\boldsymbol{X}_i)}\sum_{q=1}^{Q} w(A_q)\times L(\boldsymbol{X}_i|A_q)\times\frac{X_{ij}-P_{jq}}{P_{jq}(1-P_{jq})}\times\frac{\partial P_{jq}}{\partial \beta_j} \\ &= (\quad) \\ &= -1.7\sum_{i=1}^{N}\sum_{q=1}^{Q} P(A_q|\boldsymbol{X}_i)\times(X_{ij}-P_{jq}).\end{aligned}$$

Show that the second derivative is

$$\frac{\partial^2 l^*(\boldsymbol{X})}{\partial \beta_j^2} = -1.7^2\sum_{i=1}^{N}\sum_{q=1}^{Q} P(A_q|\boldsymbol{X}_i)\times P_{jq}\times(1-P_{jq}).$$

The above second derivative is for the purpose of estimating β_j. To obtain $SE(\beta_j)$, we need to find

$$\begin{aligned}
\frac{\partial^2 l(\boldsymbol{X})}{\partial \beta_j^2} &= -\sum_{i=1}^{N}\left\{\sum_{q=1}^{Q} p(A_q|\boldsymbol{X}_i) \times \left[\frac{X_{ij}-P_{jq}}{P_{jq}(1-P_{jq})}\right] \times \frac{\partial P_{jq}}{\partial \beta_j}\right\}^2 \\
&= -\sum_{i=1}^{N}\left\{\sum_{q=1}^{Q} p(A_q|\boldsymbol{X}_i) \times \left[\frac{X_{ij}-P_{jq}}{P_{jq}(1-P_{jq})}\right] \times (\quad)\right\}^2 \\
&= (\quad)\sum_{i=1}^{N}\left\{\sum_{q=1}^{Q} p(A_q|\boldsymbol{X}_i) \times (\quad)\right\}^2
\end{aligned}$$

2. Based on these derivations, write four programs that estimate the item parameters of the 1PL model using

(a) a complete MMLE algorithm that involves the gradient and Hessian

(b) an MMLE algorithm that involves the gradient only, and the standard errors that are determined *analytically* (same as in (a) above);

(c) an EM algorithm based on the Bock and Aitken solution; and

(d) optimization based on the original objective function only.

3. Write also two program that computes the standard errors *analytically* and *numerically*.

Generate the data using the same program in the 2PL example.

4. Compare the estimates, standard errors, and speed of the four programs for different values of N and J.

Optional Exercises: Numerical Second Derivatives

Let's start with an example involving a single variable, and the first derivative. Let $f(x) = 3x^2 - x + 5$. The first derivative of this function is $f'(x) = 6x - 1$. The slope at $x = 2$ is $f'(2) = 6(2) - 1 = 11$.

The second derivative is $f''(x) = 6$, which is a constant.

We can numerically by examining two points in the vicinity of $(2, f(2)) = (2, 15)$. This could be the points $(x - h, f(x - h))$ and $(x + h, f(x + h))$, where h is a small number, say, 0.0001.

The slope (i.e., first derivative) of f at x can be approximated by

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h},$$

which is simply the $\triangle f(x)/\triangle x$, or the familiar *rise over run*.

Applying the first derivative approximation to our example, we get

$$f'(2) \approx \frac{f(2.0001) - f(1.9999)}{2 \times 0.0001} = 11.00000000002,$$

which is close enough to 11 for government work.

Now, obtaining the second derivatives numerically requires applying the same principle, and recognizing that f'' is just the derivative of f'. That is, the second derivative is just the slope of the slopes (i.e., $\triangle f'(x)/\triangle x$).

For this we need three points: $(x, f(x))$, $(x-h, f(x-h))$ and $(x+h, f(x+h))$.

The second derivative of f at x can be approximated by

$$\begin{aligned} f''(x) &\approx \frac{\frac{f(x+h)-f(x)}{h} - \frac{f(x)-f(x-h)}{h}}{h} \\ &= \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}. \end{aligned}$$

Applying the second derivative approximation to our example, we get

$$f''(2) \approx \frac{f(2.0001) + f(1.9999) - 2f(2.0000)}{0.0001^2} = 6.0000005,$$

which, again, is close enough to 6 for government work.

We can apply these approximations to multiple variables, but let's limit our discussion to two variables.

The approximations are

$$\begin{aligned} \frac{\partial^2 f}{\partial x^2} &\approx [f(x+h, y) - 2f(x, y) + f(x-h, y)]/h^2 \\ \frac{\partial^2 f}{\partial y^2} &\approx [f(x, y+h) - 2f(x, y) + f(x, y-h)]/h^2 \\ \frac{\partial^2 f}{\partial x \partial y} &\approx [f(x-h, y-h) + f(x+h, y+h) - f(x-h, y+h) - f(x+h, y-h)]/(4h^2) \end{aligned}$$

Let $f(x, y) = x^3 + 2y^2 - 5xy^2$ be the function, and we want to find the second derivatives at the point (1,2).

Analytically,

$$\begin{aligned} \frac{\partial^2 f}{\partial x^2} &= 6x \to 6 \\ \frac{\partial^2 f}{\partial y^2} &= 4 - 10x \to -6 \\ \frac{\partial^2 f}{\partial x \partial y} &= -10y \to -20 \end{aligned}$$

Numerically,

$$\begin{aligned}
\frac{\partial^2 f}{\partial x^2} &\approx [f(1.0001, 2) - 2f(1, 2) + f(0.9999, 2)] / 0.0001^2 \to 5.9999998 \\
\frac{\partial^2 f}{\partial y^2} &\approx [f(1, 2.0001) - 2f(1, 2) + f(1, 1.9999)] / 0.0001^2 \to -6.0000005 \\
\frac{\partial^2 f}{\partial x \partial y} &\approx [f(0.9999, 1.9999) + f(1.0001, 2.0001) - f(0.9999, 2.0001) - f(1.0001, 1.9999)] / (4 \times 0.0001^2) \\
&\to -19.9999999
\end{aligned}$$

1. Write a function that numerically computes the second derivatives of a two-variable objective function.

2. Compare the approximations from this function to those of `Num2Derivative`.

3. Modify one of the estimation programs for the 2PL so that it derives the standard errors using your function for the numerical second derivatives.

The Expectation-Maximization Algorithm - Part II

EM Algorithms for Cognitive Diagnosis Models

The EM algorithms for CDMs are straightforward extensions of the EM algorithms for traditional IRT models, at least when the attribute structure has a saturated formulation. However, we need to note a few differences when CDMs are involved. First, using quadrature nodes to approximate continuous θ is no longer necessary - $\boldsymbol{\alpha}$ is already discrete. And second, for a class of CDMs (e.g., DINA, DINO, G-DINA), estimates of the item parameters can be obtained separately from a partitioning of the attribute vectors.

An EM Algorithm the G-DINA Model

Again, let's start with the marginalized likelihood of the response vector of examinee i, but this time involving CDMs:

$$L(\boldsymbol{X}_i) = \sum_{l=1}^{L} L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l) \times p(\boldsymbol{\alpha}_l),$$

where $p(\boldsymbol{\alpha}_l)$ is the prior probability of the attribute vector $\boldsymbol{\alpha}_l$, and $L = 2^K$.

For a dichotomous response,

$$L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l) = \prod_{j=1}^{J} P_{\boldsymbol{\alpha}^*_{jl}}^{X_{ij}} \times (1 - P_{\boldsymbol{\alpha}^*_{jl}})^{1-X_{ij}},$$

only consider the attributes that are measured in item j: reduced attribute vector

where $P_{\boldsymbol{\alpha}^*_{jl}} = P(X_j = 1|\boldsymbol{\alpha}^*_{jl})$ is the probability of a correct response to item j given the reduced attribute vector $\boldsymbol{\alpha}^*_{jl}$, and $jl = 1, 2, \ldots, 2^{K_j}$, K_j being $\sum_{k=1}^{K} q_{jk}$. Note that for the G-DINA model, the number of parameters for item j is equal to 2^{K_j}.

total number of required attributes in item j

To obtain the MLE of $\mathcal{P}_j$, where $\mathcal{P}_j = P_{\boldsymbol{\alpha}^*_{jl}}$, for $jl = 1, \ldots, 2^{K_j}$, we need to maximize

maximizing likelihood is the same as maximizing the loglikelihood

$$l(\boldsymbol{X}) = \log \prod_{i=1}^{N} L(\boldsymbol{X}_i) = \sum_{i=1}^{N} \log L(\boldsymbol{X}_i)$$

with respect to $\mathcal{P}_j$. As in,

$$\begin{aligned} \frac{\partial l(\boldsymbol{X})}{\partial \mathcal{P}_j} &= \sum_{i=1}^{N} \frac{1}{L(\boldsymbol{X}_i)} \times \frac{\partial L(\boldsymbol{X}_i)}{\partial \mathcal{P}_j} \\ &= \sum_{i=1}^{N} \frac{1}{L(\boldsymbol{X}_i)} \sum_{l=i}^{L} p(\boldsymbol{\alpha}_l) \times \frac{\partial L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l)}{\partial \mathcal{P}_j}. \end{aligned}$$

Now, the derivative on the right-hand side can be shown to be

$$\frac{\partial L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l)}{\partial \mathcal{P}_j} = L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l) \times \left[\frac{X_{ij} - P_{\boldsymbol{\alpha}^*_{jl}}}{P_{\boldsymbol{\alpha}^*_{jl}}(1 - P_{\boldsymbol{\alpha}^*_{jl}})} \right] \times \frac{\partial P_{\boldsymbol{\alpha}^*_{jl}}}{\partial \mathcal{P}_j}.$$

We can use this in $\partial l(\boldsymbol{X})/\partial P_{\boldsymbol{\alpha}^*_{jl}}$ to get

$$\begin{aligned}
\frac{\partial l(\boldsymbol{X})}{\partial \mathcal{P}_j} &= \sum_{i=1}^{N} \frac{1}{L(\boldsymbol{X}_i)} \sum_{l=1}^{L} p(\boldsymbol{\alpha}_l) \times L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l) \times \left[\frac{X_{ij} - P_{\boldsymbol{\alpha}^*_{jl}}}{P_{\boldsymbol{\alpha}^*_{jl}}(1 - P_{\boldsymbol{\alpha}^*_{jl}})} \right] \times \frac{\partial P_{\boldsymbol{\alpha}^*_{jl}}}{\partial \mathcal{P}_j} \\
&= \sum_{i=1}^{N} \sum_{l=1}^{L} \frac{L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l) \times p(\boldsymbol{\alpha}_l)}{L(\boldsymbol{X}_i)} \times \left[\frac{X_{ij} - P_{\boldsymbol{\alpha}^*_{jl}}}{P_{\boldsymbol{\alpha}^*_{jl}}(1 - P_{\boldsymbol{\alpha}^*_{jl}})} \right] \times \frac{\partial P_{\boldsymbol{\alpha}^*_{jl}}}{\partial \mathcal{P}_j} \\
&= \sum_{i=1}^{N} \sum_{l=1}^{L} \left[\frac{X_{ij} - P_{\boldsymbol{\alpha}^*_{jl}}}{P_{\boldsymbol{\alpha}^*_{jl}}(1 - P_{\boldsymbol{\alpha}^*_{jl}})} \right] \times P(\boldsymbol{\alpha}_l|\boldsymbol{X}_i) \times \frac{\partial P_{\boldsymbol{\alpha}^*_{jl}}}{\partial \mathcal{P}_j},
\end{aligned}$$

where $p(\boldsymbol{\alpha}_l|\boldsymbol{X}_i)$ is the posterior probability that examinee i has the attribute pattern $\boldsymbol{\alpha}_l$. At this juncture, we are dealing with two types of attribute patterns, $\boldsymbol{\alpha}_l$ and $\boldsymbol{\alpha}^*_{jl}$. However, we know that for a subset of the attribute patterns, $\boldsymbol{\alpha}_l \rightarrow \boldsymbol{\alpha}^*_{jl}$. Hence,

$$P(\boldsymbol{\alpha}^*_{jl}|\boldsymbol{X}_i) = \sum_{\forall l \ni \boldsymbol{\alpha}_l \rightarrow \boldsymbol{\alpha}^*_{jl}} P(\boldsymbol{\alpha}_l|\boldsymbol{X}_i).$$

This will lead us to the following simplification:

$$\frac{\partial l(\boldsymbol{X})}{\partial \mathcal{P}_j} = \sum_{i=1}^{N} \sum_{jl=1}^{L_j} \left[\frac{X_{ij} - P_{\boldsymbol{\alpha}^*_{jl}}}{P_{\boldsymbol{\alpha}^*_{jl}}(1 - P_{\boldsymbol{\alpha}^*_{jl}})} \right] \times P(\boldsymbol{\alpha}^*_{jl}|\boldsymbol{X}_i) \times \frac{\partial P_{\boldsymbol{\alpha}^*_{jl}}}{\partial \mathcal{P}_j},$$

where $L_j = 2^{K_j}$. This reduces the number of inside terms from 2^K to just 2^{K_j}.

By interchanging the summations, and making some rearrangements, we get,

$$\begin{aligned}
\frac{\partial l(\boldsymbol{X})}{\partial \mathcal{P}_j} &= \sum_{jl=1}^{L_j} \frac{\partial P_{\boldsymbol{\alpha}^*_{jl}}}{\partial \mathcal{P}_j} \times \left[\frac{1}{P_{\boldsymbol{\alpha}^*_{jl}}(1 - P_{\boldsymbol{\alpha}^*_{jl}})} \right] \sum_{i=1}^{N} (X_{ij} - P_{\boldsymbol{\alpha}^*_{jl}}) \times P(\boldsymbol{\alpha}^*_{jl}|\boldsymbol{X}_i) \\
&= \sum_{jl=1}^{L_j} \frac{\partial P_{\boldsymbol{\alpha}^*_{jl}}}{\partial \mathcal{P}_j} \times \left[\frac{1}{P_{\boldsymbol{\alpha}^*_{jl}}(1 - P_{\boldsymbol{\alpha}^*_{jl}})} \right] \left[\sum_{i=1}^{N} X_{ij} \times P(\boldsymbol{\alpha}^*_{jl}|\boldsymbol{X}_i) - P_{\boldsymbol{\alpha}^*_{jl}} \sum_{i=1}^{N} P(\boldsymbol{\alpha}^*_{jl}|\boldsymbol{X}_i) \right] \\
&= \sum_{jl=1}^{L_j} \frac{\partial P_{\boldsymbol{\alpha}^*_{jl}}}{\partial \mathcal{P}_j} \times \left[\frac{1}{P_{\boldsymbol{\alpha}^*_{jl}}(1 - P_{\boldsymbol{\alpha}^*_{jl}})} \right] \left[R^*_{jl} - P_{\boldsymbol{\alpha}^*_{jl}} N^*_{jl} \right],
\end{aligned}$$

where $N^*_{jl} = \sum_{i=1}^{N} P(\boldsymbol{\alpha}^*_{jl}|\boldsymbol{X}_i)$ is the expected number of examinees with *reduced* attribute pattern $\boldsymbol{\alpha}^*_{jl}$, and $R^*_{jl} = \sum_{i=1}^{N} X_{ij} \times P(\boldsymbol{\alpha}^*_{jl}|\boldsymbol{X}_i)$ is the expected number of examinees with attribute pattern $\boldsymbol{\alpha}^*_{jl}$ answering item j correctly.

By expanding the summation, the derivative can be written as

$$\begin{aligned}
\frac{\partial l(\boldsymbol{X})}{\partial \mathcal{P}_j} &= \frac{\partial P_{\boldsymbol{\alpha}^*_1}}{\partial \mathcal{P}_j} \times \left[\frac{1}{P_{\boldsymbol{\alpha}^*_1}(1 - P_{\boldsymbol{\alpha}^*_1})} \right] \left[R^*_1 - P_{\boldsymbol{\alpha}^*_1} N^*_1 \right] + \frac{\partial P_{\boldsymbol{\alpha}^*_2}}{\partial \mathcal{P}_j} \times \left[\frac{1}{P_{\boldsymbol{\alpha}^*_2}(1 - P_{\boldsymbol{\alpha}^*_2})} \right] \left[R^*_2 - P_{\boldsymbol{\alpha}^*_2} N^*_2 \right] \\
&\quad + \cdots + \frac{\partial P_{\boldsymbol{\alpha}^*_{L_j}}}{\partial \mathcal{P}_j} \times \left[\frac{1}{P_{\boldsymbol{\alpha}^*_{L_j}}(1 - P_{\boldsymbol{\alpha}^*_{L_j}})} \right] \left[R^*_{L_j} - P_{\boldsymbol{\alpha}^*_{L_j}} N^*_{L_j} \right].
\end{aligned}$$

We can now start taking the 2^{K_j} derivatives one at a time. For the jl^{th} derivative, it should be clear that

$$\frac{\partial P_{\boldsymbol{\alpha}^*_{jl'}}}{\partial P_{\boldsymbol{\alpha}^*_{jl}}} = \begin{cases} 0 & \text{if } j' \neq j \\ 1 & \text{if } j' = j \end{cases}.$$

Thus, only one of the 2^{K_j} terms will survive. For $\mathcal{P}_j = P_{\boldsymbol{\alpha}^*_{jl}}$, this can be written as

$$\frac{\partial l(\boldsymbol{X})}{\partial P_{\boldsymbol{\alpha}^*_{jl}}} = \left[\frac{1}{P_{\boldsymbol{\alpha}^*_{jl}}(1 - P_{\boldsymbol{\alpha}^*_{jl}})}\right]\left(R^*_{jl} - P_{\boldsymbol{\alpha}^*_{jl}} N^*_{jl}\right).$$

To find $\hat{P}_{\boldsymbol{\alpha}^*_{jl}}$, we need to find the zero of the derivative. Fortunately, this has a closed-form solution, as in,

$$\hat{P}_{\boldsymbol{\alpha}^*_{jl}} = \frac{R^*_{jl}}{N^*_{jl}}.$$

Because we do not need to use an iterative solution, we also do not need the second derivatives.

It should be noted finding the parameters of item j involves solving the system of equations $\bigtriangledown l(\boldsymbol{X}) = \mathbf{0}$. However, each equation is independent of the other equations so the solution can be derived one equation at a time.

As before:

1. At iteration 0, assign initial values to the item parameters, as in, $P^0_{\boldsymbol{\alpha}^*_1}, \ldots, P^0_{\boldsymbol{\alpha}^*_{L_j}}$.
2. At iteration t, find $(R^{t*}_{jl}, N^{t*}_{jl}), \ldots, (R^{t*}_{L_j}, N^{t*}_{L_j})$ based on $P^{t-1}_{\boldsymbol{\alpha}^*_1}, \ldots, P^{t-1}_{\boldsymbol{\alpha}^*_{L_j}}$.
3. From the expected quantities, find the updated item parameter estimates $P^t_{\boldsymbol{\alpha}^*_1}, \ldots, P^t_{\boldsymbol{\alpha}^*_{L_j}}$.
4. Repeat steps 2 and 3 until convergence.

With $\boldsymbol{\alpha}_l$, it is harder to determine what an appropriate prior should be. Typically, we can start with a uniform prior, but more likely than not, it is not the correct prior. To provide a more accurate prior, we can update the prior distribution using the information available in the posterior distributions of the examinees. We can estimate the prior as

$$p(\boldsymbol{\alpha}_l) = \sum_{i=1}^{N} P(\boldsymbol{\alpha}_l | \boldsymbol{X}_i)/N.$$

In determining the prior distribution using information from the data (i.e., posterior distributions), we employ what is called an *empirical Bayes* method.

Computing the Standard Errors of the G-DINA Model Parameter Estimates

At this point, we have different ways of finding the standard errors of the item parameter estimates: one is to use the information matrix, and the other is to use a numerical solution.

The information matrix of the estimator of $\boldsymbol{\mathcal{P}}_j = \{P_{\boldsymbol{\alpha}^*_{jl}}\}$ is $\boldsymbol{I}(\boldsymbol{\mathcal{P}}_j) = -E\{\partial^2 l(\boldsymbol{X})/\partial \boldsymbol{\mathcal{P}}_j^2\}$.

Again, we can start with

$$\begin{aligned}\frac{\partial^2 l(\boldsymbol{X})}{\partial P_{\boldsymbol{\alpha}^*_{jl}} \partial P_{\boldsymbol{\alpha}^*_{jl'}}} &= -\sum_{i=1}^{N} \frac{\partial l(\boldsymbol{X})}{\partial P_{\boldsymbol{\alpha}^*_{jl}}} \frac{\partial l(\boldsymbol{X})}{\partial P_{\boldsymbol{\alpha}^*_{jl'}}} \\ &= -\sum_{i=1}^{N} \left[\frac{1}{L(\boldsymbol{X_i})} \times \frac{\partial L(\mathbf{X}_i)}{\partial P_{\boldsymbol{\alpha}^*_{jl}}}\right] \times \left[\frac{1}{L(\boldsymbol{X_i})} \times \frac{\partial L(\mathbf{X}_i)}{\partial P_{\boldsymbol{\alpha}^*_{jl'}}}\right].\end{aligned}$$

Now,

$$\begin{aligned}\frac{\partial L(\mathbf{X}_i)}{\partial P_{\boldsymbol{\alpha}^*_{jl}}} &= \frac{\partial \left[\sum_{jl'=1}^{L_j} L(\mathbf{X}_i|P_{\boldsymbol{\alpha}^*_{jl'}}) \times w(P_{\boldsymbol{\alpha}^*_{jl'}})\right]}{\partial P_{\boldsymbol{\alpha}^*_{jl}}} \\ &= \sum_{jl'=1}^{L_j} \frac{\partial L(\mathbf{X}_i|P_{\boldsymbol{\alpha}^*_{jl'}})}{\partial P_{\boldsymbol{\alpha}^*_{jl}}} \times w(P_{\boldsymbol{\alpha}^*_{jl'}}) \\ &= \sum_{jl'=1}^{L_j} \left[L(\mathbf{X}_i|P_{\boldsymbol{\alpha}^*_{jl'}}) \times \frac{X_{ij} - P_{\boldsymbol{\alpha}^*_{jl'}}}{P_{\boldsymbol{\alpha}^*_{jl'}}(1 - P_{\boldsymbol{\alpha}^*_{jl'}})} \times \frac{\partial P_{\boldsymbol{\alpha}^*_{jl'}}}{\partial P_{\boldsymbol{\alpha}^*_{jl}}}\right] \times w(P_{\boldsymbol{\alpha}^*_{jl'}}) \\ &= L(\mathbf{X}_i|P_{\boldsymbol{\alpha}^*_{jl}}) \times \frac{X_{ij} - P_{\boldsymbol{\alpha}^*_{jl}}}{P_{\boldsymbol{\alpha}^*_{jl}}(1 - P_{\boldsymbol{\alpha}^*_{jl}})} \times w(P_{\boldsymbol{\alpha}^*_{jl}})\end{aligned}$$

because only one of the terms will have a nonzero derivative (i.e., the term involving $P_{\boldsymbol{\alpha}^*_{jl'}}$).

Thus,

$$\begin{aligned}\frac{1}{L(\boldsymbol{X_i})} \times \frac{\partial L(\mathbf{X}_i)}{\partial P_{\boldsymbol{\alpha}^*_{jl}}} &= \frac{1}{L(\boldsymbol{X_i})} \times \left[L(\mathbf{X}_i|P_{\boldsymbol{\alpha}^*_{jl}}) \times \frac{X_{ij} - P_{\boldsymbol{\alpha}^*_{jl}}}{P_{\boldsymbol{\alpha}^*_{jl}}(1 - P_{\boldsymbol{\alpha}^*_{jl}})} \times w(P_{\boldsymbol{\alpha}^*_{jl}})\right] \\ &= P(\boldsymbol{\alpha}_{jl}|\boldsymbol{X}_i) \times \frac{X_{ij} - P_{\boldsymbol{\alpha}^*_{jl}}}{P_{\boldsymbol{\alpha}^*_{jl}}(1 - P_{\boldsymbol{\alpha}^*_{jl}})}\end{aligned}$$

Evaluated at the solution, the second derivative of the marginalized loglikelihood for item j with respect to the parameters $P_{\boldsymbol{\alpha}^*_{jl}}$ and $P_{\boldsymbol{\alpha}^*_{jl'}}$ is given by

$$\frac{\partial^2 l(\boldsymbol{X})}{\partial P_{\boldsymbol{\alpha}^*_{jl}} \partial P_{\boldsymbol{\alpha}^*_{jl'}}} = -\sum_{i=1}^{N} \left\{P(\boldsymbol{\alpha}_{jl}|\boldsymbol{X}_i) \times \left[\frac{X_{ij} - \hat{P}_{\boldsymbol{\alpha}^*_{jl}}}{\hat{P}_{\boldsymbol{\alpha}^*_{jl}}(1 - \hat{P}_{\boldsymbol{\alpha}^*_{jl}})}\right]\right\} \times \left\{P(\boldsymbol{\alpha}_{jl'}|\boldsymbol{X}_i) \left[\frac{X_{ij} - \hat{P}_{\boldsymbol{\alpha}^*_{jl'}}}{\hat{P}_{\boldsymbol{\alpha}^*_{jl'}}(1 - \hat{P}_{\boldsymbol{\alpha}^*_{jl'}})}\right]\right\}.$$

The negative of $\boldsymbol{I}^{-1}(\hat{\boldsymbol{\mathcal{P}}}_j)$ provides an approximation of $\text{Cov}(\hat{\boldsymbol{\mathcal{P}}}_j)$, and the root of its diagonal elements represents the $\text{SE}(\hat{\boldsymbol{\mathcal{P}}}_j)$.

Implementation of EM Algorithm for the G-DINA Model in Ox

The program `GDINA_MMLE.ox` will estimate the item parameters of the G-DINA model. It requires both item responses and the Q-matrix as input.

This program includes the following headers: `oxstd.h`, `oxprob.h`, and `oxfloat.h`.

We start by declaring a few global variables. The first set are variables `N`, sample size, `J`, test length, `K`, number of attributes, `data`, item response data file name, `qmatrix`, Q-matrix filename, `crit`, maximum tolerable difference between iterations, and `maxit`, maximum number of iterations, are variables that users can define.

The remaining global variables are the $J \times K$ Q-matrix `Q`, $N \times J$ data matrix `X`, $N \times 2^K$ posterior distributions `post`, $2^K \times K$ matrix all possible attribute patterns `alpha`, $J \times 2^{K_j+1}$ matrix containing the estimates and standard errors `est`, $2^K \times 1$ vector of log-prior probabilities `lprior`, and $2^K \times J$ matrix latent group memberships `eta`.

```
decl N=1000,J=30,K=5;
decl data={"sample.dat"};
decl qmatrix={"q_sample.txt"};
decl crit=0.001,maxit=999;
decl Q,X;
decl post,alpha,est;
decl lprior,eta;
```

The function `pattern` requires a number $\mathcal{K}$ as input, and returns a $2^{\mathcal{K}} \times \mathcal{K}$ matrix containing the $2^{\mathcal{K}}$ unique attribute binary patterns. Recall return matrix is designed such that the patterns are increasing in terms of number of 1s first, and then decreasing in terms of numerical value. This can function will be called to generate `alpha` (requires K), and all the possible latent groups for each item (requires K_j).

```
pattern(const KK){
decl patt=<>;
for(decl k=1;k<=KK;k++){//creating all possible patterns
decl tmp=<>;
decl base=zeros(2^(KK-k),1)|ones(2^(KK-k),1);
for(decl l=0;l<2^(k-1);l++){
tmp=tmp|base;}//end l
patt=patt~tmp;}//end k
patt=patt~sumr(patt);
decl tmp=zeros(2^KK,1);
for(decl k=0;k<KK;k++){
tmp=tmp-10^(K-k-1)*patt[][k];}// end k
patt=patt~tmp;
//order the patterns by column KK than by column KK+1
patt=sortbyc(patt,KK~(KK+1));
return(patt[][0:KK-1]);
}//end pattern
```

The function `alphaeta` creates the $2^K \times J$ matrix `eta`, where the lj^{th} entry indicates which latent group $\boldsymbol{\alpha}_l$ will be classified with respect to item j. For example, if item 5 requires α_1 and α_3 of $K = 5$

attributes, then the attribute patterns 00001, 11000, 00111, and 10110, will be classified in latent groups 0, 1, 2, and 3, respectively.

```
alphaeta(){
eta=zeros(2^K,J);
for(decl i=0;i<2^K;i++){
for(decl j=0;j<J;j++){
decl Kj=sumr(Q[j][]);
decl Lj=vecindex(Q[j][].==1);
decl Qj=pattern(Kj);
decl tmp_alpha=alpha[i][Lj];
eta[i][j]=vecindex(prodr(ones(2^Kj)*tmp_alpha.==Qj).==1);
}}//end j; end k
}//end alphaeta
```

The function `findpost` updates the posterior distributions one examinee at a time. The first step requires creating a $2^K \times J$ matrix of item parameters based on the locations indicated in `eta`.

```
findpost(){
decl par=zeros(2^K,J);
for(decl cl=0; cl<2^K;cl++){
for(decl j=0;j<J;j++){
par[cl][j]=est[j][eta[cl][j]];//assign item par based on eta
}}//end cl; end j
for(decl i=0;i<N;i++){
decl XX=ones(2^K,1)*X[i][];
decl tmp=sumr(XX.*log(par)+(1-XX).*log(1-par))+lprior;
tmp=exp(tmp);
post[i][]=(tmp./sumc(tmp))';
}}//end i; end findpost
```

The function `initiate` makes sure that the variables have the appropriate starting values and dimensions.

For example, `lprior` is assigned to have the log of a uniform prior; `est` is a $J \times 2^{K_{j(max)}}$ matrix of -1s; and $P(\mathbf{1}) = .8$ whereas $P(\mathbf{0}) = .2$.

This is also where `alpha` and `eta` are created.

```
initiate(){
alpha=pattern(K);
alphaeta();
lprior=ones(2^K,1)./(2^K);
lprior=log(lprior/sumc(lprior));
est=-1*ones(J,2^(maxc(sumr(Q))));
for(decl j=0;j<J;j++){
decl Kj=sumr(Q[j][]);
est[j][0:(2^Kj-1)]=.2;
est[j][2^Kj-1]=.8;
}// end j
post=ones(N,2^K);
findpost();
}//end initiate
```

```
iterate(){
decl est0, tmp;// est0->estimates from prior iteration
decl maxabs=.1,it=0;
decl n1, r1;// expected # of examinees & correct resp
while (maxabs>crit && it<maxit){
est0=est;
for(decl j=0;j<J;j++){//loop for items
decl Kj=sumr(Q[j][]);
decl Lj=vecindex(Q[j][].==1);
decl Qj=pattern(Kj);
decl tmp_alpha=alpha[][Lj];//reduced alpha patterns
for(decl kj=0;kj<2^Kj;kj++){//loop for latent groups
tmp=ones(2^K,1)*Qj[kj][].==tmp_alpha;
tmp=vecindex(prodr(tmp).==1);//index of att. patt. in group kj
tmp=sumr(post[][tmp]);
n1=sumc(tmp);
r1=sumc(X[][j].*tmp);
est[j][kj]=r1/n1;//estimate of P_j(kj)
}}//end kj; end j
maxabs=max(sqrt((est0-est).^2));
it=it+1;
findpost();
lprior=log(meanc(post))';//update prior
}}//end while; end iterate
```

The function `iterate` is where the item parameters are estimated. It starts by declaring and initiating the variables that will be used within the function.

Note that the variable `tmp` takes on several meaning within the function.

The `while` loop is in effect as long as the difference in parameter estimates between two iterations remain large AND the maximum number of iterations has not been reached.

The item parameters are estimated one item at a time, and within an item, one latent group at a time.

After an iterations, the variables in the `while` conditions are updated. The posterior distributions of the examinees are updated, and the mean probabilities across examinees serves as the updated prior.

```
info_factor(const j,const tmp_alpha,const Qj, const kj){
decl tmp=ones(2^K,1)*Qj[kj][].==tmp_alpha;//Pj
tmp=vecindex(prodr(tmp).==1);
tmp=sumr(post[][tmp]);//prob
decl P=est[j][kj]*ones(N,1);
return(tmp.*(X[][j]-P)./(P.*(1-P)));
}
```

The function `info_factor` computes the factors that will used in computing the information matrix.

```
stderr(){
decl KKj=2^maxc(sumr(Q));
decl est2=ones(J,2^(maxc(sumr(Q))));
findpost();
for(decl j=0;j<J;j++){
decl Kj=sumr(Q[j][]);
decl Lj=vecindex(Q[j][].==1);
decl tmp_se2=zeros(2^Kj,2^Kj);//information matrix-upper triangle only
decl Qj=pattern(Kj);
decl tmp_alpha=alpha[][Lj];//reduced alpha pattern
for(decl kj1=0;  kj1<2^Kj;kj1++){
for(decl kj2=kj1;kj2<2^Kj;kj2++){
decl tmp_est1=info_factor(j,tmp_alpha,Qj,kj1);//finds the first factor
decl tmp_est2=tmp_est1;
if(kj2>kj1) tmp_est2=info_factor(j,tmp_alpha,Qj,kj2);//finds the second factor
tmp_se2[kj1][kj2]=sumc(tmp_est1.*tmp_est2);//sum of cross products across N
}}//end kj1,kj2
tmp_se2=tmp_se2+tmp_se2'-diag(diagonal(tmp_se2));//completing the matrix
est2[j][0:2^Kj-1]=sqrt(diagonal(invert(tmp_se2)));
}//end j
est=est~est2;
}//end stderr
```

The function `stderr` computes standard errors of the item parameter estimates, and updates `est` to include both item parameter estimates and standard errors.

The function creates `est2`, a $J \times 2^{K_{j(max)}}$ matrix of 1s, to hold the standard errors. Prior to computing the standard errors, the posterior distributions of the examinees are updated.

The standard errors are computed one item at a item. For greater efficiency, only the upper triangle of the information matrix, `tmp_se2`, is computed, and `info_factor` is called once for the diagonal entries, and twice for off-diagonal entries.

The information matrix is filled before additional operations are carried out.

Lastly, the standard errors in `est2` are appended to the parameter estimates in `est`.

```
printout(){
decl file0;
file0=fopen("est.out","w");
print("\nParameter estimates and SEs in the file: est.out\n");
fprint(file0,"%#M",est);
fclose(file0);
decl cl=zeros(2^K,1);
for(decl k=0;k<K;k++){
cl=cl+alpha[][k]*10^(K-k-1);}
decl par=sprint("    \"%0",K,".0f\"");
decl postp=meanc(post)';
print("\n","Latent Classes and their Posterior Probabilities:");
print("%cf",{par,"  %.4f   ",par,"  %.4f"},
(cl~postp)[0:(2^K)/2-1][]~(cl~postp)[(2^K)/2:2^K-1][],"\n");
print("Estimates of Attribute Prevalence:");
decl out=(range(1,K)|sumc(alpha.*(meanc(post)'*ones(1,K))))';
print("%cf",{"         %0 .0f","       %.4f "},out);
}//end printout
```

In addition to printing the filename where `est` can be found, the function `printout` also performs additional computations from the posterior distributions of the examinees.

As mentioned earlier, it computes an overall posterior distribution by averaging the posterior distributions of the examinees.

The prevalence of each attribute in the population is estimated by computing the marginal probability from the overall posterior distribution.

Finally, the `main` function reads the data and Q-matrix, and calls the functions `initiate`, `iterate`, `stderr`, and `printout`.

Additional Exercises

1. Modify the program so it can provide the EAP and MAP estimates of each examinee's attribute vector.

2. Modify the program so it can estimate the parameters of the DINA model.

Hints for modifying `GDINA_MMLE.ox`:

To be consistent with the G-DINA notation, instead of g_j and s_j, we will estimate the parameters $g_j = P_j(\not{\mathbf{1}})$ and $1 - s_j = P_j(\mathbf{1})$. We can get the estimate of s_j and its standard error by applying a simple linear transformation, as in, $\hat{s}_j = 1 - \hat{P}_j(\mathbf{1})$.

The function `pattern` is unchanged.

The function `alphaeta` is dramatically simplified. Recall in the DINA generation code, we first use `alpha*Q'` to create a $2^K \times J$ matrix that counts the number of required attributes mastered by each attribute pattern for each item. We then convert the entry in row l column j to 1 if it is equal to K_j; otherwise, it is converted to 0.

For the function `findpost`, we do not need the two loops to find the value of `par` - it is either `est[][1]` if `eta` is 1, and `est[][0]` if `eta` is 0. You should be able to write this in one line by using `eta` and `1-eta`. The loop for the `N` examinees is unchanged.

We only need to change `est` to an $J \times 2$ matrix, with the first column entries set to 0.2, and the second column entries to 0.8. No loop necessary.

In the function `iterate`, the changes are in loop for the items. We do not need K_j, L_j, Q_j, and $\boldsymbol{\alpha}_j^*$ when dealing with the DINA model. The loop for estimating the item parameters by latent group can also be taken out. The number of steps is fixed to two - one to find $\hat{P}_j(\mathbf{1})$, and another $\hat{P}_j(\not{\mathbf{1}})$.

To estimate $\hat{P}_j(\mathbf{1})$, we need to get the indices of the entries of `eta[][j]` that are equal to 1. We the proper index, `r1`, `n1`, and $\hat{P}_j(\mathbf{1})$ is unchanged.

To estimate $\hat{P}_j(\not{\mathbf{1}})$, we can find the indices of the entries of `eta[][j]` that are equal to 0. However this is not necessary. We just need to know that $r_0 + r_1 = \sum_{\forall i} X_{ij}$ and $n_0 + n_1 = N$.

The rest of the steps remain unchanged.

For the function `info_factor`, we only need to know two things: which item is needed, `j`, and which group is involved, `kj`. The remaining argument can be dropped.

For the DINA model, there are only two possible values for `kj`, 0 or 1. The indexing line should involved `eta[][j]` and `kj` to make it more generic. This is the only change for this function.

For the function `stderr`, a few simplifications are needed. One, `est2` and `tmp_se2` have fixed dimensions.

Again, in the `j` loop, we do not need K_j, L_j, Q_j, and $\boldsymbol{\alpha}_j^*$. In addition, the upper limits of the loop is

fixed - this refers to the number of item j parameters.

Next, `tmp_est1` needs to be created from the simplified `info_factor`. We can assign `tmp_est2` to be equal to `tmp_est1`. This assignment is to save time, and is correct when `kj1` and `kj2` are equal. However, when they are not (i.e., `kj1` is greater than `kj2`), we need to call `info_factor` once more.

The remaining steps of `stderr` are unchanged.

The functions `printout` and `main` can be used as they are.

Those who are interested reporting $\hat{s}_j$, what should be $SE[1 - \hat{P}_j(\mathbf{1})]$?

To verify your results, compare the results by running the DINA estimation program you were given last semester.

Optional Exercises

1. DINA Program from Scratch

Warning: This is NOT for the faint of heart!

Instead of modifying the G-DINA estimation program, you may want to write the DINA estimation program from scratch, including the finding the first derivatives and the information matrix.

2. Alternative Ways of Parameter Estimation

a) Because the estimates of the DINA and G-DINA models can be expressed in closed form, we do not need the second derivatives. However, we still the first derivatives to carry out the E-step, and find N^*_{lj} and R^*_{lj}. As an exercise, use `SolveNLE` in the M-step.

b) How about estimating the G-DINA model parameters without any derivatives, as in, use `MaxSQPF`?

c) Still another way of estimating the DINA model is to carry out the G-DINA estimation E-steps, and use `SolveNLE` in the M-step. What would system of equations need to be solved to obtain $\hat{P}_j(\mathbf{1})$, and another $\hat{P}_j(\mathbf{1})$ given N^*_{lj} and R^*_{lj}, $lj = 1, \ldots, 2^{K_j}$.

3. Related to 2b), can we also obtain the standard errors numerically?

The Expectation-Maximization Algorithm - Part III

EM Algorithms for Cognitive Diagnosis Models

Previously, we learned how to estimate the structural parameters of item response models with underlying discrete and continuous latent variables. In this section, we will combine the two models by estimating the structural parameters of higher-order CDMs. In addition to the item parameters (e.g., $P_{\boldsymbol{\alpha}^*_{jl}}$), the structural parameters include the parameters that relate θ_i and $\boldsymbol{\alpha}_i$ (i.e., λ_{0k}, λ_1).

As mentioned previously, the complete specification involved a model for the attribute structure $P(\boldsymbol{\alpha}_l)$, and a model for the item response (i.e., CDM).

Today, we will focus on an EM algorithm where $P(\boldsymbol{\alpha}_l)$ is expressed in terms of a higher-order model (i.e., $P(\boldsymbol{\alpha}_l|\theta)$), and use the DINA model as the CDM. The algorithm can be easily extended for other CDMs.

An EM Algorithm for the Higher-Order DINA Model

Let the marginalized likelihood of the response vector of examinee i be

$$L(\boldsymbol{X}_i) = \int_\theta \sum_{l=1}^{L} L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l) \times P(\boldsymbol{\alpha}_l|\theta) \times p(\theta)\partial\theta,$$

where

$$L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l) = \prod_{j=1}^{J} P_{jl}^{X_{ij}}(1-P_{jl})^{1-X_{ij}},$$

$$P(\boldsymbol{\alpha}_l|\theta) = \prod_{k=1}^{K} P(\alpha_k = 1|\theta)^{\alpha_{lk}}[1-P(\alpha_k=1|\theta)]^{1-\alpha_{lk}},$$

$P_{jl} = P(X_j = 1|\boldsymbol{\alpha}_l)$ is the probability of a correct response to item j given the attribute pattern $\boldsymbol{\alpha}_l$; $P(\boldsymbol{\alpha}_l|\theta)$ and $p(\alpha_k = 1|\theta)$ are the probabilities of the attribute pattern $\boldsymbol{\alpha}_l$ and mastering attribute k conditional on θ, respectively; and $p(\theta)$ is assumed to be $N(0,1)$.

The conditional probability of an attribute mastery given θ is defined as

$$P(\alpha_k = 1|\theta) = \frac{\exp[\lambda_1(\theta - \lambda_{0k})]}{1+\exp[\lambda_1(\theta-\lambda_{0k})]}, \text{ for } k = 1,\ldots,K,$$

where λ_1 is the common attribute discrimination parameter, and λ_{0k} is the location parameter with respect to attribute k.

Because α_{ik} is latent, a simpler higher-order model may be necessary to get stable parameter estimates.

As in the past, we can use quadrature nodes to approximate the marginalized likelihood, as in,

$$L(\boldsymbol{X}_i) \approx \sum_{q=1}^{Q}\sum_{l=1}^{L} L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l) \times P(\boldsymbol{\alpha}_l|A_q) \times W(A_q),$$

where A_q, $W(A_q)$, and Q are the quadrature node, weight of the node, and number of nodes, respectively.

Recall that for the DINA model, given the j^{th} row of the Q-matrix $\boldsymbol{q}_j$,

$$P_{jl} = \begin{cases} g_j & \text{if } \boldsymbol{\alpha}_l'\boldsymbol{q}_j < \boldsymbol{q}_j'\boldsymbol{q}_j \\ 1 - s_j & \text{if } \boldsymbol{\alpha}_l'\boldsymbol{q}_j = \boldsymbol{q}_j'\boldsymbol{q}_j \end{cases}.$$

Let $\boldsymbol{\beta} = \{\beta_p\} = \{\{\boldsymbol{\beta}_{j\eta}\}, \lambda_{01}, \ldots, \lambda_{0K}, \lambda_1\}$, where $\beta_{j0} = g_j$ and $\beta_{j1} = s_j$. For the higher-order DINA model with a common attribute discrimination parameter, the total number of parameters is $(2 \times J) + K + 1$.

To obtain the MLE of β_p, maximize

$$l(\boldsymbol{X}) = \log \prod_{i=1}^{N} L(\boldsymbol{X}_i) = \sum_{i=1}^{N} \log L(\boldsymbol{X}_i)$$

with respect to β_p:

$$\begin{aligned} \frac{\partial l(\boldsymbol{X})}{\partial \beta_p} &= \sum_{i=1}^{N} \frac{\partial L(\boldsymbol{X}_i)}{\partial \beta_p} / L(\boldsymbol{X}_i) \\ &= \sum_{i=1}^{N} \frac{1}{L(\boldsymbol{X}_i)} \sum_{q=1}^{Q} \sum_{l=1}^{L} W(A_q) \frac{\partial L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l) \times P(\boldsymbol{\alpha}_l|A_q)}{\partial \beta_p}. \end{aligned}$$

The derivations need to be carried out one parameter at a time.

Case I: $\beta_p = \beta_{j\eta}$

The derivative in $\partial l(\boldsymbol{X})/\partial \beta_p$ simplifies to

$$\frac{\partial L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l) P(\boldsymbol{\alpha}_l|A_q)}{\partial \beta_{j\eta}} = P(\boldsymbol{\alpha}_l|A_q) \times \frac{\partial L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l)}{\partial \beta_{j\eta}}.$$

As before, it can be shown that the derivative on the right-hand side is equal to

$$\frac{\partial L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l)}{\partial \beta_{j\eta}} = L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l) \times \frac{\partial P_{jl}}{\partial \beta_{j\eta}} \times \left[\frac{X_{ij} - P_{jl}}{P_{jl}(1 - P_{jl})} \right].$$

Thus, the original derivative can be written as

$$\frac{\partial l(\boldsymbol{X})}{\partial \beta_{j\eta}} = \sum_{i=1}^{N} \frac{1}{L(\boldsymbol{X}_i)} \sum_{q=1}^{Q} \sum_{l=1}^{L} W(A_q) \times L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l) \times \frac{\partial P_{jl}}{\partial \beta_{j\eta}} \times \left[\frac{X_{ij} - P_{jl}}{P_{jl}(1 - P_{jl})} \right].$$

By interchanging the summations, we can have the following simplifications:

$$\begin{aligned} \frac{\partial l(\boldsymbol{X})}{\partial \beta_{j\eta}} = \sum_{l=1}^{L} \frac{\partial P_{jl}}{\partial \beta_{j\eta}} \times \left[\frac{1}{P_{jl}(1 - P_{jl})} \right] \times \\ \sum_{i=1}^{N} \left[\frac{L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l) \times \sum_{q=1}^{Q} P(\boldsymbol{\alpha}_l|A_q) \times W(A_q)}{L(\boldsymbol{X}_i)} \right] \times [X_{ij} - P_{jl}] \end{aligned}$$

$$
\begin{aligned}
&= \sum_{l=1}^{L} \frac{\partial P_{jl}}{\partial \beta_{j\eta}} \times \left[\frac{1}{P_{jl}(1-P_{jl})}\right] \sum_{i=1}^{N} \frac{L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l) \times P(\boldsymbol{\alpha}_l)}{L(\boldsymbol{X}_i)} \times [X_{ij} - P_{jl}] \\
&= \sum_{l=1}^{L} \frac{\partial P_{jl}}{\partial \beta_{j\eta}} \times \left[\frac{1}{P_{jl}(1-P_{jl})}\right] \sum_{i=1}^{N} P(\boldsymbol{\alpha}_l|\boldsymbol{X}_i) \times [X_{ij} - P_{jl}] \\
&= \sum_{l=1}^{L} \frac{\partial P_{jl}}{\partial \beta_{j\eta}} \times \left[\frac{1}{P_{jl}(1-P_{jl})}\right] \left[\sum_{i=1}^{N} X_{ij} \times P(\boldsymbol{\alpha}_l|\boldsymbol{X}_i) - P_{jl} \sum_{i=1}^{N} P(\boldsymbol{\alpha}_l|\boldsymbol{X}_i)\right] \\
&= \sum_{l=1}^{L} \frac{\partial P_{jl}}{\partial \beta_{j\eta}} \times \left[\frac{1}{P_{jl}(1-P_{jl})}\right] [R_{jl} - P_{jl} N_l],
\end{aligned}
$$

where $P(\boldsymbol{\alpha}_l|\boldsymbol{X}_i)$ is the posterior probability that examinee i has the attribute pattern $\boldsymbol{\alpha}_l$, $N_l = \sum_{i=1}^{N} P(\boldsymbol{\alpha}_l|\boldsymbol{X}_i)$ is the expected number of examinees with attribute pattern $\boldsymbol{\alpha}_l$, and $R_{jl} = \sum_{i=1}^{N} P(\boldsymbol{\alpha}_l|\boldsymbol{X}_i) X_{ij}$ is the expected number of examinees with attribute pattern $\boldsymbol{\alpha}_l$ answering item j correctly.

It should be noted that,in obtaining $P(\boldsymbol{\alpha}_l) = \sum_{q=1}^{Q} P(\boldsymbol{\alpha}_l|A_q)W(A_q)$, the values of $\lambda_{01}, \ldots, \lambda_{0K}, \lambda_1$, are fixed to some values (i.e., estimates from the previous iteration).

For item j, this derivative can be written such that the summation across the L attribute patterns are partition into two summations: one for $\eta_j = 0$, and one for $\eta_j = 1$. As in,

$$
\begin{aligned}
\frac{\partial l(\boldsymbol{X})}{\partial \beta_{j\eta}} &= \sum_{\{\boldsymbol{\alpha}_l:\, \boldsymbol{\alpha}_l' \boldsymbol{q}_j < \boldsymbol{q}_j' \boldsymbol{q}_j\}} \frac{\partial P_{jl}}{\partial \beta_{j\eta}} \times \left[\frac{1}{P_{jl}(1-P_{jl})}\right] \times [R_{jl} - P_{jl} N_l] + \\
&\quad \sum_{\{\boldsymbol{\alpha}_l:\, \boldsymbol{\alpha}_l' \boldsymbol{q}_j = \boldsymbol{q}_j' \boldsymbol{q}_j\}} \frac{\partial P_{jl}}{\partial \beta_{j\eta}} \times \left[\frac{1}{P_{jl}(1-P_{jl})}\right] \times [R_{jl} - P_{jl} N_l] \\
&= \frac{\partial g_j}{\partial \beta_{j\eta}} \times \left[\frac{1}{g_j[1-g_j]}\right] \sum_{\{\boldsymbol{\alpha}_l:\, \boldsymbol{\alpha}_l' \boldsymbol{q}_j < \boldsymbol{q}_j' \boldsymbol{q}_j\}} [R_{jl} - g_j N_l] + \\
&\quad \frac{\partial (1-s_j)}{\partial \beta_{j\eta}} \times \left[\frac{1}{(1-s_j)s_j}\right] \sum_{\{\boldsymbol{\alpha}_l:\, \boldsymbol{\alpha}_l' \boldsymbol{q}_j = \boldsymbol{q}_j' \boldsymbol{q}_j\}} [R_{jl} - (1-s_j) N_l] \\
&= \frac{\partial g_j}{\partial \beta_{j\eta}} \times \left[\frac{1}{g_j[1-g_j]}\right] \times \left[R_{jl}^{(0)} - g_j N_{jl}^{(0)}\right] + \\
&\quad \frac{\partial (1-s_j)}{\partial \beta_{j\eta}} \times \left[\frac{1}{(1-s_j)s_j}\right] \times \left[R_{jl}^{(1)} - (1-s_j) N_{jl}^{(1)}\right],
\end{aligned}
$$

where $N_{jl}^{(0)}$ is the expected number of examinees lacking at least one of the required attributes for item j, and $R_{jl}^{(0)}$ is the expected number of examinees among $N_{jl}^{(0)}$ correctly answering item j. $N_{jl}^{(1)}$ and $R_{jl}^{(1)}$ have the same interpretation except that they pertain to the examinees with all the required attributes for item j. $N_{jl}^{(0)} + N_{jl}^{(1)}$ is equal to N_l for all j.

When $\eta = 0$ (i.e., $\beta_{j0} = g$), $\partial P_{jl}/\partial \beta_{j\eta}$ is 1 for the first term, and 0 for the second term. Thus, the maximization of $\partial l(\boldsymbol{X})$ with respect to β_{j0} simplifies to solving for g_j in the equation

$$
\left[\frac{1}{g_j(1-g_j)}\right] \times \left[R_{jl}^{(0)} - g_j \times N_{jl}^{(0)}\right] \quad = \quad 0.
$$

This gives the estimator $\hat{g}_j = R_{jl}^{(0)}/N_{jl}^{(0)}$.

Similarly, the maximization of $\partial l(\boldsymbol{X})$ with respect to β_{j1} is equivalent to solving for s_j in the equation

$$-\left[\frac{1}{(1-s_j)s_j}\right] \times \left[R_{jl}^{(1)} - (1-s_j) \times N_{jl}^{(1)}\right] \;=\; 0,$$

which results in the estimator $\hat{s}_j = [N_{jl}^{(1)} - R_{jl}^{(1)}]/N_{jl}^{(1)}$.

Case II: $\beta_p = \lambda_{0k}, k = 1, \ldots, K$

With respect to the higher-order parameters, the derivative in $\partial l(\boldsymbol{X})/\partial\beta_p$ simplifies to

$$\frac{\partial L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l)P(\boldsymbol{\alpha}_l|A_q)}{\partial\lambda_{0k}} = L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l) \times \frac{\partial P(\boldsymbol{\alpha}_l|A_q)}{\partial\lambda_{0k}}.$$

Treating α_k like X_{ij}, the derivative on the right-hand side can be written as

$$\begin{aligned}
\frac{\partial P(\boldsymbol{\alpha}_l|A_q)}{\partial\lambda_{0k}} &= \prod_{k=1}^{K} P(\alpha_k = 1|A_q)^{\alpha_{lk}}[1 - P(\alpha_k = 1|A_q)]^{1-\alpha_{lk}} \\
&\quad \times \frac{\partial P(\alpha_k = 1|A_q)}{\partial\lambda_{0k}} \times \left\{\frac{\alpha_{lk} - P(\alpha_k = 1|A_q)}{P(\alpha_k = 1|A_q) \times [1 - P(\alpha_k = 1|A_q)]}\right\} \\
&= P(\boldsymbol{\alpha}_l|A_q) \times \frac{\partial P(\alpha_k = 1|A_q)}{\partial\lambda_{0k}} \times \left\{\frac{\alpha_{lk} - P(\alpha_k = 1|A_q)}{P(\alpha_k = 1|A_q) \times [1 - P(\alpha_k = 1|A_q)]}\right\} \\
&= P(\boldsymbol{\alpha}_l|A_q) \times (-\lambda_1) \times [\alpha_{lk} - P(\alpha_k = 1|A_q)]\,.
\end{aligned}$$

By substituting and interchanging the summations,

$$\begin{aligned}
\frac{\partial l(\boldsymbol{X})}{\partial\lambda_{0k}} &= \sum_{i=1}^{N} \frac{1}{L(\boldsymbol{X}_i)} \sum_{q=1}^{Q}\sum_{L=1}^{L} W(A_q) \times L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l) \times P(\boldsymbol{\alpha}_l|A_q) \times (-\lambda_1) \times [\alpha_{lk} - P(\alpha_k = 1|A_q)] \\
&= \sum_{q=1}^{Q} -\lambda_1 \sum_{l=1}^{L} [\alpha_{lk} - P(\alpha_k = 1|A_q)] \sum_{i=1}^{N} \frac{L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l) \times P(\boldsymbol{\alpha}_l|A_q) \times W(A_q)}{L(\boldsymbol{X}_i)} \\
&= \sum_{q=1}^{Q} -\lambda_1 \sum_{l=1}^{L} [\alpha_{lk} - P(\alpha_k = 1|A_q)] \sum_{i=1}^{N} P(\boldsymbol{\alpha}_l, A_q|\boldsymbol{X}_i) \\
&= \sum_{q=1}^{Q} -\lambda_1 \sum_{l=1}^{L} [\alpha_{lk} - P(\alpha_k = 1|A_q)] \times N_{lq} \\
&= \sum_{q=1}^{Q} -\lambda_1 \sum_{l=1}^{L} [R_{lkq} - P(\alpha_k = 1|A_q) \times N_{lq}] \\
&= \sum_{q=1}^{Q} -\lambda_1 \times \left[\sum_{l=1}^{L} R_{lkq} - \sum_{l=1}^{L} P(\alpha_k = 1|A_q) \times N_{lq}\right] \\
&= \sum_{q=1}^{Q} -\lambda_1 \times [R_{kq} - P(\alpha_k = 1|A_q) \times N_q] \\
&\propto -\sum_{q=1}^{Q} [R_{kq} - P(\alpha_k = 1|A_q) \times N_q]\,,
\end{aligned}$$

where $P(\boldsymbol{\alpha}_l, A_q|\boldsymbol{X}_i)$ is the joint posterior probability of $\boldsymbol{\alpha}_l$ and A_q for examinee i, N_{lq} is the expected number of examinees with $\boldsymbol{\alpha}_l$ and A_q, R_{lkq} is the expected number of examinees with $\boldsymbol{\alpha}_l$ and A_q who have mastered attribute k, N_q and R_{kq} are the expected number examinees with A_q and expected number of examinees with A_q who have mastered α_k, respectively.

Find $\hat{\lambda}_{0k}$ by solving

$$\sum_{q=1}^{Q} [R_{kq} - P(\alpha_k = 1|A_q) \times N_q] = 0.$$

Case III: $\beta_p = \lambda_1$

As in Case II, the derivative in $\partial l(\boldsymbol{X})/\partial\beta_p$ also simplifies to

$$\frac{\partial L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l)P(\boldsymbol{\alpha}_l|A_q)}{\partial\lambda_1} = L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l)\frac{\partial P(\boldsymbol{\alpha}_l|A_q)}{\partial\lambda_1}.$$

However, it should be noted that α_1 appears across all $P(\alpha_k = 1|A_q)$. For this reason, it might be instructive to go over how the derivative is obtained.

$$\begin{aligned}\frac{\partial P(\boldsymbol{\alpha}_l|A_q)}{\partial\lambda_1} &= \sum_{k=1}^{K}\prod_{k'\neq k} P(\alpha_{k'} = 1|A_q)^{\alpha_{lk'}}[1 - P(\alpha_{k'} = 1|A_q)]^{1-\alpha_{lk'}} \times \\ &\qquad \frac{\partial P(\alpha_k = 1|A_q)^{\alpha_{lk}}[1 - P(\alpha_k = 1|A_q)]^{1-\alpha_{lk}}}{\partial\lambda_1},\end{aligned}$$

and the derivative on the right-hand side is equal to

$$\begin{aligned}&P(\alpha_k = 1|A_q)^{\alpha_{lk}}[1 - P(\alpha_k = 1|A_q)]^{1-\alpha_{lk}}\frac{\partial P(\alpha_k = 1|A_q)}{\partial\lambda_1} \times \\ &\qquad \frac{\alpha_{lk} - P(\alpha_k = 1|A_q)}{P(\alpha_k = 1|A_q)[1 - P(\alpha_k = 1|A_q)]}.\end{aligned}$$

This allows us to write

$$\begin{aligned}\frac{\partial P(\boldsymbol{\alpha}_l|A_q)}{\partial\lambda_1} &= \sum_{k=1}^{K}\left\{\prod_{k=1}^{K} P(\alpha_k = 1|A_q)^{\alpha_{lk}}[1 - P(\alpha_k = 1|A_q)]^{1-\alpha_{lk}}\right\} \\ &\qquad \times \frac{\partial P(\alpha_k = 1|A_q)}{\partial\lambda_1} \times \left\{\frac{\alpha_{lk} - P(\alpha_k = 1|A_q)}{P(\alpha_k = 1|A_q)[1 - P(\alpha_k = 1|A_q)]}\right\} \\ &= \sum_{k=1}^{K} P(\boldsymbol{\alpha}_l|A_q) \times \frac{\partial P(\alpha_k = 1|A_q)}{\partial\lambda_1} \times \left\{\frac{\alpha_{lk} - P(\alpha_k = 1|A_q)}{P(\alpha_k = 1|A_q)[1 - P(\alpha_k = 1|A_q)]}\right\} \\ &= \sum_{k=1}^{K} P(\boldsymbol{\alpha}_l|A_q) \times (A_q - \lambda_{0k}) \times [\alpha_{lk} - P(\alpha_k = 1|A_q)],\end{aligned}$$

because $\partial P(\alpha_k = 1|A_q)/\partial\lambda_1 = (A_q - \lambda_{0k}) \times P(\alpha_k = 1|A_q) \times [1 - P(\alpha_k = 1|A_q)]$.

Again, by substituting and interchanging the summations,

$$\begin{aligned}
\frac{\partial l(\boldsymbol{X})}{\partial \lambda_1} &= \sum_{i=1}^{N} \frac{1}{L(\boldsymbol{X}_i)} \sum_{q=1}^{Q} \sum_{L=1}^{L} W(A_q) L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l) \times \\
&\qquad \times \sum_{k=1}^{K} P(\boldsymbol{\alpha}_l|A_q) \times (A_q - \lambda_{0k}) \times [\alpha_{lk} - P(\alpha_k = 1|A_q)] \\
&= \sum_{q=1}^{Q} \sum_{k=1}^{K} (A_q - \lambda_{0k}) \sum_{l=1}^{L} [\alpha_{lk} - P(\alpha_k = 1|A_q)] \sum_{i=1}^{N} \frac{L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l) \times P(\boldsymbol{\alpha}_l|A_q) \times W(A_q)}{L(\boldsymbol{X}_i)} \\
&= \sum_{q=1}^{Q} \sum_{k=1}^{K} (A_q - \lambda_{0k}) \sum_{l=1}^{L} [\alpha_{lk} - P(\alpha_k = 1|A_q)] \sum_{i=1}^{N} P(\boldsymbol{\alpha}_l, A_q|\boldsymbol{X}_i) \\
&= \sum_{q=1}^{Q} \sum_{k=1}^{K} (A_q - \lambda_{0k}) \sum_{l=1}^{L} [\alpha_{lk} - P(\alpha_k = 1|A_q)] \times N_{lq} \\
&= \sum_{q=1}^{Q} \sum_{k=1}^{K} (A_q - \lambda_{0k}) \sum_{l=1}^{L} [R_{lkq} - P(\alpha_k = 1|A_q) \times N_{lq}] \\
&= \sum_{q=1}^{Q} \sum_{k=1}^{K} (A_q - \lambda_{0k}) \times \left[\sum_{l=1}^{L} R_{lkq} - \sum_{l=1}^{L} P(\alpha_k = 1|A_q) \times N_{lq} \right] \\
&= \sum_{q=1}^{Q} \sum_{k=1}^{K} (A_q - \lambda_{0k}) \times [R_{kq} - P(\alpha_k = 1|A_q) \times N_q] \,.
\end{aligned}$$

Find $\hat{\lambda}_1$ by finding the zero of the above equation.

As previously, to implement the algorithm, Step 1 starts with initial values for $\boldsymbol{g}$, $\boldsymbol{s}$, and $\boldsymbol{\lambda}$. Step 2 involves computing $N_{jl}^{(0)}$, $R_{jl}^{(0)}$, $N_{jl}^{(1)}$and $R_{jl}^{(1)}$, and finding $\hat{\boldsymbol{g}}$ and $\hat{\boldsymbol{s}}$. Step 3 involves computing N_q and R_{kq}, and finding $\hat{\boldsymbol{\lambda}}$. Steps 2 and 3 are repeated until convergence.

We can easily derive the standard errors of the item parameter estimates using the formula for the information matrix in the previous lecture. We will derive the standard errors of the higher-order parameters numerically.

Implementation of EM Algorithm for the HO-DINA model in Ox

`HODINA_MMLE.ox` is an implementation of EM algorithm for the HO-DINA model. As with other programs for CDMs, this program will require the file names of the data and the Q-matrix.

We start by including a few headers and importing the code `solvenle`.

```
#include <oxstd.h>
#include <oxfloat.h>
#import <solvenle>
```

The first set of global variables allows users to easily modify the program to suit their specific needs.

```
decl N=2000,J=30,K=5;
decl data={"X_hodina.dat"};
decl qmatrix={"Q.txt"};
decl crit=0.001,maxit=999;
```

The next set of global variables include variables that are used generally with the DINA model, and specifically with its higher-order formulation. Examples of latter are `A`, `W`, `nnodes`, `Rkq`, `Nq`, `la`, `la1`, and `l0`. The HO-specific variable `prior_q` is an $2^K \times 1$ matrix with $P(\boldsymbol{\alpha}_l|A_q)$ as entries.

```
decl Q,X;
decl A,W,nnodes;
decl Rkq,Nq;
decl la,la1,la0;
decl like,post,alpha,g,s,se,lse;
decl r0,r1,n0,n1;
decl lprior,eta;
decl it,maxabs;
decl alpha1,ml;
decl prior_q,theta;
```

The functions `pattern` and `alphaeta` are the same functions as before.

The function `findpriorq` will return a $Q \times 2^K$ matrix where each row represents the probability (likelihood) of each attribute pattern $\boldsymbol{\alpha}_l$ given a particular A_q.

```
findpriorq(const l1, const l0){//QxL
decl tmp=exp(l1*(A-l0'));
tmp=tmp./(1+tmp);
tmp=log(tmp)*alpha'+log(1-tmp)*(1-alpha');
return(exp(tmp));
}//end findpriorq
```

```
findprior(){//Lx1
decl tmp=findpriorq(la1,la0).*(W*ones(1,2^K));
return(sumc(tmp)');
}//end findprior
```

The function `findprior` calls `findpriorq`, and summarizes the probability of each $\boldsymbol{\alpha}_l$ by finding the weighted sum of the likelihoods across the Q quadrature nodes.

The function `findlike` computes the entries of an $N \times 2^K$ matrix where each row represent one examinee, and the column is the likelihood of a response vector for a fixed $\boldsymbol{\alpha}_l$.

```
findlike(){//NxL
for(decl i=0;i<N;i++){
decl XX=ones(2^K,1)*X[i][];
decl tmp=eta.*(ones(2^K,1)*(1-s)')+(1-eta).*(ones(2^K,1)*g');
tmp=sumr(XX.*log(tmp)+(1-XX).*log(1-tmp));
like[i][]=exp(tmp)';
}}//end i, findlike
```

The function `findpost` is similar to the previous functions for finding the posterior attribute distribution of each examinee except for how the prior (this time involving θ) is computed. It updates the $N \times 2^K$ variable `post`.

```
findpost(){//NxL
for(decl i=0;i<N;i++){
decl XX=ones(2^K,1)*X[i][];
decl tmp=eta.*(ones(2^K,1)*(1-s)')+(1-eta).*(ones(2^K,1)*g');
tmp=sumr(XX.*log(tmp)+(1-XX).*log(1-tmp))+lprior;
tmp=exp(tmp);
post[i][]=(tmp./sumc(tmp))';
}}//end i, findpost
```

The function `HO_DINA` contains the system of equations that needs to be solved for $\boldsymbol{\lambda}$.

```
HO_DINA(const avF, const lam){//d is always vertical
decl p=lam[0]*(A-lam[1:]'); //Q*K
p=exp(p)./(1+exp(p));
decl tmp0=sumc(W*ones(1,K).*(Rkq-N*p))'; //(1*K)'
decl tmp=-sumc(W.*sumr((A-lam[1:]').*(Rkq-Nq*ones(1,K).*p)));
avF[0]= tmp|tmp0;
return 1;
}//end HO_DINA
```

The function `initiate` gives starting values to previously declared variables.

The function `est_itempar` finds g_j and s_j. It requires the argument `j`, and will be called from `iterate`.

```
est_itempar(const jj){
decl tmp=Q[jj][]*alpha';
tmp=tmp.==(Q[jj][]*ones(K,2^K));
tmp=sumr(post.*(ones(N,1)*tmp));
n0=N-sumc(tmp);
r0=sumc(X[][jj].*(1-tmp));
n1=sumc(tmp);
r1=sumc(X[][jj].*tmp);
g[jj]=r0/n0;
s[jj]=(n1-r1)/n1;
}//est_itempar
```

The function `est_hopar` estimate $\boldsymbol{\lambda}$, and is also called from `iterate`. After computing `Nq` and Rkq, which are obtained from the first derivatives, SolveNLE is called to solved the system of equations.

```
est_hopar(){
like=zeros(N,2^K);
decl post_q;
decl Rlkq=<>, Ilq=<>;
findlike();
prior_q=findpriorq(la1,la0); //p(al|Aq)=QxL
decl isum=like*prior_q'*W; //NxL x LxQ x Qx1
for(decl q=0;q<nnodes;q++){
post_q=like.*(ones(N,1)*prior_q[q][]); //NxL
post_q=post_q./(isum*ones(1,2^K));
Ilq=Ilq~sumc(post_q)';
Rlkq=Rlkq|sumc(alpha.*sumc(post_q)'.*ones(1,K));
}//end q
Nq=sumc(Ilq)';
Rkq=Rlkq;
la=la1|la0;
decl stat=SolveNLE(HO_DINA, &la);
la1=la[0];
la0=la[1:];
}//est_hopar
```

In addition to estimating the item parameters, the `iterate` function also estimates the higher-order parameters. And as before, it also updates the prior, posteriors, and iteration counter.

It should be noted that the stopping rule is based on the improvement on the item parameters only, and does not include the higher-order parameters.

The function `SE_item` will find the standard errors of g_j and s_j. It does so analytically using the cross-products of the first derivatives. The function will be called from `stderr`.

```
SE_item(const jj){
decl tmp,tmp00,tmp11,tmp01;
tmp=Q[jj][]*alpha';
tmp=tmp.==(Q[jj][]*ones(K,2^K));
tmp=sumr(post.*(ones(N,1)*tmp));
tmp00=sumc(((X[][jj]-g[jj]*ones(N,1)).*(1-tmp)).^2)/((g[jj]*(1-g[jj])).^2);
tmp11=sumc(((X[][jj]-(1-s[jj])*ones(N,1)).*tmp).^2)/((s[jj]*(1-s[jj]))^2);
tmp01=-sumc(((X[][jj]-g[jj]*ones(N,1)).*(1-tmp)).*((X[][jj]-(1-s[jj])*ones(N,1)).*tmp)
/(g[jj]*(1-g[jj])*s[jj]*(1-s[jj])));
se=se|sqrt(diagonal(invert((tmp00~tmp01)|(tmp01~tmp11))));
}//end SE_item
```

The function `SE_HO_N` is specifically designed to compute the the marginalized loglikelihood with `l1` and `l0` as arguments. The arguments are necessary to afford the function flexibility in computing the marginalized loglikehoods around $\hat{\boldsymbol{\lambda}}$, which are need for determining the second derivative.

```
SE_HO_N(const l1,const l0){
decl tmp=0;
for(decl i=0;i<1;i++){
decl XX=ones(2^K,1)*X[i][];
decl tmp1=eta.*(ones(2^K,1)*(1-s)')+(1-eta).*(ones(2^K,1)*g');
tmp1=ones(nnodes,1)*exp(sumr(XX.*log(tmp1)+(1-XX).*log(1-tmp1)))';
tmp1=tmp1.*findpriorq(l1,l0);
tmp1=tmp1.*(W*ones(1,2^K));
tmp=tmp+sumc(sumr(log(tmp1)));
}//end i
return(tmp);
}//end SE_HO
```

The function `stderr` will compute the standard errors of $\hat{\boldsymbol{g}}$ and $\hat{\boldsymbol{s}}$, and then $\hat{\boldsymbol{\lambda}}$.

Computing the standard errors of $\hat{g}_j$ and $\hat{s}_j$ is done one item at a time, and require simply calling the function `SE_item` with `j` as the argument.

Computing the standard errors of $\hat{\boldsymbol{\lambda}}$ is done numerically. In addition to $\hat{\boldsymbol{\lambda}}$, the marginalized loglikelihood needs to also be evaluated at $\hat{\boldsymbol{\lambda}} \pm h$. The evaluations for $\hat{\lambda}_1$ and $\hat{\boldsymbol{\lambda}}_0$ are carried out separately.

Note that only the diagonal elements of the information matrix are computed. Most likely, the resulting standard errors are larger than the actual values.

Write a code that evaluates all the elements of the information matrix. Compare the standard errors computed from this matrix to those computed using the diagonal elements only.

```
stderr(){
se=<>;
lse=<>;
findpost();
for(decl j=0;j<J;j++){// Analytical SE of item parameter estimates
SE_item(j);
}// end j
// Numerical SE of HO parameter estimates; diagonals only
decl h=.0001;
decl lamh=la1-h;
decl laph=la1+h;
decl tmp=(SE_HO_N(laph,la0)+SE_HO_N(lamh,la0)-2*SE_HO_N(la1,la0))/h^2;
lse=lse|tmp;
for(decl k=0;k<K;k++){
laph=la0;
laph[k]=laph[k]+h;
lamh=la0;
lamh[k]=lamh[k]-h;
tmp=(SE_HO_N(la1,laph)+SE_HO_N(la1,lamh)-2*SE_HO_N(la1,la0))/h^2;
lse=lse|tmp;
}//end k
lse=sqrt((-1)./lse);
}//end stderr
```

```
find_theta(){
prior_q=findpriorq(la1,la0); //p(al|Aq)=QxL
findlike(); //NxL
decl tmp2=zeros(N,nnodes);
for (decl l=0;l<2^K;l++){
decl tmp1=ones(N,1)*(prior_q[][l].*W)'; //Nx1 x 1xQ
tmp1=(like[][l]*ones(1,nnodes)).*tmp1;
tmp2=tmp2+tmp1;
}//end l
tmp2=tmp2./(sumr(tmp2)*ones(1,nnodes));//normalizing
theta=sumr(tmp2.*(ones(N,1)*A'));//expected value
}//end find_theta
```

The `find_theta` function computes the $E(\theta|\boldsymbol{X}_i)$. The function is based on the following result:

$$\begin{aligned} P(\theta|\boldsymbol{X}_i,\boldsymbol{\alpha},\boldsymbol{\lambda}) &\propto P(\boldsymbol{X}_i|\theta,\boldsymbol{\alpha},\boldsymbol{\lambda}) \times P(\theta|\boldsymbol{\alpha},\boldsymbol{\lambda}) \\ &\propto L(\boldsymbol{X}_i|\boldsymbol{\alpha}) \times P(\boldsymbol{\alpha}|\theta,\boldsymbol{\lambda}) \times P(\theta|\boldsymbol{\lambda}) \\ &= L(\boldsymbol{X}_i|\boldsymbol{\alpha}) \times P(\boldsymbol{\alpha}|\theta,\boldsymbol{\lambda}) \times P(\theta). \end{aligned}$$

The function `printout` prints two examinee-related output files: `alpha.out` and `theta.out`. The

former is the attribute classification, whereas the later the expected a posteriori of θ.

In addition, the function provides the following screen output: item parameter and higher-order parameter estimates and the associated standard errors, and the joint and marginal probabilities of the attributes.

Finally, the `main` function reads the data and Q-matrix, and calls the functions `initiate`, `iterate`, `stderr`, and `printout`.

1. Using `HODINA_gen.ox`, generate data for different values of $\boldsymbol{\lambda}$, $\boldsymbol{q}$, and $\boldsymbol{s}$, and N.

2. How are the different estimates affected by the different generating values?

An E-M Algorithms for the LLM

Recall that the LLM (linear logistic model) is just the additive model under the logit link. Its item response function is given by

$$P(X_j = 1|\boldsymbol{\alpha}^*_{jl}) = P_{\alpha^*_{jl}} = \frac{\exp\left(\lambda_{j0} + \sum_{k=1}^{K_j} \lambda_{jk}\alpha_{lk}\right)}{1 + \exp\left(\lambda_{j0} + \sum_{k=1}^{K_j} \lambda_{jk}\alpha_{lk}\right)},$$

or

$$\text{logit}(P_{\alpha^*_{jl}}) = \lambda_{j0} + \sum_{k=1}^{K_j} \lambda_{jk}\alpha_{lk}.$$

As can be seen from the model, the number of LLM parameters for item j is equal to $K_j + 1$.

To obtain the MLE of $\boldsymbol{\lambda}_j$, let's again start with the marginalized likelihood of the response vector of examinee i:

$$L(\boldsymbol{X}_i) = \sum_{l=1}^{L} L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l)p(\boldsymbol{\alpha}_l).$$

where, as usual, $p(\boldsymbol{\alpha}_l)$ is the prior probability of $p(\boldsymbol{\alpha}_l)$, and for dichotomous response,

$$L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l) = \prod_{j=1}^{J} P_{\boldsymbol{\alpha}^*_{jl}}^{X_{ij}} \times (1 - P_{\boldsymbol{\alpha}^*_{jl}})^{(1-X_{ij})}.$$

Finding $\hat{\boldsymbol{\lambda}}_j$ requires maximizing

$$l(\boldsymbol{X}) = \log \prod_{i=1}^{N} L(\boldsymbol{X}_i) = \sum_{i=1}^{N} \log L(\boldsymbol{X}_i)$$

with respect to $\boldsymbol{\lambda}_j$.

Taking the derivative,

$$
\begin{aligned}
\frac{\partial l(\boldsymbol{X})}{\partial \boldsymbol{\lambda}_j} &= \sum_{i=1}^{N} \frac{1}{L(\boldsymbol{X}_i)} \times \frac{\partial L(\boldsymbol{X}_i)}{\partial \boldsymbol{\lambda}_j} \\
&= \sum_{i=1}^{N} \frac{1}{L(\boldsymbol{X}_i)} \times \sum_{l=1}^{L} p(\boldsymbol{\alpha}_l) \times \frac{\partial L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l)}{\partial \boldsymbol{\lambda}_j} \\
&= \sum_{i=1}^{N} \frac{1}{L(\boldsymbol{X}_i)} \sum_{l=1}^{L} p(\boldsymbol{\alpha}_l) \times L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l) \times \left[\frac{X_{ij} - P_{\boldsymbol{\alpha}^*_{jl}}}{P_{\boldsymbol{\alpha}^*_{jl}}(1 - P_{\boldsymbol{\alpha}^*_{jl}})} \right] \times \frac{\partial P_{\boldsymbol{\alpha}^*_{jl}}}{\partial \boldsymbol{\lambda}_j} \\
&= \sum_{i=1}^{N} \sum_{l=1}^{L} P(\boldsymbol{\alpha}_l|\boldsymbol{X}_i) \times \left[\frac{X_{ij} - P_{\boldsymbol{\alpha}^*_{jl}}}{P_{\boldsymbol{\alpha}^*_{jl}}(1 - P_{\boldsymbol{\alpha}^*_{jl}})} \right] \times \frac{\partial P_{\boldsymbol{\alpha}^*_{jl}}}{\partial \boldsymbol{\lambda}_j}
\end{aligned}
$$

where $P(\boldsymbol{\alpha}_l|\boldsymbol{X}_i)$ is the posterior probability that examinee i has the attribute pattern $\boldsymbol{\alpha}_l$.

As in the previous lecture, we can group the latent classes, as in, $\boldsymbol{\alpha}_l \rightarrow \boldsymbol{\alpha}^*_{jl}$, to get

$$
P(\boldsymbol{\alpha}_l|\boldsymbol{X}_i) = \sum_{\forall l: \boldsymbol{\alpha}_l \rightarrow \boldsymbol{\alpha}^*_{jl}} P(\boldsymbol{\alpha}_l|\boldsymbol{X}_i),
$$

which leads to the following simplification:

$$
\frac{\partial l(\boldsymbol{X})}{\partial \boldsymbol{\lambda}_j} = \sum_{jl=1}^{L_j} P(\boldsymbol{\alpha}^*_{jl}|\boldsymbol{X}_i) \left[\frac{1}{P_{\boldsymbol{\alpha}^*_{jl}}(1 - P_{\boldsymbol{\alpha}^*_{jl}})} \right] \sum_{i=1}^{N} (X_{ij} - P_{\boldsymbol{\alpha}^*_{jl}}) \times \frac{\partial P_{\boldsymbol{\alpha}^*_{jl}}}{\partial \boldsymbol{\lambda}_j},
$$

where the outer sum involves 2^{K_j}, instead of 2^K terms.

Again, by interchanging the summations, and making some rearrangements, we get

$$
\begin{aligned}
\frac{\partial l(\boldsymbol{X})}{\partial \boldsymbol{\lambda}_j} &= \sum_{jl=1}^{L_j} \frac{\partial P_{\boldsymbol{\alpha}^*_{jl}}}{\partial \boldsymbol{\lambda}_j} \times \left[\frac{1}{P_{\boldsymbol{\alpha}^*_{jl}}(1 - P_{\boldsymbol{\alpha}^*_{jl}})} \right] \sum_{i=1}^{N} (X_{ij} - P_{\boldsymbol{\alpha}^*_{jl}}) \times P(\boldsymbol{\alpha}^*_{jl}|\boldsymbol{X}_i) \\
&= \sum_{jl=1}^{L_j} \frac{\partial P_{\boldsymbol{\alpha}^*_{jl}}}{\partial \boldsymbol{\lambda}_j} \times \left[\frac{1}{P_{\boldsymbol{\alpha}^*_{jl}}(1 - P_{\boldsymbol{\alpha}^*_{jl}})} \right] \times \left[\sum_{i=1}^{N} X_{ij} \times P(\boldsymbol{\alpha}^*_{jl}|\boldsymbol{X}_i) - P_{\boldsymbol{\alpha}^*_{jl}} \times \sum_{i=1}^{N} P(\boldsymbol{\alpha}^*_{jl}|\boldsymbol{X}_i) \right] \\
&= \sum_{jl=1}^{L_j} \frac{\partial P_{\boldsymbol{\alpha}^*_{jl}}}{\partial \boldsymbol{\lambda}_j} \times \left[\frac{1}{P_{\boldsymbol{\alpha}^*_{jl}}(1 - P_{\boldsymbol{\alpha}^*_{jl}})} \right] \times \left[R^*_{jl} - P_{\boldsymbol{\alpha}^*_{jl}} \times N^*_{jl} \right],
\end{aligned}
$$

where $R^*_{jl} = \sum_{i=1}^{N} X_{ij} \times P(\boldsymbol{\alpha}^*_{jl}|\boldsymbol{X}_i)$ and $N^*_{jl} = \sum_{i=1}^{N} P(\boldsymbol{\alpha}^*_{jl}|\boldsymbol{X}_i)$.

By expanding the summation, the derivative can be written as

$$\begin{aligned}\frac{\partial l(\boldsymbol{X})}{\partial \boldsymbol{\lambda}_j} &= \frac{\partial P_{\boldsymbol{\alpha}^*_{j1}}}{\partial \boldsymbol{\lambda}_j} \times \left[\frac{1}{P_{\boldsymbol{\alpha}^*_{j1}}(1-P_{\boldsymbol{\alpha}^*_{j1}})}\right] \times \left[R^*_{j1} - P_{\boldsymbol{\alpha}^*_{j1}} \times N^*_{j1}\right] + \\ &\quad \frac{\partial P_{\boldsymbol{\alpha}^*_{j2}}}{\partial \boldsymbol{\lambda}_j} \times \left[\frac{1}{P_{\boldsymbol{\alpha}^*_{j2}}(1-P_{\boldsymbol{\alpha}^*_{j2}})}\right] \times \left[R^*_{j2} - P_{\boldsymbol{\alpha}^*_{j2}} \times N^*_{j2}\right] + \ldots + \\ &\quad \frac{\partial P_{\boldsymbol{\alpha}^*_{jL_j}}}{\partial \boldsymbol{\lambda}_j} \times \left[\frac{1}{P_{\boldsymbol{\alpha}^*_{jL_j}}(1-P_{\boldsymbol{\alpha}^*_{jL_j}})}\right] \times \left[R^*_{jL_j} - P_{\boldsymbol{\alpha}^*_{jL_j}} \times N^*_{jL_j}\right].\end{aligned}$$

Because

$$\frac{\partial P_{\boldsymbol{\alpha}^*_{jl}}}{\partial \boldsymbol{\lambda}_j} = P_{\boldsymbol{\alpha}^*_{jl}} \times (1 - P_{\boldsymbol{\alpha}^*_{jl}}) \times \frac{\partial \text{logit}(P_{\boldsymbol{\alpha}^*_{jl}})}{\partial \boldsymbol{\lambda}_j},$$

and by denoting $R^*_{jl} - P_{\boldsymbol{\alpha}^*_{jl}} \times N^*_{jl}$ as w_{jl} and $\text{logit}(P_{\boldsymbol{\alpha}_{jl}})$ as ℓ_{jl}, we can write $\partial l(\boldsymbol{X})/\partial \boldsymbol{\lambda}_j$ in matrix notation, as in,

$$\begin{aligned}\frac{\partial l(\boldsymbol{X})}{\partial \boldsymbol{\lambda}_j} &= \left[\frac{\partial \boldsymbol{\ell}_{jl}}{\partial \boldsymbol{\lambda}_j}\right]'_{L_j \times (K_j+1)} \times \boldsymbol{w}_j \\ &= \begin{bmatrix} \frac{\partial \ell_1}{\partial \lambda_{j0}} & \frac{\partial \ell_1}{\partial \lambda_{j1}} & \cdots & \frac{\partial \ell_1}{\partial \lambda_{jK_j}} \\ \frac{\partial \ell_2}{\partial \lambda_{j0}} & \cdots & \cdots & \vdots \\ \vdots & \cdots & \cdots & \vdots \\ \frac{\partial \ell_{L_j}}{\partial \lambda_{j0}} & \cdots & \cdots & \frac{\partial \ell_{L_j}}{\partial \lambda_{jK_j}} \end{bmatrix}' \times \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_{L_j} \end{bmatrix}.\end{aligned}$$

The logit of the item response function of LLM can also be written in matrix notation, as in,

$$\begin{bmatrix} \ell_1 \\ \ell_2 \\ \vdots \\ \ell_{L_j} \end{bmatrix} = \begin{bmatrix} \mathbf{1} & \mathbf{A}^*_j \end{bmatrix} \times \begin{bmatrix} \lambda_{j0} \\ \lambda_{j1} \\ \vdots \\ \lambda_{jL_j} \end{bmatrix},$$

where $\mathbf{1}$ is an $L_j \times 1$ matrix of 1s, and $\mathbf{A}^*_j$ is $2^{L_j} \times L_j$ matrix of stacked $\boldsymbol{\alpha}^*_{jl}$. $[\mathbf{1} \quad \mathbf{A}^*_j]$ is also called our *design* matrix.

For example, if two attributes are required for item j, we have,

$$\begin{bmatrix} \ell_1 \\ \ell_2 \\ \ell_3 \\ \ell_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} \lambda_{j0} \\ \lambda_{j1} \\ \lambda_{j2} \end{bmatrix}.$$

It should be easy to extract what each row means. For instance, for row 2, which represents P_{10},

$$\ell_2 = \text{logit}(P_{10}) = \lambda_{j0} + \lambda_{j1}.$$

Note that the derivative of ℓ_2 with respect to λ_0, λ_1, and λ_2 is simply $[1 \quad 1 \quad 0]$, which corresponds to the second row of the design matrix.

It can be easily shown that, in general,

$$\begin{bmatrix} \frac{\partial \ell_1}{\partial \lambda_{j0}} & \frac{\partial \ell_1}{\partial \lambda_{j1}} & \cdots & \frac{\partial \ell_1}{\partial \lambda_{jK_j}} \\ \frac{\partial \ell_2}{\partial \lambda_{j0}} & \cdots & \cdots & \vdots \\ \vdots & \cdots & \cdots & \vdots \\ \frac{\partial \ell_{L_j}}{\partial \lambda_{j0}} & \cdots & \cdots & \frac{\partial \ell_{L_j}}{\partial \lambda_{jK_j}} \end{bmatrix} = \begin{bmatrix} \mathbf{1} & \mathbf{A}_j^* \end{bmatrix}.$$

Thus,

$$\frac{\partial l(\boldsymbol{X})}{\partial \boldsymbol{\lambda}_j} = \begin{bmatrix} \mathbf{1} & \mathbf{A}_j^* \end{bmatrix}' \times \boldsymbol{w}_j.$$

We get $\hat{\boldsymbol{\lambda}}_j$ by solving for $\boldsymbol{\lambda}_j$ in

$$\begin{bmatrix} \mathbf{1} & \mathbf{A}_j^* \end{bmatrix}' \times \boldsymbol{w}_j = \mathbf{0}.$$

If we use `solveNLE` in Ox, we do not need the second derivatives to get the MLEs.

Finding the MLEs and the corresponding standard errors are two distinct steps. We will consider our options for the latter later.

Implementation of EM Algorithm for the LLM in Ox (I)

The program `LLM_NLE.ox` will estimate the item parameters of the LLM by solving the zero of the first derivative of the marginalized loglikelihood. It requires both item responses and the Q-matrix as input.

This program includes the following headers:

```
#include <oxstd.h>
#include <oxprob.h>
#include <oxfloat.h>
#import <maxsqp>
#import <solvenle>
```

```
decl N=1000, J=30,K=5;
decl X,Q,Qj;
decl Kj,Kjmax;
decl Ng,Rg,Ngj,Rgj;
decl alpha,eta,est,post,coef;
decl prior,lprior;
decl lambda,SE,jj,fL,Prob;
```

We first declare a few basic global variables. Some of which are the same as the variables that we have defined previously.

In the program we declare additional global variables: `Ng` and `Rg` are the matrix for the number of examinees in each latent group and the number of examinees in each latent group answering item correctly; `Ngj` and `Rgj` are the j^{th} row of `Ng` and `Rg`; and `coef` is a matrix for item parameters, which in our case is $\boldsymbol{\lambda}_j$.

The last set of variables will be used in conjunction with the objective function, which has a specific form. The objective function will be needed for the numerical second derivatives.

The functions `pattern`, `alphaeta` and `findpost` are the same functions as before.

The function `NgRg` will calculate `Ng` and `Rg` for item j.

```
NgRg(){
for (decl j=0;j<J;j++){ //j loop: for each item
decl Kj=sumr(Q[j][]);
decl Qj=pattern(Kj);
decl Lj=vecindex(Q[j][].==1);
decl Rjtmp=<>;
decl Ijtmp=<>;
//re-group the 2^K groups in to 2^Kj groups
for(decl kj=0;kj<2^Kj;kj++){
decl tmp_loc=vecindex(prodr(ones(2^K,1)*Qj[kj][].==alpha[][Lj]).==1);
decl tmp=sumr(post[][tmp_loc]);
Ijtmp=Ijtmp|sumc(tmp);   //2^Kj col-vector
Rjtmp=Rjtmp|sumc(X[][j].*tmp);   //2^Kj col-vector
}//end kj
Ng[j][0:2^Kj-1]=Ijtmp';  //# of examinees expected in latent class
Rg[j][0:2^Kj-1]=Rjtmp';  //# of examinees expected to answer correctly
}//end j
} //end NgRg
```

To solve the system of first derivatives, we will use the `SolveNLE` function in Ox, rather than writing our own zero-finder function.

The function `LLM_NLE`, which will be called in `SolveNLE`, can be used for this purpose. In this code, `M` is the design matrix $[\mathbf{1} \quad \boldsymbol{\alpha}_j^*]$, `Qj` is the reduced attribute pattern $\boldsymbol{\alpha}_{jl}$ obtained from the function `pattern` with the argument of `Kj`, and `w` is w_{jl}.

```
LLM_NLE(const avF, const d){
decl M=ones(2^Kj,1)~Qj;
decl P=M*d;
P=exp(P)./(1+exp(P));
decl w=(Rgj-Ngj.*P);
avF[0]=M'*w;
return 1;
}//end LLM_NLE
```

As with the 2PL, `fullLike` provides an $N \times 2^K$ matrix where the l^{th} element in row i is $L(\boldsymbol{X}_i|\boldsymbol{\alpha}_l)$.

```
fullLike(){//returns N x 2^K
fL=<>;
Prob=zeros(2^K,J);
for(decl  j=0;j<J;j++){
decl tmp=vecindex(Q[j][].==1);
decl astar=(1~alpha[][tmp])*coef[j][0:sumr(Q[j][])]';
Prob[][j]=exp(astar)./(1+exp(astar));
}//end j
for(decl  i=0;i<N;i++){
decl  Xi=ones(2^K,1)*X[i][];
decl tmp=sumr(Xi.*log(Prob)+(1-Xi).*log(1-Prob));
fL=fL|exp(tmp');
}} //end i, fullLike
```

`LLM_obj_func` is the objective function that we will call in finding the standard errors *numerically*. Note that this function uses the discounted likelihood to carry out the computations more efficiently.

```
LLM_obj_func(const vP, const adFunc, const avScore, const amHessian){
decl Prob0=ones(N,1)*Prob[][jj]';                    //N X 2^K
decl Xj=X[][jj]*ones(1,2^K);                        //N X 2^K
decl lj0=(Prob0.^Xj).*(1-Prob0).^(1-Xj);
decl fL_j=fL./lj0;
decl tmp=vecindex(Q[jj][].==1);
decl astar=(1~alpha[][tmp])*vP;
decl Prob=exp(astar)./(1+exp(astar));//2^K x 1
Prob=ones(N,1)*Prob';
adFunc[0]=sumc(log(sumr(fL_j.*(Prob.^Xj).*
      ((1-Prob).^(1-Xj)).*(ones(N,1)*prior'))));
return 1;
}//end LLM_obj_func
```

The function `findSE` is used to find the standard errors numerically once the item parameters have been estimated.

```
findSE(){
fullLike();
decl  mhess;
for (jj=0;jj<J;jj++){
decl vP=coef[jj][0:sumr(Q[jj][])]';
if (Num2Derivative(LLM_obj_func,  vP,  &mhess)){
SE[jj][0:sumr(Q[jj][])]=sqrt(diagonal(-invert(mhess)));
}}}//end if, j, findSE
```

The function `initiate` initializes the global variables. In addition to assigning specific values, it also called the functions `pattern`, `alphaeta`, `NgRg` and `findpost` functions. The initial values for the probabilities, $\boldsymbol{\lambda}$, and the prior distribution are also specified here.

```
initiate(){
alpha=pattern(K);
alphaeta();
Kjmax=max(sumr(Q));
est=-1*ones(J,2^(maxc(sumr(Q))));
coef=-1*ones(J,Kjmax+1);
Ng=zeros(J,2^Kjmax);
Rg=zeros(J,2^Kjmax);
for(decl  j=0;j<J;j++){
Kj=sumr(Q[j][]);
est[j][0:(2^Kj-1)]=.2;
est[j][2^Kj-1]=.8;
coef[j][0:Kj]=0.3;
}//  end  j
lprior=ones(2^K,1)./(2^K);
lprior=log(lprior/sumc(lprior));
post=ones(N,2^K);
findpost();
NgRg();
}//end  initiate
```

The function `iterate` is where the item parameters are estimated. The `while` loop is in effect as long as the difference in parameter estimates between two iterations remain large AND the maximum number of iterations has not been reached.

Within the `while` loop, `est`, the success probabilities of the different latent groups for each item, will be calculated after finishing the estimation of λs.

The posterior distributions, E-step, iteration number, and maximum item difference between two consecutive iterations will be updated after each iteration.

```
iterate(){
decl crit=0.001;
decl maxit=999;
decl it=0;
decl diff=<>;
decl maxabs=1;
decl f;
while (maxabs>crit&&it<maxit){
diff=coef;
for (decl j=0;j<J;j++) {
Kj=sumr(Q[j][]);
Qj=pattern(Kj);
Rgj=Rg[j][0:2^Kj-1]';
Ngj=Ng[j][0:2^Kj-1]';
decl coefj=coef[j][0:Kj]';
SolveNLE(LLM_NLE, &coefj);
coef[j][0:Kj]=coefj';
lambda=(ones(2^Kj,1)~Qj)*coef[j][0:Kj]';
est[j][0:(2^Kj-1)]=(exp(lambda)./(1+exp(lambda)))';
}//end j
prior=meanc(post)';
lprior=log(prior/sumc(prior));
findpost();
NgRg();
it++;
maxabs=max(sqrt((diff-coef).^2));
}//end while
}//end iterate
```

Finally, the main function will be used to read responses and Q-matrix, call the *initiate* and *iterate* function.

The main function also provides a few screen output:

The first set of output is $\hat{\boldsymbol{\lambda}}_j$ from `coef`. As before, -1 means no value.

The second set of output is $SE(\hat{\boldsymbol{\lambda}}_j)$ from `SE`. As before, 1 means no value.

The last set of output is $\hat{\boldsymbol{\lambda}}_j \rightarrow P_{\boldsymbol{\alpha}^*_{jl}}$ from `est`.

```
main(){
format(10000);
decl file0, tmp0;
file0=fopen("sample_data.dat","r");
fscan(file0,"%#M",N,J,&X);
fclose(file0);
file0=fopen("sample_Q.dat","r");
fscan(file0,"%#M",J,K,&Q);
fclose(file0);
initiate();
iterate();
print ("\nEstimates of Lambdas:",coef);
decl SE=findSE();
print("\nStandard Errors of Lambda Estimates:",SE);
print("\n Latent Group Success Probabilities:",est);
}//end main
```

An Implementation of EM Algorithm for the LLM in Ox (II)

Recall that given a provisional posterior distribution and the response vector $\boldsymbol{X}_j$, we can compute $R^*_{jl} = \sum_{i=1}^N X_{ij} \times P(\boldsymbol{\alpha}^*_{jl}|\boldsymbol{X}_i)$ and $N^*_{jl} = \sum_{i=1}^N P(\boldsymbol{\alpha}^*_{jl}|\boldsymbol{X}_i)$.

We can use this information as another way of estimating $\boldsymbol{\lambda}_j$. We can to maximize the loglikelihood associated with item j given R^*_{jl} and N^*_{jl}, and this loglikelihood is given by

$$\begin{aligned} l(\boldsymbol{X}_j) &= \log \prod_{l=1}^{L_j} P_{\boldsymbol{\alpha}^*_{jl}}^{R^*_{jl}} (1 - P_{\boldsymbol{\alpha}^*_{jl}})^{N^*_{jl} - R^*_{jl}} \\ &= \sum_{l=1}^{L_j} \left[R^*_{jl} \times \log(P_{\boldsymbol{\alpha}^*_{jl}}) + (N^*_{jl} - R^*_{jl}) \times \log(1 - P_{\boldsymbol{\alpha}^*_{jl}}) \right]. \end{aligned}$$

In `LLM_SQP.ox`, the function associated with $l(\boldsymbol{X}_j)$ is given below:

```
LLM_obj_func0( const vP, const avF,const avS, const avM){
Qj=pattern(Kj);
decl M=ones(2^Kj,1)~Qj;
decl P=M*vP;
P=exp(P)./(1+exp(P));
avF[0]=sumc(Rgj.*log(P)+(Ngj-Rgj).*log(1-P));
return 1;
}// end LLM_obj_func0
```

This function differentiates `LLM_SQP.ox` from `LLM_NLE.ox`. Of course, we need to change how we obtain the estimate in `iterate`: from `SolveNLE` to `MaxSQPF`.

Note that with `MaxSQPF`, we don't need to know the derivatives. Thus, this can be used with any CDM!

Additional Exercises

1. Comparing the saturated DINA and HO-DINA models

a) Generate DINA model data using attributes that have i) uniformly distributed structure, and ii) higher-order structure.

b) Estimate the data using the estimation programs for the saturated DINA and HO-DINA models.

c) Compare the item parameter estimates, attribute distribution, attribute prevalence, and attribute classification of the two models.

2. Estimating the parameters of the A-CDM by maximizing the log-likelihood.

What we need to do first is to modify the following code in `iterate` function in `LLM_SQP.ox`, to it to be applicable to the A-CDM.

```
est[j][0:(2^Kj-1)]=(exp(lambda)./(1+exp(lambda)))';
```

Note that the above code is based on the item response function of the LLM IRF. Recall, this item response function is

$$P(X_j = 1|\boldsymbol{\alpha}_{jl}^*) = \frac{\exp\left(\lambda_{j0} + \sum_{k=1}^{K_j^*} \lambda_{jk}\alpha_{lk}\right)}{1 + \exp\left(\lambda_{j0} + \sum_{k=1}^{K_j^*} \lambda_{jk}\alpha_{lk}\right)}.$$

For the A-CDM, that above code should be replaced by:

```
est[j][0:(2^Kj-1)]=lambda';
```

Also, because of the formulation in the identity link, we need to add constraints to the parameters for the estimates to be acceptable. Specifically, we use the constraints:

$$0 \leq P_{\boldsymbol{\alpha}_{jl}^*} \leq 1, \quad jl = 1, \ldots, L_j.$$

The code below can be used as the constraints, and needs to be called in `MaxSQPF`. Modify `MaxSQPF` to include these constraints.

```
cfunc(const avF, const vP){
decl M=ones(2^Kj,1)~Qj;
decl P=M*vP;
avF[0]=P|(1-P);
return 1;
}//end cfunc
```

Optional Exercises

1. Normality of $\boldsymbol{\theta}$ Not Assumed (not sure this can be done)

To relax the normality assumption about θ, $W(A_q)$ can be estimated instead of being derived from $N(0,1)$. To do this, we can find $W(A_q|\boldsymbol{X})$, the posterior weight of the quadrature node A_q given the data.

The posterior weight can be written as

$$\begin{aligned} W(A_q|\boldsymbol{X}) &\propto p(\boldsymbol{X}|A_q) \times W(A_q) \\ &= \left[\sum_{l=1}^{L} p(\boldsymbol{X}, \boldsymbol{\alpha_l}|A_q)\right] \times W(A_q) \\ &= \left[\sum_{l=1}^{L} p(\boldsymbol{X}|\boldsymbol{\alpha_l}, A_q) \times p(\boldsymbol{\alpha_l}|A_q)\right] \times W(A_q) \\ &= \left[\sum_{l=1}^{L} p(\boldsymbol{X}|\boldsymbol{\alpha_l}) \times p(\boldsymbol{\alpha_l}|A_q)\right] \times W(A_q). \end{aligned}$$

The last equality follows from the conditional independence of $\boldsymbol{X}$ given $\boldsymbol{\alpha}_l$. The posterior weights need to be normalized to ensure that $\sum_{q=1}^{Q} W(A_q|\boldsymbol{X}) = 1$, and the mean and variance of the latent trait are always equal to 0 and 1, respectively.

This step can be viewed as Step 4 of the HO-IRT EM iteration process, and needs to be repeated together with Steps 2 and 3.

2. As mentioned before, the logit of the higher-order structure written in two ways.

$$\lambda_1(\theta - \lambda_{0k}) \quad \text{or} \quad \lambda_1 \times \theta + \lambda^*_{0k}.$$

The former is called the *discrimination-difficulty* form; the latter, the *slope-intercept* form. The latter has at least two advantages. First, it is computationally more stable. That is, even if the $\lambda_1 = 0$, we can still estimate λ^*_{0k}.

Why is not possible with the former formulation?

Second, it makes higher-order CDMs more general. Recall that when $\lambda_1 = 0$, we have the independence model, as in,

$$P(\boldsymbol{\alpha}_l|\theta) = \prod_{\forall k} P(\alpha_{lk}|\theta) = \prod_{\forall k} P(\alpha_{lk}).$$

However, $P(\alpha_{lk})$, the probability of mastering attribute k, can vary in size (i.e., some attributes are more difficult to master than others.

A special case is when $\lambda_{0k}* = 0$, $P(\alpha_{lk}) = 0.5$ for all k. This will produce a uniform attribute structure.

Note that, whenever $\lambda_1 = 0$, $\lambda_{0k}^* = 0$ is implied in the discrimination-difficulty formulation.

If we do not want $\lambda_1 = 0$ to imply $\lambda_{0k}* = 0$, we need to estimate the higher-order parameters using the slope-intercept form.

Write an EM algorithm for the higher-order DINA model that estimates λ_1 and $\lambda_{0k}*$.

Latent Variable Modeling and Statistical Computing
Spring 2016

Markov Chain Monte Carlo - Part I

Introduction

Background

Statistical models are formal mathematical expressions of processes that provide description of the assumed structure of the observed data.

Notwithstanding the inability of simple conventional models to describe accurately the intricate structures of most real-life phenomena, these models remain attractive and in use because of their computational ease.

However in many situations, adequate understanding of these phenomena can only be facilitated by using more complex models.

In these settings, traditional methods of estimation (e.g., numerical evaluation, analytic approximation) cannot address the computational requisites inherent in these models.

Simulation-based analyses such as MCMC are particularly suited for high-dimensional, complex problems. The application of this method had been limited in the past by its computational requirements.

However, the advent of affordable and fast computing machines has rendered this impediment a non-issue, and has renewed the interest of more than a few researchers in MCMC. As a result, the breadth of its application has increased tremendously in recent years.

Methods of Analysis

A researcher's statistical persuasion - frequentist or Bayesian - determines the choice of techniques and inferences.

From a frequentist perspective, random quantities are comprised of the data, and perhaps, random effects for some applications.

From a Bayesian perspective, no distinction exists between the observable data and the unobservable parameters - both are considered random variables.

The frequentist inference is based on the likelihood function, whereas the Bayesian inference is based on the posterior distribution.

The mathematical computations in each perspective seek solutions to different problems. The frequentist problem is finding an explicit definition of estimators as solutions to maximization problems, whereas the Bayesian problem is finding an implicit representation of estimators as an integral.

Although MCMC lends itself naturally to Bayesian inference, it can also be applied to optimization problems.

Bayesian Inference in IRT

A Bayesian approach to parameter estimation in IRT has been done in the past. Although the estimation procedures have a Bayesian flavor, they differ from the current line of Bayesian estimation in that numerical procedures and approximation are used instead of stochastic integration.

Let $\boldsymbol{X}$ denote the response data, and $\boldsymbol{\zeta}$ denote the collection of parameters (e.g., ability and item parameters).

In a Bayesian context, inference about $\boldsymbol{\zeta}$ given $\boldsymbol{X}$ requires setting up the joint distribution $p(\boldsymbol{\zeta}, \boldsymbol{X})$.

The joint distribution can be written as the product of the prior distribution $p(\boldsymbol{\zeta})$ and the sampling distribution or likelihood function $p(\boldsymbol{X}|\boldsymbol{\zeta})$:

$$p(\boldsymbol{\zeta}, \boldsymbol{X}) = p(\boldsymbol{\zeta})p(\boldsymbol{X}|\boldsymbol{\zeta}).$$

Using Bayes theorem, the posterior distribution of $\boldsymbol{\zeta}$ given $\boldsymbol{X}$ is

$$\begin{aligned} p(\boldsymbol{\zeta}|\boldsymbol{X}) &= \frac{p(\boldsymbol{\zeta})p(\boldsymbol{X}|\boldsymbol{\zeta})}{\int p(\boldsymbol{\zeta})p(\boldsymbol{X}|\boldsymbol{\zeta})\partial\boldsymbol{\zeta}} \\ &= \frac{p(\boldsymbol{\zeta})p(\boldsymbol{X}|\boldsymbol{\zeta})}{p(\boldsymbol{X})}. \end{aligned}$$

For given data, $\boldsymbol{X}$ is considered constant and $p(\boldsymbol{X})$ does not depend on $\boldsymbol{\zeta}$ so it can be omitted from the posterior. Hence, it can be written in an unnormalized form as

$$p(\boldsymbol{\zeta}|\boldsymbol{X}) \propto p(\boldsymbol{\zeta})p(\boldsymbol{X}|\boldsymbol{\zeta}).$$

It is not uncommon in Bayesian applications for the posterior to be known only up a proportionality constant.

Inference about any the features of posterior (e.g., moments, quantiles, regions of highest posterior density) can be made. Specifically, finding the posterior expectations of a function of $\boldsymbol{\zeta}$ is of particular interest.

Denoting the function of $\boldsymbol{\zeta}$ as $f(\boldsymbol{\zeta})$, its posterior expectation is given by

$$\begin{aligned} E[f(\boldsymbol{\zeta})|\boldsymbol{X}] &= \int f(\boldsymbol{\zeta})p(\boldsymbol{\zeta}|\boldsymbol{X})\partial\boldsymbol{\zeta} \\ &= \frac{\int f(\boldsymbol{\zeta})p(\boldsymbol{\zeta})p(\boldsymbol{X}|\boldsymbol{\zeta})\partial\boldsymbol{\zeta}}{\int p(\boldsymbol{\zeta})p(\boldsymbol{X}|\boldsymbol{\zeta})\partial\boldsymbol{\zeta}}. \end{aligned}$$

For high-dimensional, non-standard and complex models, analytical or numerical solutions or approximations of $E[f(\boldsymbol{\zeta})$ are not typically available.

One possible solution to this problem without constraining the models (e.g., by using conjugate priors) is to perform Monte Carlo integration based on Markov chains that reduces the computational involvedness of $E[f(\boldsymbol{\zeta})]$ into a sequence of simpler calculations.

Monte Carlo Integration

For notational convenience and greater generality, denote the distribution of interest as $\pi(\boldsymbol{\zeta})]$, which in this case is the posterior distribution. The expectation problem can then be reexpressed as

$$E[f(\boldsymbol{\zeta})] = \int f(\boldsymbol{\zeta})\pi(\boldsymbol{\zeta})\partial\boldsymbol{\zeta}.$$

If a sample $\boldsymbol{\zeta}^{(t)}$, $t = 1, 2, \ldots, T$, can be drawn from $\pi(\boldsymbol{\zeta})$, the expected value of any transformation of $\boldsymbol{\zeta}$ can be estimated by the average of the corresponding transformation of $\boldsymbol{\zeta}^{(t)}$. Hence, the estimator of $E[f(\boldsymbol{\zeta})]$ is

$$\widetilde{E}[f(\boldsymbol{\zeta})] = \frac{1}{T}\sum_{i=1}^{T} f(\boldsymbol{\zeta}^{(t)}).$$

This estimator is a method of moments estimator of $E[f(\boldsymbol{\zeta})]$, and by the strong law of large numbers is a strongly consistent estimator of $E[f(\boldsymbol{\zeta})]$.

The approximation error of $E[f(\boldsymbol{\zeta})]$ can be probabilistically determined, and because the sample size T is determined by the researcher, the approximation error can be made as small as desired by increasing the number of draws from $\pi(\boldsymbol{\zeta})$.

In Bayesian inference, however, sampling directly from $\pi(\boldsymbol{\zeta})$ is not always possible.

One solution to this problem is to indirectly generate *independent* samples from $\pi(\boldsymbol{\zeta})$ using the acceptance-rejection (A-R) method.

Acceptance-Rejection Sampling

From the equation of the posterior distribution, it is easy to see that the it can be written as

$$\pi(\boldsymbol{\zeta}) = g(\boldsymbol{\zeta})/c$$

where $g(\boldsymbol{\zeta}) = p(\boldsymbol{\zeta})p(\boldsymbol{\zeta}|\boldsymbol{X})$ and c is the normalizing constant.

Suppose that for all values of $\boldsymbol{\zeta}$, a distribution $q(\boldsymbol{\zeta})$ which can be easily simulated and a known constant k are available such that the blanketing condition $g(\boldsymbol{\zeta}) \leq kq(\boldsymbol{\zeta})$ is satisfied, then a sample from $\pi(\boldsymbol{\zeta})$ can be drawn as follows:

1. Draw a candidate $\boldsymbol{\zeta}^*$ from $q(\boldsymbol{\zeta})$;
2. Draw u from $\mathcal{U}(0,1)$;
3. Accept $\boldsymbol{\zeta}^*$ if $u \leq g(\boldsymbol{\zeta}^*)/kq(\boldsymbol{\zeta}^*)$; otherwise, repeat steps 1 through 3.

Recall that this procedure is known as the *acceptance-rejection* (A-R) sampling. For the method to be efficient, the k must be as small as possible.

However, finding the optimum k is in itself computationally formidable.

Hence, there is still a need for a straightforward method of drawing samples from any arbitrary distribution (i.e., MCMC).

Markov Chains

A sequence of random variable $\{\boldsymbol{\zeta}^{(0)}, \boldsymbol{\zeta}^{(1)}, \ldots, \boldsymbol{\zeta}^{(t)}, \ldots\}$ is said to be a Markov chain if the transition kernel $p(\boldsymbol{s}, A)$ satisfies the condition

$$\begin{aligned} p(\boldsymbol{s}, A) &= p(\boldsymbol{\zeta}^{(t+1)} \in A | \boldsymbol{\zeta}^{(0)} = \boldsymbol{s}_0, \ldots, \boldsymbol{\zeta}^{(t-1)} = \boldsymbol{s}_{t-1}, \boldsymbol{\zeta}^{(t)} = \boldsymbol{s}) \\ &= p(\boldsymbol{\zeta}^{(t+1)} \in A | \boldsymbol{\zeta}^{(t)} = \boldsymbol{s}), \end{aligned}$$

Stated differently, this equation means that, given the current state, the future and the past states are independent.

The transition kernel is the conditional probability that a chain will move from $\boldsymbol{s}$ to a point in A.

The ergodic theorem states that if the expected value of a function of $\boldsymbol{\zeta}$, $E_\pi[f(\boldsymbol{\zeta})]$, is finite, then the ergodic average $\bar{f}(\boldsymbol{\zeta}^{(T)}) = 1/T \sum_{t=1}^{T} f(\boldsymbol{\zeta}^{(t)})$ converges in probability to $E_\pi[f(\boldsymbol{\zeta})]$ as $T \to \infty$. The central limit theorem for ergodic Markov chains provides the asymptotic distribution of the ergodic average as

$$\lim_{T \to \infty} \sqrt{T} \frac{\bar{f}(\boldsymbol{\zeta}^{(T)}) - E_\pi[f(\boldsymbol{\zeta})]}{\sigma(f)} = N(0, 1),$$

where $\sigma(f)$ is real-valued but may be hard to obtain because of the dependence in the samples. The results above apply to most Markov chains used in simulations at present.

The Metropolis-Hastings Algorithm

This method was originally developed by *Metropolis*, Rosenbluth, Rosenbluth, Teller, and Teller (1953); a generalization of the method is provided by *Hastings* (1970).

As in the A-R method, M-H starts by sampling $\boldsymbol{z}$ from a candidate-generating distribution q. However, unlike in A-R, the Markovian nature of the chain allows $\boldsymbol{z}$ to depend on the current state $\boldsymbol{s}$.

For MCMC, the candidate-generating distribution is defined as $q(\boldsymbol{s}, \boldsymbol{z})$. For most applications, the use of $q(\boldsymbol{s}, \boldsymbol{z})$ alone will not satisfy an important condition (i.e., *reversibility*) for the samples to be deemed acceptable.

For instance, $q(\boldsymbol{s}, \boldsymbol{z})$ can be such that the process moves more often from $\boldsymbol{s}$ to $\boldsymbol{z}$ than the other way around. To achieve equilibrium, the number of moves from $\boldsymbol{s}$ to $\boldsymbol{z}$ can be restricted by introducing $\alpha(\boldsymbol{s}, \boldsymbol{z}) < 1$, the probability of move from $\boldsymbol{s}$ to $\boldsymbol{z}$.

The M-H transition kernel from $\boldsymbol{s}$ to $\boldsymbol{z}$ is given by

$$p_{MH}(\boldsymbol{s}, \boldsymbol{z}) = q(\boldsymbol{s}, \boldsymbol{z}) \alpha(\boldsymbol{s}, \boldsymbol{z}).$$

Whereas the A-R algorithm samples a new candidate value when $\boldsymbol{z}$ is rejected, the M-H algorithm simply returns the value $\boldsymbol{s}$.

Hence, reversibility is satisfied if the transition kernel $p_{MH}(\boldsymbol{s}, \boldsymbol{z})$ is defined with

$$\alpha(\boldsymbol{s}, \boldsymbol{z}) = \min\left\{\frac{\pi(\boldsymbol{z})q(\boldsymbol{z}, \boldsymbol{s})}{\pi(\boldsymbol{s})q(\boldsymbol{s}, \boldsymbol{z})}, 1\right\}.$$

An important feature of the M-H algorithm, particularly for Bayesian applications, is that **the normalizing constant is not required** in computing $\alpha(\boldsymbol{s}, \boldsymbol{z})$ because it appears in both the numerator and denominator of $\alpha(\boldsymbol{s}, \boldsymbol{z})$.

Below are the steps in drawing T draws from $\pi(\boldsymbol{\zeta})$ using the M-H algorithm:

1. Start from an arbitrary value $\boldsymbol{s}^{(0)}$;
2. Draw a candidate $\boldsymbol{z}^*$ from $q(\boldsymbol{s}^{(t)}, \boldsymbol{z})$;
3. Draw u from $\mathcal{U}(0,1)$;
4. Return $\boldsymbol{s}^{(t+1)} = \boldsymbol{z}^*$ if $u \leq \alpha(\boldsymbol{s}^{(t)}, \boldsymbol{z}^*)$; otherwise, return $\boldsymbol{s}^{(t+1)} = \boldsymbol{s}^{(t)}$.
5. Repeat steps 2 to 4 for $t = 1, 2, \ldots, T$;

The Gibbs Sampler

Suppose the elements of $\boldsymbol{\zeta}$ can be written as $\boldsymbol{\zeta} = \{\zeta_1, \zeta_2, \ldots, \zeta_d, \ldots, \zeta_D\}$.

The joint distribution $\pi(\boldsymbol{\zeta})$ can be written as $\pi(\zeta_d, \boldsymbol{\zeta}_d^{'})$ where $\boldsymbol{\zeta}_d^{'} = \{\zeta_1, \ldots, \zeta_{d-1}, \zeta_{d+1}, \ldots, \zeta_D\}$ for $d = 1, \ldots, D$.

One particular goal in a multivariate set-up is to be able to describe the *marginal* distribution $\pi(\zeta_d)$ defined to be

$$\pi(\zeta_d) = \int \pi(\zeta_d, \boldsymbol{\zeta}_d^{'}) \partial \boldsymbol{\zeta}_d^{'}.$$

For example, one may be interested in the expected value of a function of ζ_d, $E[f(\zeta_d)]$. If a sample $\{\zeta_d^{(1)}, \zeta_d^{(2)}, \ldots, \zeta_d^{(T)}\}$ can be drawn directly from $\pi(\zeta_d)$, then, as before, the expected value can be estimated by

$$\widetilde{E}[f(\zeta_d)] = \frac{\sum_{t=1}^{T} f(\zeta_d^{(t)})}{T}.$$

The discussion in the previous section on M-H algorithm assumes that $\boldsymbol{\zeta}$ can be sampled from a multivariate distribution, and hence allows the algorithm to be carried out by treating $\boldsymbol{\zeta}$ en bloc. This may not always be easy or efficient to implement.

An alternative technique that often simplifies the calculation is to apply the algorithm to the elements of $\boldsymbol{\zeta}$ one at a time. Suppose the transition kernel $q_d(s_d, z_d)$ is defined as

$$q_d(s_d, z_d) = \pi(\zeta_d | \boldsymbol{\zeta}_d^{'}) = \frac{\pi(\zeta_d, \boldsymbol{\zeta}_d^{'})}{\int \pi(\zeta_d, \boldsymbol{\zeta}_d^{'}) \partial \zeta_d},$$

and is called the *full conditional* distribution of ζ_d.

Suppose further that samples can be drawn directly from these conditional distributions. If a sequence is drawn as follows:

1. Start from an arbitrary value $\boldsymbol{\zeta}^{(0)} = \{\zeta_1^{(0)}, \zeta_2^{(0)}, \ldots, \zeta_D^{(0)}\}$;

2. At iteration $t+1$, draw a candidate ζ_d^* from $\pi\left(\zeta_d^{(t)} | \boldsymbol{\zeta}_{<d}^{(t+1)}, \boldsymbol{\zeta}_{>d}^{(t)}\right)$;

3. Repeat for $d = 1, 2, \ldots, D$;

4. Repeat steps 2 and 3 for $t = 1, 2, \ldots, T$;

then the process that yields the sequence

$$\{\zeta_1^{(0)}, \zeta_2^{(0)}, \ldots, \zeta_D^{(0)}, \zeta_1^{(1)}, \zeta_2^{(1)}, \ldots, \zeta_D^{(1)}, \ldots, \zeta_1^{(T)}, \zeta_2^{(T)}, \ldots, \zeta_D^{(T)}\}$$

is said to be the *Gibbs sampler*.

In Gibbs sampling, all candidate values are accepted (i.e., $\alpha(s_d, z_d) = 1$).

For distribution with multiple components, there are many ways of obtaining draws from the marginal distribution which can be derived from the joint and conditional distributions.

However, the defining characteristic of the Gibbs sampler is its use of the full set of the univariate conditional distributions to define the iterations.

In practice, Gibbs sampling has referred to any algorithm that recursively samples from the full conditional distributions, univariate or multivariate, directly or indirectly.

Under mild conditions

$$\pi(\zeta_1^{(t)}, \zeta_2^{(t)}, \ldots, \zeta_D^{(t)}) \quad \xrightarrow{\mathcal{D}} \quad \pi(\zeta_1, \zeta_2, \ldots, \zeta_D)$$

as $t \to \infty$, where $\xrightarrow{\mathcal{D}}$ denotes convergence in distribution.

This implies that for all d, $\pi(\zeta_d^{(t)}) \xrightarrow{\mathcal{D}} \pi(\zeta_d)$ holds.

Moreover, provided that each of the components of $\boldsymbol{\zeta}$ is visited infinitely often, convergence still holds even if the components are updated in no particular order, or only some of components are updated in each iteration.

From the above discussion, it can be seen that the Gibbs sampler is a special case of the more general M-H algorithm.

Because the Gibbs sampler is a special case of M-H algorithm, the ergodic theorem and convergence also apply to samples obtained via the Gibbs sampling.

However, for many researchers, this process is more appropriately called the *M-H-within-Gibbs algorithm*. That is, the sampling is from full conditional distributions (i.e., Gibbs sampling), and M-H is employed *within* Gibbs whenever the full condition distributions cannot be sampled directly.

That is, in situations where the full conditional distributions are unknown and cannot be sampled from directly, sampling from $q_d(s_d, z_d)$, which is typically different from $\pi(\zeta_d | \boldsymbol{\zeta}_d^{'})$, and then computing the corresponding $\alpha(s_d, z_d)$ can be used in conjunction with direct sampling from the known full conditional distributions.

Implementation Issues

Candidate-Generating Distribution

Described below are some of the families of candidate-generating distributions $q(\boldsymbol{s}, \boldsymbol{z})$ that are used in the M-H algorithm.

Random Walk Chain

One family of distribution for simulating $\boldsymbol{z}$ uses a random perturbation $\boldsymbol{\epsilon}$ from a distribution $q(\epsilon)$ such that

$$\boldsymbol{z} = \boldsymbol{s} + \boldsymbol{\epsilon}.$$

It follows that $q(\boldsymbol{s}, \boldsymbol{z})$ is of the form $q(\boldsymbol{z} - \boldsymbol{s})$. Because the candidate value is just the current values plus some noise, the Markov chain associated with this process is called a random walk.

The multivariate normal, t, and uniform distributions have been used as possible options for $q(\boldsymbol{\epsilon})$. The advantage of the above distributions, and any symmetric distributions for that matter (i.e., $q(\boldsymbol{\epsilon})$=$q(-\boldsymbol{\epsilon})$), is that the probability of move simplifies to

$$\alpha(\boldsymbol{s}, \boldsymbol{z}) = \min\left\{\frac{\pi(\boldsymbol{z})q(\boldsymbol{z}, \boldsymbol{s})}{\pi(\boldsymbol{s})q(\boldsymbol{s}, \boldsymbol{z})},\ 1\right\} = \min\left\{\frac{\pi(\boldsymbol{z})}{\pi(\boldsymbol{s})},\ 1\right\}.$$

Independence Chain

Another family of distributions for simulating $\boldsymbol{z}$ are those that draw the candidate value independent of $\boldsymbol{s}$ (i.e, $q(\boldsymbol{s}, \boldsymbol{z}) = q(\boldsymbol{z})$).

This gives a sequence that is called an *independence chain*.

Whereas the scale parameters are sufficient for generating the random walk chains, both location and scale parameters are needed to be specified in generating an independence chain.

It should be noted that although the process of generating candidate values are independent of the current values, the resulting chain is still a Markov chain because the $\alpha(\boldsymbol{s}, \boldsymbol{z})$ depends on $\boldsymbol{s}$.

M-H Acceptance-Rejection Sampling

The M-H acceptance-rejection method circumvents the problem of having to find the constant k that satisfies the blanketing condition required in the acceptance-rejection sampling.

For this method, the choice of k does not necessarily result in the target distribution being blanketed or dominated for all values of $\boldsymbol{s}$.

The method proceeds by dividing the support into two regions: dominated and non-dominated. The probability of move $\alpha(\boldsymbol{s}, \boldsymbol{z})$ is then adjusted depending on whether $\boldsymbol{s}$ or $\boldsymbol{z}$ or both fall in the dominated region.

Other families of candidate-generating distributions exist including those that give autoregressive chains and grid-based chains.

Regardless of how the candidate values are sampled, the whole support must be traversed for the chain to converge to the desired distribution. This is why the choice of $q(\boldsymbol{s}, \boldsymbol{z})$, which determines $\alpha(\boldsymbol{s}, \boldsymbol{z})$, is of particular importance.

The size of the move or jump needs to be carefully managed. On one hand, candidate-generating distributions that propose small moves will have high acceptance rate. On the other hand, distributions that propose big moves will invariably produce values that will have low probabilities of acceptance.

In both cases, it will take a long time for the chain to traverse the entire support, particularly in the tails.

Hence, a large number of iterations is needed before the chain can converge. This problem can be diagnosed by large autocorrelations among the sampled values.

To improve the rate of convergence, candidate-generating distributions similar to the target distribution can be used. Sampling from this distribution can produce high acceptance rate without making moves that are too small.

Another way of achieving faster convergence is to fine-tune the scale parameter so that an optimal acceptance rate can be obtained.

Although the results are not conclusive, some heuristic rules exist regarding the optimal acceptance rate. It is recommended that for one or two dimensions, the optimal rate is around 0.50, whereas for higher dimensions, the optimal rate is around 0.25.

Equivalently, in rare occasions where the asymptotic variance of the $\zeta^{(t)}$ is available, the scale parameter of $2.28/\sqrt{D}$ times the asymptotic variance can be used.

It is possible that even with these optimal acceptance rates, the autocorrelation may remain high. Such situations warrant the use of other families of candidate-generating distributions.

High autocorrelation does not mean the chain will not converge. It simply means we need more iterations to get there.

Blocking

The updating in M-H and Gibbs sampling sections can be viewed as two types of blocking that are used in MCMC.

However, aside from updating the elements of $\boldsymbol{\zeta}$ one at a time or simultaneously, the algorithm can also be applied to blocks of $\boldsymbol{\zeta}$.

Using blocks avoids the problem of low acceptance rates that are often encountered when updating is done simultaneously.

Furthermore, in some applications the elements of $\boldsymbol{\zeta}$ can have natural and meaningful groupings, updating the elements one at a time may be inefficient.

A recommendation is to block elements that are correlated and belong to the same context in the

model formulation.

Updating the chain is done by sampling directly from the conditional distributions of the blocks whenever possible, or using the M-H algorithm otherwise.

Number of Chains

Several options as to the number of chains that need to be run have been proposed.

The first is the use of many short chains as suggested. However, unless independent samples from the stationary distribution are required, this approach may be wasteful because independence is not required for the ergodic theorem to hold.

The more serious debate is choosing between several long chains and one very long chain.

This issue is closely related to the problem of convergence diagnosis (i.e., determining which is the better procedure in assessing convergence).

One advantage of using multiple chains is that if the chains have not reached the stationary distribution, the use of multiple chains from dispersed starting points may detect the lack of convergence.

There are cases where each of the chains appear to have converged when examined individually but may exhibit lack of convergence when compared to other chains.

Another benefit of using multiple chains is that by starting from dispersed points, the dependence of the chains on the starting values is reduced.

However, one drawback of using multiple chains in that convergence is dictated by the slower chains. Also, the MCMC algorithm needs to be altered to process samples obtained from different chains.

Finally, because a single chain that gives misleading results do not happen too often, the extra cost in implementing this procedure may not be worthwhile.

The major shortcoming of using a single chain is that it severely suffers from the "you've only seen where you've been" defect (i.e., parts of the support that have not been visited by the chain may not be detected).

The primary strength of using one long chain, especially for applications where the algorithm mixes slowly, is that it has a higher chance of reaching the stationary distribution compared to comparative number of samples from multiple but shorter chains.

Starting Values

Since the MCMC algorithm is based on Markov chains, it is assured to converge to the stationary distribution $\pi(\boldsymbol{\zeta})$ regardless of the starting value - as long as the chain is run long enough, it will forget its initial value.

However, care should still be exercised in selecting the initial values because extreme values can increase the number of iterations before convergence.

In severe cases, because of the numerical instability at the extreme tails of the distribution, the chain may fail to converge to the main support of the distribution.

Assessing Convergence

In practical applications of the MCMC methods, two of the important decisions to be made pertain to the length of burn-in and the length of the chain. These two questions are part of a bigger issue, that of convergence.

Burn-in refers to the sufficiently long number of iterations m before a sample $\boldsymbol{\zeta}^{(m+1)}$ and values thereafter, are said to be drawn from the stationary distribution $\pi(\boldsymbol{\zeta})$.

The samples drawn during the burn-in period, presumably still affected by the starting values, are usually discarded, and inference is based on the remaining $T - m$ samples only.

The most obvious and commonly used method of determining burn-in is visual inspection of the MCMC output.

Formal methods of determining the length of burn-in are available, which is subsumed under convergence diagnostics.

The more practical of these methods are those that provide assessment of convergence based of the output of the chain alone without resorting to the transition kernel.

However, it has been suggested that if extreme starting values are avoided, computing m may not be necessary for a sufficiently long chain because it is usually negligible, between 1% to 2% of the total length of the chain.

Determining the length of the chain is motivated by obtaining a desired precision of the estimate of $E[f(\boldsymbol{\zeta})]$.

The variance of $\widetilde{E}[f(\boldsymbol{\zeta})]$ is called the *Monte Carlo error*. One obvious way of computing the Monte Carlo error is to run several chains and then compute the variance of the estimates.

We want to minimize this variance relative to $\widetilde{V}[f(\boldsymbol{\zeta})]$ by using a sufficiently long enough chain.

The components of $\boldsymbol{\zeta}$ can be assessed for convergence one at a time using the Gelman-Rubin potential scale reduction factor statistics, or en bloc using the Brooks-Gelman multivariate potential scale reduction factor.

Output Analysis

Aside monitoring convergence, the output of the chain is primarily used for inference.

The earlier discussion of ergodic theorem specifically addressed the asymptotic behavior of the mean of a function of the sample.

However, as stated earlier, in practice chains are not run infinitely, and for a multi-component $\boldsymbol{\zeta}$, the interest lies in the characteristics of the marginal distributions.

The characteristics of interest of a particular component of $\boldsymbol{\zeta}$ can be computed by using the draws after the burn-in $\{\zeta_d^{(m+1)}, \ldots, \zeta_d^{(T)}\}$, and at the same time ignoring the output for other components.

For example, the mean and variance of a function of ζ_d can be estimated as

$$\widetilde{E}[f(\zeta_d)] = \frac{1}{T-m} \sum_{t=m+1}^{T} f\big(\zeta_d^{(t)}\big),$$

and

$$\widetilde{V}[f(\zeta_d)] = \frac{1}{T-m-1} \sum_{t=m+1}^{T} \Big(f\big(\zeta_d^{(t)}\big) - \widetilde{E}[f(\zeta_d)]\Big)^2,$$

respectively.

The same procedure can be followed in computing the quantiles and credibility intervals of interest.

Important Note: $f(\zeta_d) = \zeta_d$ is just a special case. For example, instead of ζ_d, we might be interested in $\log(\zeta_d)$.

In addition, marginal densities can also be obtained from the output as the mean of the conditional densities.

For some purposes, saving all the MCMC draws may require a large storage. As a result, a strategy of saving only every r^{th} draws after the burn-in, called *thinning*, has been implemented to deal with this problem.

At the same time, thinning the chain, which produces approximately independent draws, has been used to handle the dependency problem in the samples.

However, aside from advantages is computer storage, this method produces no gain in efficiency, and estimation based on the thinned chain is, in fact, less precise compared to estimation based on the entire chain.

Preliminary Examples

Univariate Normal Distribution

Suppose $X|\theta \sim N(\theta, \sigma^2)$, where σ^2 is known. We also assume the prior distribution $\theta \sim N(\mu_0, \tau_0^2)$.

We can write the prior and likelihood as:

$$\begin{aligned} p(\theta) &= \frac{1}{\sqrt{2\pi}\tau} \exp\left[-\frac{1}{2}\frac{(\theta-\mu_0)^2}{\tau_0^2}\right] \\ &\propto \exp\left[-\frac{1}{2}\frac{(\theta-\mu_0)^2}{\tau_0^2}\right] \end{aligned}$$

$$\begin{aligned} p(X|\theta) &= \frac{1}{\sqrt{2\pi}\sigma}\exp\left[-\frac{1}{2}\frac{(x-\theta)^2}{\sigma^2}\right] \\ &\propto \exp\left[-\frac{1}{2}\frac{(x-\theta)^2}{\sigma^2}\right]. \end{aligned}$$

What we are interested in is the posterior distribution of θ given X. As in, what is $p(\theta|X)$?

Note that a normal distribution is considered a *conjugate prior* for a normal likelihood because the resulting posterior is also normal.

This mean that we are assured that $p(\theta|X)$ is normal. We just need to find the posterior mean μ_1 and variance τ_1^2.

By definition of a posterior,

$$\begin{aligned} p(\theta|X) &\propto p(X|\theta)\times p(\theta) \\ &\propto \exp\left[-\frac{1}{2}\frac{(X-\theta)^2}{\sigma^2}\right]\times\exp\left[-\frac{1}{2}\frac{(\theta-\mu_0)^2}{\tau_0^2}\right] \\ &= \exp\left\{-\frac{1}{2}\left[\frac{(X-\theta)^2}{\sigma^2}+\frac{(\theta-\mu_0)^2}{\tau_0^2}\right]\right\}. \end{aligned}$$

We need to simplify the exponent so the posterior will be of the form

$$p(\theta|X)\propto\exp\left[-\frac{1}{2}\frac{(\theta-\mu_1)^2}{\tau_1^2}\right].$$

Note that for this problem the only random variable of interest is θ because once the data (i.e., x) are observed, they are no longer random (i.e., they can be considered fixed).

To get the desired form, we first need to expand the terms in the exponent.

$$\begin{aligned} p(\theta|X) &\propto \exp\left[-\frac{1}{2}\left(\frac{X^2}{\sigma^2}-2\frac{X\theta}{\sigma^2}+\frac{\theta^2}{\sigma^2}+\frac{\theta^2}{\tau_0^2}-2\frac{\mu_0\theta}{\tau_0^2}+\frac{\mu_0^2}{\tau_0^2}\right)\right] \\ &\propto \exp\left[-\frac{1}{2}\left(-2\frac{X\theta}{\sigma^2}+\frac{\theta^2}{\sigma^2}+\frac{\theta^2}{\tau_0^2}-2\frac{\mu_0\theta}{\tau_0^2}\right)\right] \\ &= \exp\left\{-\frac{1}{2}\left[\theta^2\left(\frac{1}{\tau_0^2}+\frac{1}{\sigma^2}\right)-2\theta\left(\frac{\mu_0}{\tau_0^2}+\frac{x}{\sigma^2}\right)\right]\right\} \end{aligned}$$

To complete the square, we can let

$$\begin{aligned} \frac{1}{\tau_1^2} &= \frac{1}{\tau_0^2}+\frac{1}{\sigma^2} \\ \mu_1 &= \frac{\frac{1}{\tau_0^2}\mu_0+\frac{1}{\sigma^2}X}{\frac{1}{\tau_0^2}+\frac{1}{\sigma^2}} \\ &= \left(\frac{\mu_0}{\tau_0^2}+\frac{X}{\sigma^2}\right)\times\tau_1^2. \end{aligned}$$

Then

$$\begin{aligned} p(\theta|X) &\propto \exp\left[-\frac{1}{2}\left(\frac{\theta^2}{\tau_1^2} - 2\theta\frac{\mu_1}{\tau_1^2}\right)\right] \\ &= \exp\left[-\frac{1}{2}\left(\frac{\theta^2 - 2\theta\mu_1}{\tau_1^2}\right)\right] \\ &= \exp\left[-\frac{1}{2}\left(\frac{\theta^2 - 2\theta\mu_1 + \mu_1^2}{\tau_1^2}\right) + c\right] \\ &\propto \exp\left[-\frac{1}{2}\frac{(\theta-\mu_1)^2}{\tau_1^2}\right]. \end{aligned}$$

Therefore,

$$\begin{aligned} \theta|X &\sim N(\mu_1, \tau_1^2) \\ &= N\left(\frac{\frac{1}{\tau_0^2}\mu_0 + \frac{1}{\sigma^2}X}{\frac{1}{\tau_0^2} + \frac{1}{\sigma^2}}, \left(\frac{1}{\tau_0^2} + \frac{1}{\sigma^2}\right)^{-1}\right). \end{aligned}$$

What happens to the influence of the prior on the posterior as τ_0^2 gets larger?

If we have N independent draws from $x_i|\theta \sim N(\theta, \sigma^2)$, then the posterior can be written as

$$\begin{aligned} \theta|\boldsymbol{X} &\sim N(\mu_N, \tau_N^2) \\ &= N\left(\frac{\frac{1}{\tau_0^2}\mu_0 + \frac{N}{\sigma^2}\bar{X}}{\frac{1}{\tau_0^2} + \frac{N}{\sigma^2}}, \left(\frac{1}{\tau_0^2} + \frac{N}{\sigma^2}\right)^{-1}\right) \end{aligned}$$

1. Show that the posterior distribution of $\theta|\boldsymbol{X}$ is indeed $N(\mu_N, \tau_N^2)$.

2. What happens to the posterior when the prior becomes less and less informative (i.e., $\tau_0^2 \to \infty$)?

Once we know the posterior distribution, we can now make inferences about the distribution. For example, we can now find its mean $E(\theta|\boldsymbol{X})$, variance $V(\theta|\boldsymbol{X})$, and the 95% CI (credibility interval) for $\theta|\boldsymbol{X}$.

At this point, we have a few ways of doing it.

1. Since we know the exact form of the posterior distribution, we can derive μ_N, τ_N^2, and 95% CI for $\theta|\boldsymbol{X}$ analytically.

2. Again since we know the form of the distribution, can sample *directly* from the posterior.

3. For pedagogical purposes, we can also sample from the posterior indirectly using the M-H algorithm.

For 1, the solution is:

$$\begin{aligned} \tilde{E}(\theta|\boldsymbol{X}) &= \mu_N \\ \tilde{V}(\theta|\boldsymbol{X}) &= \tau_N^2 \\ 95\%\ CI\ \text{for}\ \theta|\boldsymbol{X} &: [\mu_N - 1.96 \times \tau_N, \mu_N + 1.96 \times \tau_N] \end{aligned}$$

For 2, the solution is:

First, sample T draws from $N(\theta_N, \tau_N^2)$. Call the the sample $\boldsymbol{\theta}^* = \{\theta^{*(t)}\}$.

$$\begin{aligned} \tilde{E}(\theta|\boldsymbol{X}) &= \bar{\theta}^* = \frac{\sum_{t=1}^{T} \theta^{*(t)}}{T} \\ \tilde{V}(\theta|\boldsymbol{X}) &= \frac{\sum_{t=1}^{T} (\theta^{*(t)} - \bar{\theta^*})^2}{T-1} \\ 95\%\ CI\ \text{for}\ \theta|\boldsymbol{X} &: [q_{.025}(\boldsymbol{\theta}^*), q_{.975}(\boldsymbol{\theta}^*)] \end{aligned}$$

where $q_\alpha(\boldsymbol{x})$ represents the α^{th} quantile of the draws in $\boldsymbol{x}$.

For 3, we need to first set up the M-H algorithm.

We will use a random walk chain, where $\epsilon \sim N(0, \sigma_\epsilon^2)$, as in, $\theta^* = \theta^{*(t)} + \epsilon$.

Next we need to define the transition probability $\alpha\left(\theta^{*(t)}, \theta^*\right)$ at iteration $t+1$. Because of the symmetric nature of the candidate-generating distribution, we do not need $q\left(\theta^{*(t)}, \theta^*\right)$ and $q\left(\theta^*, \theta^{*(t)}\right)$ because they will cancel each other.

In this example,

$$q\left(\theta^{*(t)}, \theta^*\right) \propto \exp\left[-\frac{1}{2}\frac{(\theta^* - \theta^{*(t)})^2}{\sigma_\epsilon}\right]$$

and

$$q\left(\theta^*, \theta^{*(t)}\right) \propto \exp\left[-\frac{1}{2}\frac{(\theta^{*(t)} - \theta^*)^2}{\sigma_\epsilon}\right],$$

which are equal to each other.

Thus,

$$\begin{aligned} \alpha\left(\theta^{*(t)}, \theta^*\right) &= \min\left\{\frac{p\left(\theta^*|\boldsymbol{X}\right)}{p\left(\theta^{*(t)}|\boldsymbol{X}\right)}, 1\right\} \\ &= \min\left\{\frac{\frac{1}{\sqrt{2\pi}\sigma}\exp\left[-\frac{1}{2}\sum_{i=1}^{N}\frac{(X_i-\theta^*)^2}{\sigma^2}\right] \times \frac{1}{\sqrt{2\pi}\tau_0}\exp\left[-\frac{1}{2}\frac{(\theta^*-\mu_0)^2}{\tau^2}\right]}{\frac{1}{\sqrt{2\pi}\sigma}\exp\left[-\frac{1}{2}\sum_{i=1}^{N}\frac{(X_i-\theta^{*(t)})^2}{\sigma^2}\right] \times \frac{1}{\sqrt{2\pi}\tau_0}\exp\left[-\frac{1}{2}\frac{(\theta^{*(t)}-\mu_0)^2}{\tau^2}\right]}, 1\right\} \\ &= \min\left\{\frac{\exp\left[-\frac{1}{2}\sum_{i=1}^{N}\frac{(X_i-\theta^*)^2}{\sigma^2}\right] \times \exp\left[-\frac{1}{2}\frac{(\theta^*-\mu_0)^2}{\tau^2}\right]}{\exp\left[-\frac{1}{2}\sum_{i=1}^{N}\frac{(X_i-\theta^{*(t)})^2}{\sigma^2}\right] \times \exp\left[-\frac{1}{2}\frac{(\theta^{*(t)}-\mu_0)^2}{\tau^2}\right]}, 1\right\} \end{aligned}$$

As can be seen from above, when using M-H, we only need the posterior distribution to up a proportionality constant. In addition, with symmetric candidate-generating distribution $q(\theta^{(t)}, \theta^*) = q(\theta^*, \theta^{(t)})$, we only need to compute the $p(\boldsymbol{X}|\theta) \times p(\theta)$.

We will accept θ^*, as in, set $\theta^{*(t+1)} = \theta^*$ if $\alpha\left(\theta^{*(t)}, \theta^*\right) > u$; otherwise, $\theta^{*(t+1)} = \theta^{*(t)}$.

Let m be the burn-in. We need to sample $T + m$ draws using the M-H algorithm. Call the sample $\boldsymbol{\theta}^{**} = \{\theta^{**(t)}\}$, where $t = 1, \ldots, T + m$.

From this sample, discard the first m draws. Call the remaining sampled draws $\boldsymbol{\theta}^{*} = \{\theta^{**(t+m)}\} = \{\theta^{*(t)}\}$, where $t = 1, \ldots, T + m$.

Once we have $\boldsymbol{\theta}^{*}$, proceed in the same manner as in 2.

MCMC Implementation in Ox for a Univariate Normal with Unknown Mean

The program `uniN.ox` will makes inferences about the unknown μ in $N(\mu, \sigma^2)$. Inferences are made in three different ways: analytically, by direct sampling, and by M-H sampling.

```
#include <oxstd.h>
#include <oxprob.h>
#include <oxfloat.h>
```

We need to declare a few global variables. Except for `X`, these variables can be manipulated by the user.

```
decl N=1000; //sample size
decl mu0=0,tau20=1; //prior parameters
decl mu=0.5,sigma2=2; //parameters for X|mu
decl T=10000; //size of sample from the posterior
decl m=5000; //burn-in
decl X; //data
```

The function `MH_N` samples from the posterior using the M-H algorithm. The argument `s0` is the current value $\theta^{*(t)}$ of the chain.

```
MH_N(decl s0){//s0 is the current value
decl s1=.1*rann(1,1)+s0;//sampling candidate value from N(s0,.1^2)
decl ll0=sumc(-.5*(X-s0).^2/sigma2)+(-.5*(s0-mu0)^2/tau20);//posterior when par=s0
decl ll1=sumc(-.5*(X-s1).^2/sigma2)+(-.5*(s1-mu0)^2/tau20);//posterior when par=s1
decl alpha=exp(ll1-ll0);//acceptance probability
if(alpha>ranu(1,1)) s0=s1;//decide to accept sampled value s1 or keep s0
return s0;
}//end MH_N
```

In the `main` function, we set the random seed, and declare a few local variables.

```
ranseed(120);
decl smp; //sample from the posterior
decl muN,tau2N; //posterior parameters
decl m_est,t2_est; //est of mu_N and tau2N
decl LL,UL; //lower and upper limits
```

The next code generates the data `X`, and solves μ_N and τ_N^2 analytically.

```
X=rann(N,1)*sqrt(sigma2)+mu;//generating data
tau2N=1/(1/tau20+N/sigma2);//analytic posterior variance
muN=(mu0/tau20+N/sigma2*meanc(X))*tau2N;//analytic posterior mean
```

For the `analytic` solution, we use μ_N, τ_N^2, and $\mu_N \pm 1.96\tau_N$ as our estimates.

```
print("\nAnalytic Solution");
m_est=muN;
t2_est=tau2N;
LL=muN-1.96*sqrt(t2_est);
UL=muN+1.96*sqrt(t2_est);
print("\nPosterior Mean and Variance: ",m_est~t2_est);
print("95% Credibility Interval: ",LL~UL);
```

For the *direct-sampling* solution, we first sample `smp` from $N(\mu_N, \tau_N^2)$. We then estimate the parameters of interest using their sample counterparts.

```
print("\nSolution Based on Direct Sampling");
smp=rann(T,1)*sqrt(tau2N)+muN;
m_est=meanc(smp);
t2_est=varc(smp);
LL=quantilec(smp,.025);
UL=quantilec(smp,.975);
print("\nPosterior Mean and Variance: ",m_est~t2_est);
print("95% Credibility Interval: ",LL~UL);
```

For the *indirect-sampling* solution, we use $\bar{x}$ as the initial value; `smp` is set to a null vector. We then write a loop to sample indirectly from the posterior using the function `MH_N`.

We then drop the burn-in (i.e., the first m draws) from `smp`. The remaining steps are identical to the solution above.

```
print("\nSolution Based on Indirect Sampling");
decl smp0=meanc(X);
smp=<>;
for(decl t=0;t<T+m;t++){//M-H step
smp0=MH_N(smp0);
smp=smp|smp0;
}//end t
smp=smp[m:];
m_est=meanc(smp);
t2_est=varc(smp);
LL=quantilec(smp,.025);
UL=quantilec(smp,.975);
print("\nPosterior Mean and Variance: ",m_est~t2_est);
print("95% Credibility Interval: ",LL~UL);
```

Change the values of N, μ_0, τ_0^2, μ, σ^2, T, and m to see how it affects the results.

Multivariate Normal Distribution

Suppose $\boldsymbol{X}|\boldsymbol{\theta} \sim N(\boldsymbol{\theta}, \boldsymbol{\Sigma})$, where $\boldsymbol{\Sigma}$ is known. We also assume the prior distribution $\boldsymbol{\theta} \sim N(\boldsymbol{\mu}_0, \boldsymbol{\Lambda}_0)$. Both $\boldsymbol{\Sigma}$ and $\boldsymbol{\Lambda}_0$ need to be positive definite.

In the multivariate normal, the prior and likelihood are written as:

$$\begin{aligned} p(\boldsymbol{\theta}) &\propto \exp\left[-\frac{1}{2}(\boldsymbol{\theta}-\boldsymbol{\mu}_0)'\boldsymbol{\Lambda}_0^{-1}(\boldsymbol{\theta}-\boldsymbol{\mu}_0)\right] \\ p(\boldsymbol{X}|\theta) &\propto \exp\left[-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\theta})'\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\theta})\right]. \end{aligned}$$

What we need is the posterior distribution $p(\boldsymbol{\theta}|\boldsymbol{X})$.

Again, because a multivariate normal distribution is considered a conjugate prior for a normal likelihood, we are assured that $p(\boldsymbol{\theta}|\boldsymbol{X})$ is also normal.

This means that we just need to find the posterior mean $\boldsymbol{\mu}_1$ and variance $\boldsymbol{\Lambda}_1$.

Again,

$$\begin{aligned} p(\boldsymbol{\theta}|\boldsymbol{X}) &\propto p(\boldsymbol{X}|\boldsymbol{\theta}) \times p(\boldsymbol{\theta}) \\ &= \exp\left\{-\frac{1}{2}\left[(\boldsymbol{X}-\boldsymbol{\theta})'\boldsymbol{\Sigma}^{-1}(\boldsymbol{X}-\boldsymbol{\theta}) + (\boldsymbol{\theta}-\boldsymbol{\mu}_0)'\boldsymbol{\Lambda}_0^{-1}(\boldsymbol{\theta}-\boldsymbol{\mu}_0)\right]\right\}. \end{aligned}$$

We need to simplify the exponent so the posterior will be of the form

$$p(\boldsymbol{\theta}|\boldsymbol{X}) \propto \exp\left[-\frac{1}{2}(\boldsymbol{\theta}-\boldsymbol{\mu}_1)'\boldsymbol{\Lambda}_1^{-1}(\boldsymbol{\theta}-\boldsymbol{\mu}_1)\right].$$

To get the desired form, we need to expand the terms in the exponent.

$$\begin{aligned} p(\boldsymbol{\theta}|\boldsymbol{X}) &\propto \exp\left[-\frac{1}{2}\left(\boldsymbol{X}'\boldsymbol{\Sigma}^{-1}\boldsymbol{X} - 2\boldsymbol{X}'\boldsymbol{\Sigma}^{-1}\boldsymbol{\theta} + \boldsymbol{\theta}'\boldsymbol{\Sigma}^{-1}\boldsymbol{\theta} + \boldsymbol{\theta}'\boldsymbol{\Lambda}_0^{-1}\boldsymbol{\theta} - 2\boldsymbol{\theta}'\boldsymbol{\Lambda}_0^{-1}\boldsymbol{\mu}_0 + \boldsymbol{\mu}_0'\boldsymbol{\Lambda}_0^{-1}\boldsymbol{\mu}_0\right)\right] \\ &\propto \exp\left[-\frac{1}{2}\left(-2\boldsymbol{X}'\boldsymbol{\Sigma}^{-1}\boldsymbol{\theta} + \boldsymbol{\theta}'\boldsymbol{\Sigma}^{-1}\boldsymbol{\theta} + \boldsymbol{\theta}'\boldsymbol{\Lambda}_0^{-1}\boldsymbol{\theta} - 2\boldsymbol{\theta}'\boldsymbol{\Lambda}_0^{-1}\boldsymbol{\mu}_0\right)\right] \\ &= \exp\left\{-\frac{1}{2}\left[\boldsymbol{\theta}'\left(\boldsymbol{\Lambda}_0^{-1} + \boldsymbol{\Sigma}^{-1}\right)\boldsymbol{\theta} - 2\boldsymbol{\theta}'\left(\boldsymbol{\Lambda}_0^{-1}\boldsymbol{\mu}_0 + \boldsymbol{\Sigma}^{-1}\boldsymbol{X}\right)\right]\right\} \end{aligned}$$

Why is $\boldsymbol{X}'\boldsymbol{\Sigma}^{-1}\boldsymbol{\theta} = \boldsymbol{\theta}'\boldsymbol{\Sigma}^{-1}\boldsymbol{X}$?

To complete the square, let

$$\begin{aligned} \boldsymbol{\Lambda}_1^{-1} &= \boldsymbol{\Lambda}_0^{-1} + \boldsymbol{\Sigma}^{-1} \\ \boldsymbol{\mu}_1 &= \boldsymbol{\Lambda}_1\left(\boldsymbol{\Lambda}_0^{-1}\boldsymbol{\mu}_0 + \boldsymbol{\Sigma}^{-1}\boldsymbol{x}\right). \end{aligned}$$

Then

$$\begin{aligned} p(\boldsymbol{\theta}|\boldsymbol{X}) &\propto \exp\left[-\frac{1}{2}\left(\boldsymbol{\theta}'\boldsymbol{\Lambda}_1^{-1}\boldsymbol{\theta} - 2\boldsymbol{\theta}'\boldsymbol{\Lambda}_1^{-1}\boldsymbol{\mu}_1\right)\right] \\ &= \exp\left[-\frac{1}{2}\left(\boldsymbol{\theta}'\boldsymbol{\Lambda}_1^{-1}\boldsymbol{\theta} - 2\boldsymbol{\theta}'\boldsymbol{\Lambda}_1^{-1}\boldsymbol{\mu}_1 + \boldsymbol{\mu}_1'\boldsymbol{\Lambda}_1^{-1}\boldsymbol{\mu}_1\right) + c\right] \\ &\propto \exp\left[-\frac{1}{2}\left(\boldsymbol{\theta} - \boldsymbol{\mu}_1\right)'\boldsymbol{\Lambda}_1^{-1}\left(\boldsymbol{\theta} - \boldsymbol{\mu}_1\right)\right]. \end{aligned}$$

Therefore,

$$\begin{aligned} \boldsymbol{\theta}|\boldsymbol{X} &\sim N(\boldsymbol{\mu}_1, \boldsymbol{\Lambda}_1) \\ &= N\left(\left(\boldsymbol{\Lambda}_0^{-1} + \boldsymbol{\Sigma}^{-1}\right)^{-1}\left(\boldsymbol{\Lambda}_0^{-1}\boldsymbol{\mu}_0 + \boldsymbol{\Sigma}^{-1}\boldsymbol{X}\right), \left(\boldsymbol{\Lambda}_0^{-1} + \boldsymbol{\Sigma}^{-1}\right)^{-1}\right) \end{aligned}$$

If we have N independent draws from $\boldsymbol{x}_i|\boldsymbol{\theta} \sim N(\boldsymbol{\theta}, \boldsymbol{\Sigma})$, show that the posterior can be written as

$$\begin{aligned} \boldsymbol{\theta}|\mathbf{X} &\sim N(\boldsymbol{\mu}_N, \boldsymbol{\Lambda}_N) \\ &= N\left(\left(\boldsymbol{\Lambda}_0^{-1} + N\boldsymbol{\Sigma}^{-1}\right)^{-1}\left(\boldsymbol{\Lambda}_0^{-1}\boldsymbol{\mu}_0 + N\boldsymbol{\Sigma}^{-1}\bar{\boldsymbol{X}}\right), \left(\boldsymbol{\Lambda}_0^{-1} + N\boldsymbol{\Sigma}^{-1}\right)^{-1}\right) \end{aligned}$$

What does it mean for the prior to be uninformative (i.e., what should $\boldsymbol{\Lambda}_0$ be)?

Let $\boldsymbol{X}_{2\times 1} \sim N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, where $\boldsymbol{\Sigma}$ is known. Suppose $\mathbf{X}_{N\times 2}$, and we are interested in estimating $E(\boldsymbol{\theta}|\mathbf{X})$, $V(\boldsymbol{\theta}|\mathbf{X})$, and 95% CIs for $\theta_1|\mathbf{X}$ and $\theta_2|\mathbf{X}$. (Note that these are *marginal* (i.e., individual) CIs, not *joint* (i.e., simultaneous) credibility ellipse.)

As in the univariate case, we have at least three ways of solving the problem: analytically, sampling directly from the posterior, and sampling indirectly from the posterior.

For the analytic solution:

$$\begin{aligned} \tilde{E}(\boldsymbol{\theta}|\mathbf{X}) &= \boldsymbol{\mu}_N \\ \tilde{V}(\boldsymbol{\theta}|\mathbf{X}) &= \boldsymbol{\Lambda}_N \\ 95\%\ CI\ \text{for}\ \theta_d|\mathbf{X} &: \mu_{Nd} \pm 1.96 \times \sqrt{\lambda_{Ndd}} \end{aligned}$$

for $d = 1, 2$, and where λ_{Ndd} is the d^{th} diagonal element of $\mathbf{\Lambda}_N$.

For the direct sampling solution:

We sample T draws from $N(\boldsymbol{\mu}_N, \mathbf{\Lambda}_N)$, and call it $\mathbf{\Theta}^* = \{\boldsymbol{\theta}^{*(t)}\}$.

$$\begin{aligned}
\tilde{E}(\boldsymbol{\theta}|\mathbf{X}) &= \bar{\boldsymbol{\theta}}^* = \frac{\sum_{t=1}^{T} \boldsymbol{\theta}^{*(t)}}{T} \\
\tilde{V}(\boldsymbol{\theta}|\mathbf{X}) &= \frac{\sum_{t=1}^{T} \left(\boldsymbol{\theta}^{*(t)} - \bar{\boldsymbol{\theta}^*}\right)\left(\boldsymbol{\theta}^{*(t)} - \bar{\boldsymbol{\theta}^*}\right)'}{T-1} \\
95\%\ CI\ \text{for}\ \theta_d|\mathbf{X} &: [q_{.025}(\boldsymbol{\theta}_d^*), q_{.975}(\boldsymbol{\theta}_d^*)]
\end{aligned}$$

for $d = 1, 2$, and where $\boldsymbol{\theta}_d^*$ is the vector of sampled draws for dimension d.

For the indirect sampling solution, we need to first set up the M-H algorithm.

We will again use a random walk chain and symmetric candidate generating distribution, as in $\theta_d^* = \theta_d^{*(t)} + \epsilon_d$, where $\epsilon_d \sim N(0, \sigma_\epsilon^2)$. Even though $\boldsymbol{\theta}$ is multivariate, and possibly correlated, it is acceptable to use univariate and independent candidate generating distributions.

Next we need to define the transition probability $\alpha\left(\theta^{*(t)}, \theta^*\right)$ at iteration $t+1$.

Thus,

$$\begin{aligned}
\alpha\left(\boldsymbol{\theta}^{*(t)}, \boldsymbol{\theta}^*\right) &= \min\left\{\frac{p\left(\boldsymbol{\theta}^*|\mathbf{X}\right)}{p\left(\boldsymbol{\theta}^{*(t)}|\mathbf{X}\right)}, 1\right\} \\
&= \min\left\{\frac{\exp\left[-\frac{1}{2}\sum_{i=1}^{N}\left(\boldsymbol{X}_i - \boldsymbol{\theta}^*\right)'\mathbf{\Sigma}^{-1}\left(\boldsymbol{X}_i - \boldsymbol{\theta}^*\right) - \frac{1}{2}\left(\boldsymbol{\theta}^* - \boldsymbol{\mu}_0\right)'\mathbf{\Lambda}_0^{-1}\left(\boldsymbol{\theta}^* - \boldsymbol{\mu}_0\right)\right]}{\exp\left[-\frac{1}{2}\sum_{i=1}^{N}\left(\boldsymbol{X}_i - \boldsymbol{\theta}^{*(t)}\right)'\mathbf{\Sigma}^{-1}\left(\boldsymbol{X}_i - \boldsymbol{\theta}^{*(t)}\right) - \frac{1}{2}\left(\boldsymbol{\theta}^{*(t)} - \boldsymbol{\mu}_0\right)'\mathbf{\Lambda}_0^{-1}\left(\boldsymbol{\theta}^{*(t)} - \boldsymbol{\mu}_0\right)\right]}, 1\right\}
\end{aligned}$$

The ratio involves the term of the form $\sum_{i=1}^{N} \boldsymbol{y}_i'\mathbf{V}\boldsymbol{y}_i$ in the exponent. Computationally, this can be expensive to implement as it would require writing a `for` loop that computes $\boldsymbol{y}'\mathbf{V}\boldsymbol{y}$ one observation at a time because of the two matrix multiplications involved.

As an alternative, we can use Cholesky decomposition to get $\mathbf{V} = \mathbf{C}\mathbf{C}'$, where $\mathbf{C}$ is a lower triangular matrix.

This allows us to write

$$\begin{aligned}
\boldsymbol{y}'\mathbf{V}\boldsymbol{y} &= \boldsymbol{y}'\mathbf{C}\mathbf{C}'\boldsymbol{y} \\
&= (\mathbf{C}'\boldsymbol{y})'\mathbf{C}'\boldsymbol{y} \\
&= \boldsymbol{y}^{*\prime}\boldsymbol{y}^* \\
&= \sum_{d=1}^{D} y_d^{*2}
\end{aligned}$$

This means that, instead of two the matrix multiplications in $\boldsymbol{y}'\mathbf{V}\boldsymbol{y}$, we only need to carry out one matrix multiplication (i.e., $\mathbf{C}'\boldsymbol{y}$), and then sum the square of the resulting terms.

In Ox, this means you we can now use `sumr` and `sumc` without using a `for` loop.

Note that $\alpha\left(\boldsymbol{\theta}^{*(t)}, \boldsymbol{\theta}^{*}\right)$ is a scalar so we will accept or reject $\boldsymbol{\theta}^{*}$ en bloc.

Set $\boldsymbol{\theta}^{*(t+1)} = \boldsymbol{\theta}^{*}$ if $\alpha\left(\boldsymbol{\theta}^{*(t)}, \boldsymbol{\theta}^{*}\right) > u$; otherwise, $\boldsymbol{\theta}^{*(t+1)} = \boldsymbol{\theta}^{*(t)}$.

Again, let m be the burn-in. We need to sample $T + m$ draws using the M-H algorithm. Call the sample $\boldsymbol{\Theta}^{**} = \{\boldsymbol{\theta}^{**(t)}\}$, where $t = 1, \ldots, T + m$.

From this sample, discard the first m draws. Call the remaining sampled draws $\boldsymbol{\Theta}^{*} = \{\boldsymbol{\theta}^{**(t+m)}\} = \{\boldsymbol{\theta}^{*(t)}\}$, where $t = 1, \ldots, T + m$.

As in the univariate case, once we have $\boldsymbol{\Theta}^{*}$, we can proceed as above.

Because $\boldsymbol{\theta}$ is multivariate, we actually have a **fourth** option - to use Gibbs sampling.

Note that Gibbs sampling is particularly useful if we do not know how to sample $\boldsymbol{\theta}|\mathbf{X}$, which is definitely not the case here. We still do it just because it's fun!

To implement Gibbs sampling, we need to find the *full conditional distributions* $p(\theta_1|\mathbf{X}, \theta_2)$ and $p(\theta_2|\mathbf{X}, \theta_1)$.

We already know the posterior distribution $p(\boldsymbol{\theta}|\mathbf{X})$, which is $N(\boldsymbol{\mu}_N, \boldsymbol{\Lambda}_N)$. By partitioning $\boldsymbol{\theta}$, we get the following (full) conditional distributions:

$$\begin{aligned} p(\theta_1|\mathbf{X}, \theta_2) &= N\left(\mu_{N1} + \frac{\lambda_{N12}}{\lambda_{N22}}(\theta_2 - \mu_{N2}), \lambda_{N11} - \frac{\lambda_{N12}^2}{\lambda_{N22}}\right) \\ p(\theta_2|\mathbf{X}, \theta_1) &= N\left(\mu_{N2} + \frac{\lambda_{N12}}{\lambda_{N11}}(\theta_1 - \mu_{N1}), \lambda_{N22} - \frac{\lambda_{N12}^2}{\lambda_{N11}}\right) \end{aligned}$$

This gives us two univariate distributions, which we know how to sample from directly. If we do not know how to sample directly from one or more of these distributions, we can use A-R or M-H.

For this example, we will use direct sampling. Because we are using a Markov chain, we need to sample $T + m$ draws from the posterior. We accomplish this by sampling iteratively from the full conditional distributions.

At iteration $t + 1$:

- Draw $\theta_1^{(t+1)}$ from $N\left(\mu_{N1} + \frac{\lambda_{N12}}{\lambda_{N22}}(\theta_2^{(t)} - \mu_{N2}), \lambda_{N11} - \frac{{\lambda_{N12}}^2}{\lambda_{N22}}\right)$
- Then, draw $\theta_2^{(t+1)}$ from $N\left(\mu_{N2} + \frac{\lambda_{N12}}{\lambda_{N11}}(\theta_1^{(t+1)} - \mu_{N1}), \lambda_{N22} - \frac{{\lambda_{N12}}^2}{\lambda_{N11}}\right)$

Because we employ direct sampling, all draws are accepted. However, we still need to discard the burn-in from $\boldsymbol{\Theta}^{**}$, and based our inferences on $\boldsymbol{\Theta}^{*}$ only.

MCMC Implementation in Ox for a Multivariate Normal with Unknown Mean Vector

The program `multiN.ox` provides a few inferences about unknown $\boldsymbol{mu}_{2\times 1}$ in $N(\boldsymbol{mu}, \boldsymbol{\Lambda}$. The solutions are derived analytically, and using direct sampling of the posterior, M-H algorithm, and Gibbs sampler.

We use the same headers as `uniN.ox`

The global variables the multivariate counterparts of the univariate global variables.

```
decl N=1000; //sample size
decl mu0=<0;0>,Lambda0=<1,0;0,1>; //prior parameters
decl mu=<.5;-1>,Sigma=<1,.5;.5,1>; //parameters for X|mu
decl T=10000; //size of sample from the posterior
decl m=5000; //burn-in
decl X; //data
```

The function `MH_N` generalizes to accommodate the multivariate nature of $\boldsymbol{\mu}$. For example, we get the initial draw from `rann(2,1)`, and `sumr(((X-s0')*choleski(invert(Sigma))).^2)` computes $(\boldsymbol{x}_i - \boldsymbol{\theta}^{*(t)})'\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}_i - \boldsymbol{\theta}^{*(t)})$.

```
MH_N(decl s0){//s0 is the current value
decl s1=.1*rann(2,1)+s0;//sampling candidate value from N(s0,.1^2)
decl ll0=sumr(((X-s0')*choleski(invert(Sigma))).^2);
ll0=-.5*sumc(ll0)+(-.5*(s0-mu0)'invert(Lambda0)*(s0-mu0));//post. when par=s0
decl ll1=sumr(((X-s1')*choleski(invert(Sigma))).^2);
ll1=-.5*sumc(ll1)+(-.5*(s1-mu0)'invert(Lambda0)*(s1-mu0));//post. when par=s1
decl alpha=exp(ll1-ll0);//acceptance probability
if(alpha>ranu(1,1)) s0=s1;//decide to accept sampled value s1 or keep s0
return s0;
}//end MH_N
```

The function `Gibbs` is new to this program. It first computes the mean and variance of the full conditional $p(\theta_1|\mathbf{X}, \theta_2)$, before sampling from this distribution. The same this is done for $\theta_2|\mathbf{X}, \theta_1$.

```
Gibbs(decl s0,M,L){//s0=value, M=post mean, L=post var
decl m=M[0]+L[0][1]/L[1][1]*(s0[1]-M[1]);//mean of par1
decl v=L[0][0]-L[0][1]^2/L[1][1];//variance of par1
s0[0]=rann(1,1)*sqrt(v)+m;//sampling par1 given par 2
m=M[1]+L[0][1]/L[0][0]*(s0[0]-M[0]);//mean of par2
v=L[1][1]-L[0][1]^2/L[0][0];//variance of par2
s0[1]=rann(1,1)*sqrt(v)+m;//sampling par2 given par 1
return s0;
}//end Gibbs
```

The `main` function is largely similar to the original `main` function accept for this time we are making sure that the dimensions permit matrix operations, and a code is added for the Gibbs sampler solution.

Change the values of N, $\boldsymbol{\mu}_0$, $\boldsymbol{\Lambda}_0$, $\boldsymbol{\mu}$, $\boldsymbol{\Sigma}$, T, and m to see how it affects the results.

MCMC Estimation of Ability

In this example, let N and J be the number of examinees, and test length, respectively, and $P_{ij} = P(X_j = 1|\theta_i)$ be the 2PL. Also, let $p(\theta_i) \sim N(0, 1)$. The goal is to find $p(\theta_i|\boldsymbol{X}_i)$, for $i = 1, \ldots, N$.

Specifically, we are interested in $E(\theta_i|\boldsymbol{X}_i)$ and $V(\theta_i|\boldsymbol{X}_i)$.

To start, we need to find the full conditional distribution for each examinee.

The joint posterior is $p(\mathbf{X}, \boldsymbol{\theta})$, and the posterior distribution is

$$\begin{aligned} p(\boldsymbol{\theta}|\mathbf{X}) &\propto p(\mathbf{X}|\boldsymbol{\theta}) \times p(\boldsymbol{\theta}) \\ &= \prod_{i=1}^{N} p(\boldsymbol{X}_i|\theta_i) \times p(\theta_i) \\ &= \prod_{i=1}^{N} \left[\prod_{j=1}^{J} P_{ij}^{X_{ij}} (1 - P_{ij})^{1-X_{ij}} \right] \times p(\theta_i) \end{aligned}$$

Let

$$\begin{aligned} \mathbf{X}^{(-i)} &= (\boldsymbol{X}_1, \ldots, \boldsymbol{X}_{i-1}, \boldsymbol{X}_{i+1}, \ldots, \boldsymbol{X}_N) \\ \boldsymbol{\theta}^{(-i)} &= (\theta_1, \ldots, \theta_{i-1}, \theta_{i+1}, \ldots, \theta_N) \end{aligned}$$

The full conditional distribution for θ_i is

$$\begin{aligned} p(\theta_i|\mathbf{X}, \boldsymbol{\theta}^{(-i)}) &\propto p(\mathbf{X}|\theta_i, \boldsymbol{\theta}^{(-i)}) \times p(\theta_i|\boldsymbol{\theta}^{(-i)}) \\ &= p(\boldsymbol{X}_i|\theta_i) \times p(\mathbf{X}^{(-i)}|\boldsymbol{\theta}^{(-i)}) \times p(\theta_i) \\ &= p(\boldsymbol{X}_i|\theta_i) \times p(\theta_i) \times c \\ &\propto p(\boldsymbol{X}_i|\theta_i) \times p(\theta_i) \end{aligned}$$

Why can we let $p(\mathbf{X}^{(-i)}|\boldsymbol{\theta}^{(-i)}) = c$?

The $p(\theta_i|\mathbf{X}, \boldsymbol{\theta}^{-i})$ indicates that, as far as examinee i, is concerned, we only need to deal with $\boldsymbol{X}_i$. This dramatically simplifies the problem because the original N-dimensional problem can be solved one dimension at a time.

However, although $p(\theta_i|\mathbf{X}, \boldsymbol{\theta}^{-i})$ is unidimensional, we actually do not know how to sample from this distribution. Therefore, we will use M-H.

To start, at iteration $t + 1$, the candidate value will be drawn as $\theta_i^* \sim N(\theta_i^{(t)}, \sigma_\epsilon^2)$.

The acceptance probability is

$$\alpha(\theta_i^{(t)}, \theta_i^*) = \min\left\{ \frac{p(\boldsymbol{X}_i|\theta_i^*) \times p(\theta_i^*)}{p(\boldsymbol{X}_i|\theta_i^{(t)}) \times p(\theta_i^{(t)})}, 1 \right\}.$$

We set $\theta_i^{(t+1)} = \theta_i^*$ if $\alpha(\theta_i^{(t)}, \theta_i^*) > u$; otherwise $\theta_i^{(t+1)} = \theta_i^{(t)}$.

An Important Note: Although we deal with the examinees one at a time, it does not mean we need to also do the different steps of the M-H algorithm *separately* (i.e., write a loop for the N examinees). For greater efficiency, they can be done *simultaneously*.

In this example, the M-H code can be written such that we should be able to *simultaneously* sample $\boldsymbol{\theta}^*$, *simultaneously* evaluate the acceptance probabilities, and *simultaneously* determine $\boldsymbol{\theta}^{(t+1)}$, and still ensure that the elements of $\boldsymbol{\theta}^{(t+1)}$ are updated *separately*, rather than *en bloc*.

Also, for this problem, saving all the draws means we have to store a $T + m \times N$ matrix of draws. This may not be an issue at this point, but other applications that deal with multidimensional data and require structural parameter estimation, it can dramatically slow down the algorithm.

If we already know what inferences we are interested in, it would suffice to just save the sufficient statistics.

For the $E(\theta|\boldsymbol{X})$ and $V(\theta|\boldsymbol{X})$, the sufficient statistics are

$$\begin{aligned} S_\theta &= \sum_{t=m+1}^{T+m} \theta^{(t)} \\ S_{\theta^2} &= \sum_{t=m+1}^{T+m} {\theta^{(t)}}^2 \end{aligned}$$

Given the sufficient statistics,

$$\begin{aligned} \tilde{E}(\theta|\boldsymbol{X}) &= S_\theta / T \\ \tilde{V}(\theta|\boldsymbol{X}) &= \frac{S_{\theta^2} - T \times \tilde{E}^2(\theta|\boldsymbol{X})}{T-1} \end{aligned}$$

In practice, we do not need to get all the $T + m$ draws before computing S_θ and S_{θ^2}. Instead, we can keep a running total of S_θ and S_{θ^2} at each iteration after the burn-in period.

This mean that instead of a $T + m \times N$ matrix, we need to only keep track of an $N \times 2$ matrix.

Note that the latter is not a function of T, so the number of iterations does not affect the size of the matrix that needs to be stored in the memory. In some application, this can dramatically affect the efficiency of the algorithm.

MCMC Implementation in Ox for a Finding the EAP of the Ability

The program `eap_theta.ox` estimates $E(\theta_i|\boldsymbol{X}_i)$ and $V(\theta_i|\boldsymbol{X}_i)$ using MCMC, specifically M-H algorithm, for the 2PL. In addition to the estimates, it also evaluates how well the parameters are estimated.

We start with the basic headers.

```
#include <oxstd.h>
#include <oxprob.h>
#include <oxfloat.h>
```

Next, we define the global variables. These variables are defined globally to make the implementation of the M-H algorithm more convenient.

```
decl N=500,J=10; //N=sample size, J=test length
decl T=5000; //size of sample from the posterior
decl m=1000; //burn-in
decl theta,alpha,beta,X; //ability, item parameters,data
decl theta0; //current value of theta
decl s_theta,s_theta2; //sufficient statistics
```

The function `TwoPL` computes the 2PL $N \times J$ probabilities given the $\boldsymbol{\theta}$, $\boldsymbol{\alpha}$, and $\boldsymbol{\beta}$.

```
TwoPL(decl t,a,b){
decl tmp=exp(1.7*t*a'-ones(N,1)*(a.*b)');
return(tmp./(1+tmp));
}//TwoPL
```

The function `MH_draw()` updates `theta0`. Note that drawing $\boldsymbol{\theta}^*$, evaluating the acceptance probabilities, and determining which elements of `theta0` need to be updated is done simultaneously for all the N examinees. The use of `vecindex` allows us to selectively update the elements `theta0`.

```
MH_draw(){
decl theta1=theta0+.25*rann(N,1);
decl tmp=TwoPL(theta0,alpha,beta);
decl ll0=sumr(X.*log(tmp)+(1-X).*log(1-tmp))+log(densn(theta0));
tmp=TwoPL(theta1,alpha,beta);
decl ll1=sumr(X.*log(tmp)+(1-X).*log(1-tmp))+log(densn(theta1));
decl acc=exp(ll1-ll0);//acceptance probability
decl ind=vecindex(acc.>ranu(N,1));
theta0[ind][]=theta1[ind][];
}//end MH_draw
```

The function `initiate` makes sure that the variables are set up properly. When multiple chains are involved, this becomes more crucial. Of course, we may need to add some noise to the initial value of `theta0`. Otherwise, the different chains will all have the same starting values.

```
initiate(){
theta0=quann(.025+.95*meanr(X));
s_theta=0,s_theta2=0;
}//end initiate
```

The function `update` keeps a running total of S_θ and S_{θ^2}. It is called in the `main` for iterations $t = m+1, \ldots, T+m$.

```
update(){
s_theta=s_theta+theta0;
s_theta2=s_theta2+theta0.^2;
}//end update
```

The first part of the `main` function generates $\boldsymbol{\theta}$, $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, and simulates the $N \times J$ data matrix `X`.

It then calls the `initiate` function. The `i` loop draws from the posterior using `MH_draw`, and calls `update` after the burn-in.

```
initiate();
for(decl i=0;i<T+m;i++){
MH_draw();
if(i>=m) update();
}//end i
```

The last few lines estimate $\tilde{\boldsymbol{\theta}} = \tilde{E}(\boldsymbol{\theta}|\mathbf{X})$ and $\tilde{V}(\boldsymbol{\theta}|\mathbf{X})$, and evaluates the quality of the estimates by computing correlation between the true and estimate $\boldsymbol{\theta}$, mean posterior standard deviation, and root mean squared error:

$$\begin{aligned} \tilde{\rho}_{\boldsymbol{\theta},\tilde{\boldsymbol{\theta}}} &= Corr(\boldsymbol{\theta}, \tilde{\boldsymbol{\theta}}) \\ MPSD &= \sqrt{\sum_{i=1}^{N} \frac{\tilde{V(\theta_i|\boldsymbol{X}_i)}}{N}} \\ RMSE &= \sqrt{\sum_{i=1}^{N} \frac{(\tilde{\theta_i} - \theta_i)^2}{N}} \end{aligned}$$

```
decl est=s_theta/T;
decl tmp=correlation(theta~est);
print("\nCorr(True,Est):    ",tmp[0][1]);
tmp=(s_theta2-T*est.^2)/(T-1);
print("\nMean Posterior SD: ",sqrt(meanc(tmp))[0]);
tmp=sqrt(meanc((theta-est).^2));
print("\nRMSE:              ",tmp[0],"\n");
```

Change N, J, $\boldsymbol{\alpha}$, and $\boldsymbol{\beta}$ to see how they affect the quality of the estimates.

Additional Exercises

1. Monte Carlo Variance

Whenever our inferences are based on draws from the posterior, Markov chain or otherwise, *Monte Carlo* (MC) *errors* are involved. This simply means that we can expect the estimates to vary from one sample to another.

The amount of variability is called the *MC variance*, which we can denote as σ^2_{MC}. We can approximate σ^2_{MC} by drawing S samples, and computing,

$$\sigma^2_{MC} = \frac{\sum_{s=1}^{S}\left[\tilde{E}^{(s)}(\theta|\boldsymbol{X}) - \overline{\tilde{E}(\theta|\boldsymbol{X})}\right]^2}{S},$$

if the variable of interest is $E(\theta|\boldsymbol{X})$, which is usually the case.

Thus, the total uncertainty associated with $\theta|\boldsymbol{X}$ is the posterior variance plus the Monte Carlo variance (i.e., $V(\theta|\boldsymbol{X}) + \sigma^2_{MC}$).

The Monte Carlo variance can represent a substantial proportion of the total uncertainty if the number of draws in a sample is not sufficiently large.

a) Modify `uniN.ox` to allow you to obtain S samples for the same data.

b) Find the σ^2_{MC} associated with the estimates based on direct and indirect samplings procedures.

c) Manipulate T and m to see how they affect σ^2_{MC} for the different sampling procedures.

2. Extend `multiN.ox` to make it applicable to the case $D = 3$.

3. Comparing Different Methods for Estimating the Ability θ

Previously, we learned how to find the MLE $\hat{\theta}$ and EAP $\tilde{\theta}$ through analytic maximization, and the use of quadrature nodes, respectively.

The current method of finding $\tilde{\theta}$ is based on a Monte Carlo method, specifically, MCMC.

The standard errors of $\hat{\theta}$ and $\tilde{\theta}$ are computed from the second derivative and the posterior variance, respectively.

a) Using the correlation between the true and estimated θ, mean standard error, and root mean squared error, compare the three methods of estimating θ and their standard errors.

b) What happens to the estimates and the standard errors as J increases.

Markov Chain Monte Carlo - Part II

An MCMC Algorithm for the 2PL

One advantage of MCMC is the ease by which existing algorithms can be extended to accommodate more complex models.

In this example, we will extend the algorithm for obtaining the EAP of θ in the context of the 2PL to include item parameter estimation.

Again, let N and J be the number of examinees, and test length, respectively, and $P_{ij} = P(X_j = 1|\theta_i)$ be the 2PL.

The goal this time is to make inferences about $p(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\theta}|\mathbf{X})$, where $\boldsymbol{\alpha} = \{\alpha_j\}$, $\boldsymbol{\beta} = \{\beta_j\}$, and $\boldsymbol{\theta} = \{\theta_i\}$.

Prior, Posterior, and Full Conditional Distributions

Let the priors be

$$\begin{aligned} p(\theta_i) &\sim N(0,1) \\ p(\alpha_j) &\sim \log N(0,1) \\ p(\beta_j) &\sim N(0,1) \end{aligned}$$

where $x \sim \log N(\mu, \sigma^2)$ means $\log x \sim N(\mu, \sigma^2)$.

The likelihood of of the data given $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, and $\boldsymbol{\theta}$ is

$$\begin{aligned} L(\mathbf{X}|\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\theta}) &= \prod_{i=1}^{N} L(\boldsymbol{X}_i|\boldsymbol{\alpha}, \boldsymbol{\beta}, \theta_i) \\ &= \prod_{j=1}^{J} L(\boldsymbol{X}_j|\alpha_j, \beta_j, \boldsymbol{\theta}) \\ &= \prod_{i=1}^{N}\prod_{j=1}^{J} P_{ij}^{X_{ij}}(1-P_{ij})^{1-X_{ij}}, \end{aligned}$$

where

$$P_{ij} = P(X = 1|\alpha_j, \beta_j, \theta_i) = \frac{\exp\left[1.7\alpha_j(\theta_i - \beta_j)\right]}{1 + \exp\left[1.7\alpha_j(\theta_i - \beta_j)\right]}.$$

The joint posterior distribution is

$$\begin{aligned} p(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\theta}|\mathbf{X}) &\propto L(\mathbf{X}|\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\theta}) \times p(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\theta}) \\ &= L(\mathbf{X}|\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\theta}) \times p(\boldsymbol{\alpha}) \times p(\boldsymbol{\beta}) \times p(\boldsymbol{\theta}) \\ &= L(\mathbf{X}|\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\theta}) \times \prod_{j=1}^{J} p(\alpha_j) \times \prod_{j=1}^{J} p(\beta_j) \times \prod_{i=1}^{N} p(\theta_j). \end{aligned}$$

To implement Gibss sampling, we need the full conditional distributions, and for this example, we will define the following full distributions: $p(\alpha_j, \beta_j | \mathbf{X}, \boldsymbol{\alpha}^{(-j)}, \boldsymbol{\beta}^{(-j)}, \boldsymbol{\theta})$ for $j = 1, \ldots, J$, and $p(\theta_i | \mathbf{X}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\theta}^{(-i)})$ for $i = 1, \ldots, N$, where

$$\begin{aligned} \boldsymbol{\alpha}^{(-j)} &= \{\alpha_1, \ldots, \alpha_{j-1}, \alpha_{j+1}, \ldots, \alpha_J\} \\ \boldsymbol{\beta}^{(-j)} &= \{\beta_1, \ldots, \beta_{j-1}, \beta_{j+1}, \ldots, \beta_J\} \\ \boldsymbol{\theta}^{(-i)} &= \{\theta_1, \ldots, \theta_{i-1}, \theta_{i+1}, \ldots, \theta_N\} \end{aligned}$$

Based on these formulations, we will update the parameters of item j *en bloc*.

The full conditional distribution of $\{\alpha_j, \beta_j\}$ is

$$\begin{aligned} p(\alpha_j, \beta_j | \mathbf{X}, \boldsymbol{\alpha}^{(-j)}, \boldsymbol{\beta}^{(-j)}, \boldsymbol{\theta}) &\propto L(\mathbf{X} | \alpha_j, \beta_j, \boldsymbol{\alpha}^{(-j)}, \boldsymbol{\beta}^{(-j)}, \boldsymbol{\theta}) \times p(\alpha_j, \beta_j | \boldsymbol{\alpha}^{(-j)}, \boldsymbol{\beta}^{(-j)}, \boldsymbol{\theta}) \\ &\propto L(\boldsymbol{X}_j | \alpha_j, \beta_j, \boldsymbol{\theta}) \times p(\alpha_j, \beta_j) \\ &= L(\boldsymbol{X}_j | \alpha_j, \beta_j, \boldsymbol{\theta}) \times p(\alpha_j) \times p(\beta_j) \end{aligned}$$

What is the second proportionality due to, the first or the second factor?

The full conditional distribution of θ_i is

$$\begin{aligned} p(\theta_i | \mathbf{X}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\theta}^{(-i)}) &\propto L(\mathbf{X} | \boldsymbol{\alpha}, \boldsymbol{\beta}, \theta_i, \boldsymbol{\theta}^{(-i)}) \times p(\theta_i | \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\theta}^{(-i)}) \\ &\propto L(\boldsymbol{X}_i | \boldsymbol{\alpha}, \boldsymbol{\beta}, \theta_i) \times p(\theta_i) \end{aligned}$$

Note that neither $p(\alpha_j, \beta_j | \mathbf{X}, \boldsymbol{\alpha}^{(-j)}, \boldsymbol{\beta}^{(-j)}, \boldsymbol{\theta})$ nor $p(\theta_i | \mathbf{X}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\theta}^{(-i)})$ is a known distribution so we would have to use M-H to obtain draws from the full conditional distributions.

At iteration $t + 1$:

For $j = 1, \ldots, J$, draw α_j^* from $N(\alpha_j^{(t)}, \sigma_\alpha^2)$ and β_j^* from $N(\beta_j^{(t)}, \sigma_\beta^2)$.

The acceptance probability $\mathcal{A}$ associated with the parameters of item j is

$$\mathcal{A}(\{\alpha_j^{(t)}, \beta_j^{(t)}\}, \{\alpha_j^*, \beta_j^*\}) = \min \left\{ \frac{L(\boldsymbol{X}_j | \alpha_j^*, \beta_j^*, \boldsymbol{\theta}^{(t)}) \times p(\alpha_j^*) \times p(\beta_j^*)}{L(\boldsymbol{X}_j | \alpha_j^{(t)}, \beta_j^{(t)}, \boldsymbol{\theta}^{(t)}) \times p(\alpha_j^{(t)}) \times p(\beta_j^{(t)})}, 1 \right\}.$$

Set $\{\alpha_j^{(t+1)}, \beta_j^{(t+1)}\} = \{\alpha_j^*, \beta_j^*\}$ if $\mathcal{A}(\{\alpha_j^{(t)}, \beta_j^{(t)}\}, \{\alpha_j^*, \beta_j^*\}) > u$; otherwise, set $\{\alpha_j^{(t+1)}, \beta_j^{(t+1)}\} = \{\alpha_j^{(t)}, \beta_j^{(t)}\}$.

Then for $i = 1, \ldots, N$, draw θ_i^* from $N(\theta_i^{(t)}, \sigma_\theta^2)$.

The acceptance probability $\mathcal{A}$ associated with ability i is

$$\mathcal{A}(\theta_i^{(t)}, \theta_i^*) = \min \left\{ \frac{L(\boldsymbol{X}_i | \boldsymbol{\alpha}^{(t+1)}, \boldsymbol{\beta}^{(t+1)}, \theta_i^*) \times p(\theta_i^*)}{L(\boldsymbol{X}_i | \boldsymbol{\alpha}^{(t+1)}, \boldsymbol{\beta}^{(t+1)}, \theta_i^{(t)}) \times p(\theta_i^{(t)})}, 1 \right\}.$$

Set $\theta_i^{(t+1)} = \theta_i^*$ if $\mathcal{A}(\theta_i^{(t)}, \theta_i^*) > u$; otherwise, set $\theta_i^{(t+1)} = \theta_i^{(t)}$.

Again, although updating of the item parameters will be done one item at a time, we will sample the candidate values simultaneously to make the algorithm more efficient.

Whenever possible, we will sample the draws en bloc even though updating will be done one item or person at a time.

Note that the sampling process for θ_i is essentially identical to the sampling process when the item parameters are fixed.

The only difference is that we are now using $\boldsymbol{\alpha}^{(t+1)}$ and $\boldsymbol{\beta}^{(t+1)}$ instead of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$.

A Detour: The Four-Parameter Beta Distribution

The four-parameter beta random variable is a linear transformation of a beta random variable. If $X \sim \text{Beta}(\upsilon, \omega)$, for $0 < x < 1$, its density function is

$$f(x) = \frac{x^{\upsilon-1}(1-x)^{\omega-1}}{\beta(\upsilon,\omega)},$$

where $\beta(\upsilon,\omega) = \int_0^1 u^{\upsilon-1}(1-u)^{\omega-1}\partial u$. The mean and variance of the distribution are

$$\begin{aligned} E(X) &= \upsilon/(\upsilon+\omega); \\ V(X) &= \upsilon\omega/\{(\upsilon+\omega)^2(\upsilon+\omega+1)\}. \end{aligned}$$

A random variable $Y = (b-a)X + a$ is distributed as 4-Beta(υ, ω, a, b) for $a < b$, and has a density function

$$g(y) = \frac{(y-a)^{\upsilon-1}(b-y)^{\omega-1}}{\beta(\upsilon,\omega)(b-a)^{\upsilon+\omega-1}} \quad \text{for} \quad a < y < b.$$

The corresponding mean and variance are

$$\begin{aligned} E(Y) &= (b-a)\upsilon/(\upsilon+\omega) + a; \\ V(Y) &= (b-a)^2\upsilon\omega/\{(\upsilon+\omega)^2(\upsilon+\omega+1)\}. \end{aligned}$$

Below are some features of the distribution that makes it attractive.

First, the support of the four-parameter beta distribution can be altered by adjusting a and b. Hence, the beta distribution is a special case of the 4-parameter beta distribution (i.e., when $a = 0$ and $b = 1$).

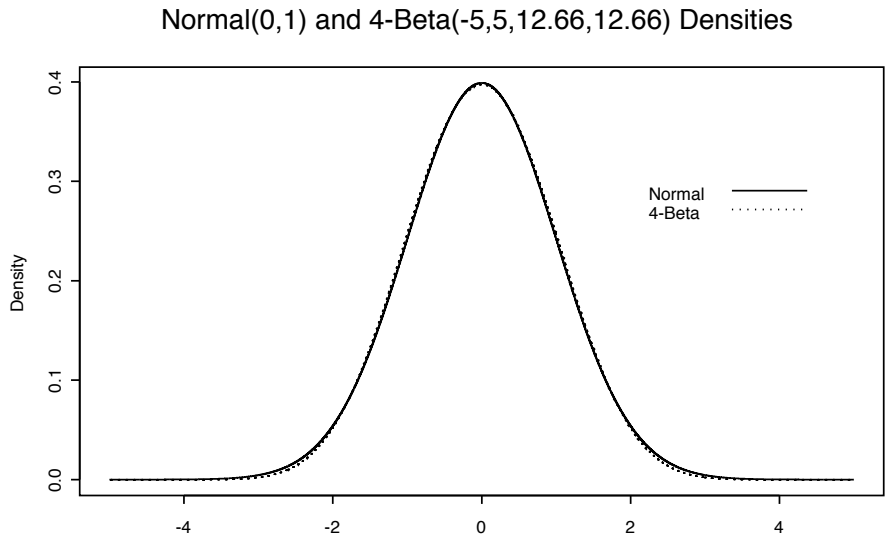
Second, a symmetric or asymmetric distribution can be obtained by varying υ and ω. A symmetric distribution is produced when $\upsilon = \omega$.

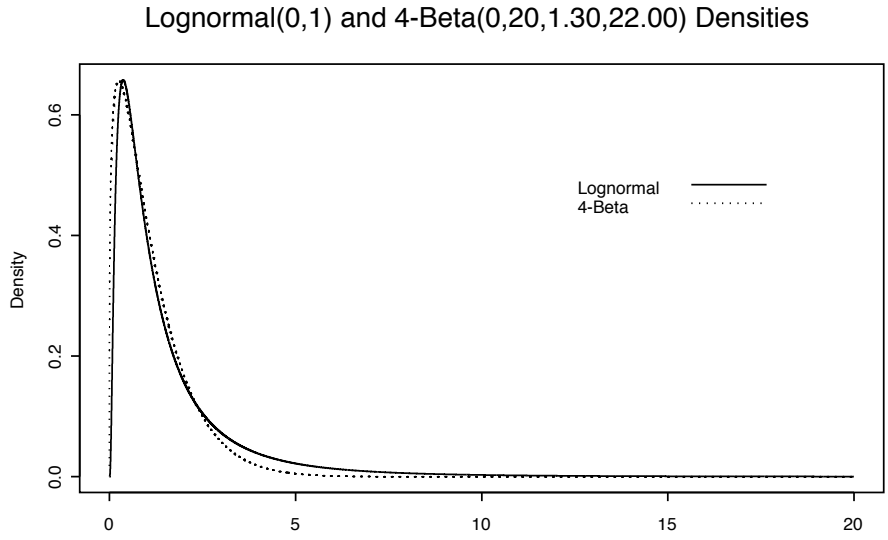
A special case is when $\upsilon = \omega = 1$; it is the uniform distribution over the interval (a, b).

When $\upsilon = \omega \approx 12.66$ and $a = -5$ and $b = 5$, the maximum difference between the standard normal and the 4-parameter beta densities over the interval $(-5, 5)$ is about 0.005.

When $\upsilon < \omega$, the distribution is skewed to the right, and vice versa.

Finally, the variance of the distribution can be modified by changing the magnitudes of υ and ω: larger values correspond to a smaller variances.





The figure below shows the 4-Beta approximations to some of the commonly used prior distributions associated with the item parameters.

In using priors with constrained support in the interval (a, b), we are assured that the parameter estimates will not run away.

For example, if instead of $N(0, 1)$, we use the prior 4-Beta$(-5, 5, \upsilon, \omega)$ for the difficulty parameter, we are guaranteed that $|\tilde{\beta}_j| < 5$.

End of Detour

Implementation in Ox of the MCMC Algorithm for the 2PL

The program `mcmc_2PL.ox` is Bayesian estimation program for estimating the item parameters of the 2PL. It is a straightforward extension of `eap_theta.ox`.

The headers are the same.

The global variables are mostly the same except we added a few variable related to the item parameters.

The variables `alpha0` and `beta0` will hold the current values of the item parameters; the variable `draws` will be $T \times 2J$ matrix holding *all* the $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ draws after the burn-in period.

```
decl N=1000,J=10; //N=sample size, J=test length
decl T=2000; //size of sample from the posterior
decl m=1000; //burn-in
decl theta,alpha,beta,X; //ability, item parameters, data
decl theta0,alpha0,beta0; //current value of theta, alpha, beta
decl draws,s_theta,s_theta2;//itempar draws,sufficient statistics
```

The function TwoPL is the same.

```
draw_ab(){
decl alpha1=alpha0+.05*rann(J,1);
decl  beta1= beta0+.10*rann(J,1);
decl tmp=TwoPL(theta0,alpha0,beta0);
decl ll0=sumc(X.*log(tmp)+(1-X).*log(1-tmp))'+
 log(densbeta(alpha0/20,1.3,22))+
 log(densbeta((beta0+5)/10,12.66,12.66));
tmp=TwoPL(theta0,alpha1,beta1);
decl ll1=sumc(X.*log(tmp)+(1-X).*log(1-tmp))'+
 log(densbeta(alpha1/20,1.3,22))+
 log(densbeta((beta1+5)/10,12.66,12.66));
decl acc=exp(ll1-ll0);//acceptance probability
decl ind=vecindex(acc.>ranu(J,1));
alpha0[ind]=alpha1[ind];
 beta0[ind]= beta1[ind];
}//end draw_ab
```

The function `draw_ab` draws the item parameters. Again, we will use a symmetric candidate-generating functions (i.e., normal) to make it easier for us to compute the acceptance probabilities.

Note that σ_α and σ_β need not be the same.

In fact, in some cases where convergence rates vary extremely across items, we may actually need to use a different candidate variance for each parameter, as in, we may need σ_{α_j} and σ_{β_j}.

For the priors of α_j and β_j, we use four-parameter beta distributions instead of the lognormal and normal distributions.

Note that `acc` is $J \times 1$ so α_j^* and β_j^* will be accepted or rejected at the same time.

The function `draw_theta` is almost the same to `MH_draw` in `eap_theta.ox`. But instead of `alpha` and `beta`, the function involves `alpha0` and `beta0` to reflect the fact that the item parameters may be different from one iteration to another.

In addition to the original variable, the function `initiate` also provides starting values for `alpha0` and `beta0`, and sets `draws` as a null matrix.

In the `update` function will also have `draws=draws|(alpha0'~beta0');` to keep track of all the item parameter draws after the burn-in.

```
evaluate(){
decl est=meanc(draws);
est=shape(est,J,2);
decl se=sqrt(varc(draws));
se=shape(se,J,2);
print("\n     True Alpha   True Beta");
print(alpha~beta);
print("\n     Est. Alpha   Est. Beta    SE(Alpha)     SE(Beta)");
print(est~se);
est=s_theta/T;
decl tmp=correlation(theta~est);
println("\n      Ability Estimates  ");
print("Corr(True,Est):    ",tmp[0][1]);
tmp=(s_theta2-T*est.^2)/(T-1);
print("\nMean Posterior SD: ",sqrt(meanc(tmp))[0]);
tmp=sqrt(meanc((theta-est).^2));
print("\nRMSE:              ",tmp[0],"\n");
}//end evaluate
```

Because this program needs to performs more analyses, we need to write a separate function, `evaluate`, to examine the quality of the item parameter and ability estimates.

In addition to the measures of the quality of the ability estimates, this function also gives the item parameter estimates (i.e., posterior means) and the standard errors (i.e., posterior standard deviations).

Note that the original dimension of the posterior means and standard deviations is $1 \times 2J$. We need to use the function `shape` to change them to $J \times 2$.

As before, the `main` function generates the data, calls `initiate`, uses a loop to call `draw_ab` and `draw_theta` $T + m$ times, and updates the variables of interest within the loop after m draws.

After the draws, the `main` function also calls `evaluate`, and then prints the time it takes to carry out the MCMC algorithm.

1. Modify T and m to see how they affect the results.

2. How do N and J affect the time it takes to implement the algorithm?

An MCMC Algorithm for the HO-DINA Model

Let N, J, and K be the number of examinees, test length, and number of attributes, respectively.

We will assume a 1PL the $P(\alpha_k = 1|\theta_i)$, and the DINA model for $P_{ij} = P(X_j = 1|\boldsymbol{\alpha}_i)$.
In addition, we also assume $P(\boldsymbol{\alpha}_i|\boldsymbol{\lambda}, \theta_i) = \prod_{k=1}^{K} P(\alpha_{ik}|\boldsymbol{\lambda}, \theta_i)$.

The goal this time is to make inferences about $p(\boldsymbol{\lambda}, \boldsymbol{g}, \boldsymbol{s}, \boldsymbol{\theta}, \boldsymbol{\alpha}|\mathbf{X})$, where $\boldsymbol{\lambda} = \{\lambda_1, \boldsymbol{\lambda}_0\} = \{\lambda_1, \lambda_{01}, \ldots, \lambda_{0K}\}$, $\boldsymbol{g} = \{g_j\}$, $\boldsymbol{s} = \{s_j\}$, $\boldsymbol{\theta} = \{\theta_i\}$, and $\boldsymbol{\alpha} = \{\boldsymbol{\alpha}_i\}$.

Prior, Posterior, and Full Conditional Distributions

Let the priors be $\boldsymbol{\lambda}$, $\boldsymbol{\theta}$, $\boldsymbol{\alpha}$, $\boldsymbol{g}$ and $\boldsymbol{s}$:

$$\begin{aligned}
p(\lambda_1) &\sim 4 - \text{Beta}(0, 16, 1.5, 21.5) \\
p(\lambda_{0k}) &\sim N(-1, 1) \\
p(\theta_i) &\sim 4 - \text{Beta}(-4, 4, 8.21, 8.21) \\
p(\alpha_{ik} = 1|\boldsymbol{\lambda}, \theta_i) &\sim \text{Ber}\left(\{1 + \exp\left[-1.7\lambda_1(\theta_i - \lambda_{0k})\right]\}^{-1}\right) \\
p(g_j) &\sim 4 - \text{Beta}(0, .9, 1.5, 2) \\
p(1 - s_j) &\sim 4 - \text{Beta}(.1, 1, 2, 1.5)
\end{aligned}$$

The likelihood of the data given, $\boldsymbol{g}$, $\boldsymbol{s}$, and $\boldsymbol{\alpha}$ is

$$\begin{aligned}
L(\mathbf{X}|\boldsymbol{g}, \boldsymbol{s}, \boldsymbol{\alpha}) &= \prod_{i=1}^{N} L(\boldsymbol{X}_i|\boldsymbol{g}, \boldsymbol{s}, \boldsymbol{\alpha}_i) \\
&= \prod_{j=1}^{J} L(\boldsymbol{X}_j|g_j, s_j, \boldsymbol{\alpha}) \\
&= \prod_{i=1}^{N}\prod_{j=1}^{J} \left[s_j^{1-X_{ij}}(1 - s_j)^{X_{ij}}\right]^{\eta_{ij}} \left[g_j^{X_{ij}}(1 - g_j)^{1-X_{ij}}\right]^{1-\eta_{ij}},
\end{aligned}$$

where $\eta_{ij} = I(\boldsymbol{\alpha}_i'\boldsymbol{q}_j = \boldsymbol{q}_j'\boldsymbol{q}_j)$.

The joint posterior distribution is

$$p(\boldsymbol{\lambda}, \boldsymbol{g}, \boldsymbol{s}, \boldsymbol{\theta}, \boldsymbol{\alpha}|\mathbf{X}) \propto L(\mathbf{X}|\boldsymbol{g}, \boldsymbol{s}, \boldsymbol{\alpha}) \times p(\boldsymbol{\alpha}|\boldsymbol{\lambda}, \boldsymbol{\theta}) \times p(\boldsymbol{\theta}) \times p(\boldsymbol{\lambda}) \times p(\boldsymbol{g}) \times p(\boldsymbol{s}).$$

To implement Gibss sampling, we need the following full conditional distributions:

$$\begin{aligned}
p(\lambda_1|\mathbf{X},\boldsymbol{\lambda}_0,\boldsymbol{\theta},\boldsymbol{\alpha},\boldsymbol{g},\boldsymbol{s}) &\propto p(\boldsymbol{\alpha}|\boldsymbol{\lambda},\boldsymbol{\theta})\times p(\lambda_1)\\
p(\lambda_{0k}|\mathbf{X},\lambda_1,\boldsymbol{\lambda}_0^{(-k)},\boldsymbol{\theta},\boldsymbol{\alpha},\boldsymbol{g},\boldsymbol{s}) &\propto p(\boldsymbol{\alpha}_k|\lambda_1,\lambda_{0k},\boldsymbol{\theta})\times p(\lambda_{0k})\\
p(\theta_i|\mathbf{X},\boldsymbol{\lambda},\boldsymbol{\theta}^{(-i)},\boldsymbol{\alpha},\boldsymbol{g},\boldsymbol{s}) &\propto p(\boldsymbol{\alpha}_i|\boldsymbol{\lambda},\boldsymbol{\theta}_i)\times p(\theta_i)\\
p(\boldsymbol{\alpha}_i|\mathbf{X},\boldsymbol{\lambda},\boldsymbol{\theta},\boldsymbol{\alpha}^{(-i)},\boldsymbol{g},\boldsymbol{s}) &\propto L(\boldsymbol{X}_i|\boldsymbol{g},\boldsymbol{s},\boldsymbol{\alpha}_i)\times p(\boldsymbol{\alpha}_i|\boldsymbol{\lambda},\boldsymbol{\theta}_i)\\
p(g_j|\mathbf{X},\boldsymbol{\lambda},\boldsymbol{\theta},\boldsymbol{\alpha},\boldsymbol{g}^{(-j)},\boldsymbol{s}) &\propto L(\boldsymbol{X}_j|g_j,s_j,\boldsymbol{\alpha})\times p(g_j)\\
p(s_j|\mathbf{X},\boldsymbol{\lambda},\boldsymbol{\theta},\boldsymbol{\alpha},\boldsymbol{g},\boldsymbol{s}^{(-j)}) &\propto L(\boldsymbol{X}_j|g_j,s_j,\boldsymbol{\alpha})\times p(s_j)
\end{aligned}$$

Verify that the above full conditional distributions are correct.

Based on these formulations, we will update all the parameters *one at a time*, including the item parameters.

None of the full conditional distribution is known so we have to use the M-H algorithm.

At iteration $t+1$:

1. Draw λ_1^* from $N(\lambda_1^{(t)},\sigma^2_{\lambda_1})$, and accept λ_1^* with probability

$$\mathcal{A}(\lambda_1^{(t)},\lambda_1^*)=\min\left\{\frac{p(\boldsymbol{\alpha}^{(t)}|\lambda_1^*,\boldsymbol{\lambda}_0^{(t)},\boldsymbol{\theta}^{(t)})\times p(\lambda_1^*)}{p(\boldsymbol{\alpha}^{(t)}|\lambda_1^{(t)},\boldsymbol{\lambda}_0^{(t)},\boldsymbol{\theta}^{(t)})\times p(\lambda_1^{(t)})},1\right\}.$$

2. For $k=1,\ldots,K$, draw λ_{0k}^* from $N(\lambda_{0k}^{(t)},\sigma^2_{\lambda_0})$, and accept λ_{0k}^* with probability

$$\mathcal{A}(\lambda_{0k}^{(t)},\lambda_{0k}^*)=\min\left\{\frac{p(\boldsymbol{\alpha}^{(t)}|\lambda_1^{(t+1)},\lambda_{0k}^*,\boldsymbol{\lambda}_0^{(-j)(t)},\boldsymbol{\theta}^{(t)})\times p(\lambda_{0k}^*)}{p(\boldsymbol{\alpha}^{(t)}|\lambda_1^{(t+1)},\lambda_{0k}^{(t)},\boldsymbol{\lambda}_0^{(-j)(t)},\boldsymbol{\theta}^{(t)})\times p(\lambda_{0k}^{(t)})},1\right\}.$$

3. For $i=1,\ldots,N$, draw θ_i^* from $N(\theta_i^{(t)},\sigma^2_{\theta})$, and accept λ_{0k}^* with probability

$$\mathcal{A}(\theta_i^{(t)},\theta_i^*)=\min\left\{\frac{p(\boldsymbol{\alpha}_i^{(t)}|\boldsymbol{\lambda}^{(t+1)},\boldsymbol{\theta}^*)\times p(\theta_i^*)}{p(\boldsymbol{\alpha}_i^{(t)}|\boldsymbol{\lambda}^{(t+1)},\boldsymbol{\theta}^{(t)})\times p(\theta_i^{(t)})},1\right\}.$$

4. For $i=1,\ldots,N$ and $k=1,\ldots,K$, draw α_{ik}^* from Ber(.5), and accept $\boldsymbol{\alpha}_i^*$ with probability

$$\mathcal{A}(\boldsymbol{\alpha}_i^{(t)},\boldsymbol{\alpha}_i^*)=\min\left\{\frac{L(\boldsymbol{X}_i|\boldsymbol{g}^{(t)},\boldsymbol{s}^{(t)},\boldsymbol{\alpha}^*)\times p(\boldsymbol{\alpha}_i^*|\boldsymbol{\lambda}^{(t+1)},\boldsymbol{\theta}^{(t+1)})}{L(\boldsymbol{X}_i|\boldsymbol{g}^{(t)},\boldsymbol{s}^{(t)},\boldsymbol{\alpha}^{(t)})\times p(\boldsymbol{\alpha}_i^{(t)}|\boldsymbol{\lambda}^{(t+1)},\boldsymbol{\theta}^{(t+1)})},1\right\}.$$

5. For $j=1,\ldots,J$, draw g_j^* from $N(g_j^{(t)},\sigma^2_{g})$, and accept g_j^* with probability

$$\mathcal{A}(g_j^{(t)},g_j^*)=\min\left\{\frac{L(\boldsymbol{X}_j|g_j^*,s_j^{(t)},\boldsymbol{\alpha}^{(t+1)})\times p(g_i^*)}{L(\boldsymbol{X}_j|g_j^{(t)},s_j^{(t)},\boldsymbol{\alpha}^{(t+1)})\times p(g_i^{(t)})},1\right\}.$$

6. For $j=1,\ldots,J$, draw s_j^* from $N(s_j^{(t)},\sigma^2_{s})$and accept s_j^* with probability

$$\mathcal{A}(s_j^{(t)},s_j^*)=\min\left\{\frac{L(\boldsymbol{X}_j|g_j^{(t+1)},s_j^*,\boldsymbol{\alpha}^{(t+1)})\times p(s_i^*)}{L(\boldsymbol{X}_j|g_j^{(t+1)},s_j^{(t)},\boldsymbol{\alpha}^{(t+1)})\times p(s_i^{(t+1)})},1\right\}.$$

A Detour: The Gelman-Rubin Convergence Statistic R

In addition to graphical method, which can be unreliable, we can also compute the Gelman-Rubin statistic to diagnose the convergence of the chain.

The statistic, which involves multiple parallel chains with dispersed starting values to estimate the portion of the posterior mean estimator that is due to the Monte Carlo error, is used to assess the convergence of scalar parameters.

This means that even if $\boldsymbol{\theta}$ is multivariate, we assess the convergence of the chain element by element.

Let θ be the scalar parameter, and let $\theta_c^{(t)}$ denote the t^{th} draw of the c^{th} chain, where $c = 1, \ldots, C$.

We define the between- and within-chain variances as

$$\begin{aligned} B &= \frac{T}{C-1}\sum_{c=1}^{C}(\bar{\theta}_c - \bar{\theta})^2 \\ W &= \frac{1}{C}\sum_{c=1}^{C} S_c^2, \end{aligned}$$

where

$$\begin{aligned} \bar{\theta}_c &= \frac{1}{T}\sum_{t=1}^{T}\theta_c^{(t)} \\ \bar{\theta} &= \frac{1}{C}\sum_{c=1}^{C}\bar{\theta}_c \\ S_c^2 &= \frac{1}{T-1}\sum_{t=1}^{T}(\theta_c^{(t)} - \bar{\theta}_c)^2 \end{aligned}$$

Note that multiple chains are need to be able to compute B.

Note also that $\sqrt{\hat{R}}$ is computed using means and variances of the draws across the different chains.

This means that we can monitor convergence by simply computing the sufficient statistics $s_c = \sum_{t=1}^{T}\theta_c^{(t)}$ and $ss_c = \sum_{t=1}^{T}\theta_c^{(t)2}$, without having to save to the draws.

The marginal posterior variance of the parameter, $V(\theta|\mathbf{X})$, can be estimated by computing the weighted average of W and B as follows:

$$\hat{V}^*(\theta|\mathbf{X}) = \frac{T-1}{T}W + \frac{1}{T}B.$$

If the chain has not reached the target distribution, $\bar{\theta}_c$ from the different chains will be different from each other resulting in $\hat{V}^*(\theta|\mathbf{X})$ overestimating $V(\theta|\mathbf{X})$.

On the other hand, if the chain has reached its target distribution, $\hat{V}^*(\theta|\mathbf{X}) \approx W$ and $B \approx 0$.

This means that $\hat{V}^*(\theta|\mathbf{X}) \to V(\theta|\mathbf{X})$ as $n \to \infty$.

The *potential scale reduction factor* is estimated by

$$\sqrt{\hat{R}} = \sqrt{\frac{\hat{V}^*(\theta|\mathbf{X})}{W}},$$

which approaches 1 when $T \to \infty$.

As a rule of thumb, $\sqrt{\hat{R}} < 1.2$ is considered acceptable. However, in some applications, something smaller (e.g., 1.1), might be preferable.

If the chains have converged, all the draws after the burn-in are from the same target distribution.

Therefore, we can combine all the draws, and base our inferences on the pooled draws, instead of the draws from a single chain.

With C chains each with T draws after burn-in, we will have $T \times C$ instead of T draws to work with. With more draws, the Monte Carlo error can be made smaller.

For the us to be able to conclude that convergence is reached for entire distribution, potential scale reduction factor must be small for all the parameters of interest.

In latent variable modeling, we typically limit this to the structural parameters.

For example, in the HO-DINA model, we monitor the convergence of $\boldsymbol{\lambda}$, $\boldsymbol{g}$, and $\boldsymbol{s}$.

For us to conclude that convergence has been reached for entire distribution, we only need to make sure that $\sqrt{\hat{R}}_{\max}$ is sufficiently small.

It should be noted that, for some researchers, convergence of single parameters is not sufficient to establish *joint* convergence of the parameters.

To monitor joint convergence, the Brooks-Gelman *multivariate potential scale reduction factor* may be used.

However, in some applications, the number of parameters involved in latent variable models can be extremely large. This can make computing of the multivariate potential scale reduction factor difficult.

An alternative is to monitor the convergence of the parameters by block. For example, the multivariate potential scale reduction factor can be computed for $\boldsymbol{\lambda}$, and $\{g_j, s_j\}$ separately.

End of Detour

Implementation in Ox of the MCMC Algorithm for the HO-DINA Model

The program `mcmc_hodina.ox` implements the MCMC algorithm for the HO-DINA model outlined above. In addition to parameter estimation, the program also runs multiple (i.e., `Nchain`) chains to assess convergence. The program stops when `max_sqrtR` (i.e., $\sqrt{\hat{R}}_{\max}$) is less than `crit_sqrtR`, or the number of draws (per chain) is equal to `max_it`.

The minimum number of draws is set to `min_it`, of which `m` draws are discarded as burn-in.

After the initial `min_it` draws, convergence is assessed. If convergence is achieved, sampling is terminated; if not, additional `inc` draws are obtained before convergence is assessed again.

We declare the sufficient statistics needed for monitoring convergence. The current number of variables is for `Nchain=4`. We also declare `ndraws` to keep tract of the total number of iterations after burn-in - the program can have different stopping points. This number can reflect the thinned value.

We only declare two types of sufficient statistics: one that is chain-specific, and another that cuts across chain.
For the former type, we have the sufficient statistics for the structural parameters $\boldsymbol{\lambda}$, $\boldsymbol{g}$, and $\boldsymbol{s}$, which are also used to monitor convergence.
For the latter type, we have the sufficient statistics for the incidental variables $\boldsymbol{\theta}$ and $\boldsymbol{\alpha}$.

```
decl N=536,J=15,K=5;//sample size, test length, # of attributes
decl files={"qmatrix.txt","X536.dat"};//data and Q-matrix filenames
//min and max chain lengths, burn-in, increment; # of it->not fixed
decl min_it=2000,max_it=50000,m=1000,inc=1000;
decl crit_sqrtR=1.2,tot_it;//conv. criterion, tot_it per chain
decl Q,X;//Q-matrix and data variables
decl theta0,lambda0,gs0,alpha0;//variables for current values
//Candidate variances
decl cand_theta=.25,cgs0=.02,cgs1=.025,clambda0=.1,clambda1=.05;
decl Nchain=4,max_sqrtR;//number of chains, max of the sqrtR
//Sufficient stats for monitoring convergence
decl c1s,c1s2,c2s,c2s2,c3s,c3s2,c4s,c4s2,ndraws;
//Sufficient stats for incidental parameters
decl stheta,sstheta,salpha,ssalpha;
```

The function `OnePL` computes the $N \times K$ probabilities based on the arguments `t` and `l`.

```
OnePL(decl t,l){
return ones(N,K)./(1+exp(-1.7*l[0][0]*(t*ones(1,K)-ones(N,1)*l[][1]')));
}end OnePL
```

The function `draw_theta` updates the higher-order latent trait $\boldsymbol{\theta}$. Note that for this program, by using a 4-Beta$(-4, 4, 8.21, 8.21)$, we are effectively constraining θ_i to be in the interval $[-4, 4]$.

Again, $\boldsymbol{\theta}$ is sampled en bloc, but updated individually.

Note that Ox does not have a function for the four-parameter beta distribution so Y needs to be transformed $(Y - a)/(b - a)$ so it can be evaluated using the usual beta distribution function.

```
draw_theta(){
decl theta1=cand_theta*rann(N,1)+theta0;
decl p=OnePL(theta0,lambda0);
decl d0=sumr(alpha0.*log(p)+(1-alpha0).*log(1-p))+
        log(densbeta((theta0+4)/8,8.21,8.21));
p=OnePL(theta1,lambda0);
decl d1=sumr(alpha0.*log(p)+(1-alpha0).*log(1-p))+
        log(densbeta((theta1+4)/8,8.21,8.21));
decl acc=d1-d0;
decl accind=vecindex(ranu(N,1).<exp(acc));
theta0[accind][]=theta1[accind][];
}//end draw_theta
```

The function `draw_lambda` updates the higher-order structural parameter $\boldsymbol{\lambda}$. It consists of two steps - one for updating $\boldsymbol{\lambda}_0$, and one for updating λ_1.

For $\boldsymbol{\lambda}_0$, we need to sum across the rows; for λ_1, we need to sum across the rows and the columns.

```
draw_lambda(){
//Step for the difficulty parameters
decl lambda1=lambda0[][0]~(clambda0*rann(K,1)+lambda0[][1]);
decl d0=sumc(alpha0.*log(OnePL(theta0,lambda0))+
   (1-alpha0).*log(1-OnePL(theta0,lambda0)))+log(densn(lambda0[][1]'+1));
decl d1=sumc(alpha0.*log(OnePL(theta0,lambda1))+
   (1-alpha0).*log(1-OnePL(theta0,lambda1)))+log(densn(lambda1[][1]'+1));
decl acc=d1-d0;
decl accind=vecindex(ranu(1,K).<exp(acc));
lambda0[accind][]=lambda1[accind][];
//Step for the common discrimination parameter
lambda1=(clambda1*rann(1,1)+lambda0[][0])~lambda0[][1];
d0=sumr(sumc(alpha0.*log(OnePL(theta0,lambda0))+
   (1-alpha0).*log(1-OnePL(theta0,lambda0))))
   +log(densbeta(lambda0[0][0]/16,1.5,21.5));
d1=sumr(sumc(alpha0.*log(OnePL(theta0,lambda1))+
   (1-alpha0).*log(1-OnePL(theta0,lambda1))))
   +log(densbeta(lambda1[0][0]/16,1.5,21.5));
acc=d1-d0;
if(ranu(1,1)<exp(acc)) lambda0[][0]=lambda1[][0];
}//end draw_lambda
```

Note that because we are using a common discrimination parameter, we need to make sure that, at each iteration, only one value is sampled, and its prior density is counted only once.

The function `draw_alpha` updates $\boldsymbol{\alpha}$. The candidate values are sampled as α_{ik}, but accepted or rejected as $\boldsymbol{\alpha}_i$.

```
draw_alpha(){
decl alpha1=ranbinomial(N,K,1,.5);
decl tmp=alpha0*Q';
tmp=tmp.==ones(N,1)*sumc(Q');
tmp=tmp.*(ones(N,1)*gs0[][1]')+(1-tmp).*(ones(N,1)*gs0[][0]');
decl d0=sumr(X.*log(tmp)+(1-X).*log(1-tmp))
   +sumr(alpha0.*log(OnePL(theta0,lambda0))+
   (1-alpha0).*log(1-OnePL(theta0,lambda0)));
tmp=alpha1*Q';
tmp=tmp.==ones(N,1)*sumc(Q');
tmp=tmp.*(ones(N,1)*gs0[][1]')+(1-tmp).*(ones(N,1)*gs0[][0]');
decl d1=sumr(X.*log(tmp)+(1-X).*log(1-tmp))
   +sumr(alpha1.*log(OnePL(theta0,lambda0))+
   (1-alpha1).*log(1-OnePL(theta0,lambda0)));
decl acc=d1-d0;
decl accind=vecindex(ranu(N,1).<exp(acc));
alpha0[accind][]=alpha1[accind][];
}//end draw_alpha
```

```
draw_gs(){
decl tmp0=alpha0*Q';
tmp0=tmp0.==ones(N,1)*sumc(Q');
decl gs1=(cgs0*rann(J,1)+gs0[][0])~gs0[][1];
decl tmp=tmp0.*(ones(N,1)*gs0[][1]')+(1-tmp0).*(ones(N,1)*gs0[][0]');
decl d0=sumc(X.*log(tmp)+(1-X).*log(1-tmp))'+log(densbeta(gs0[][0]/.9,1.5,2));
tmp=tmp0.*(ones(N,1)*gs1[][1]')+(1-tmp0).*(ones(N,1)*gs1[][0]');
decl d1=sumc(X.*log(tmp)+(1-X).*log(1-tmp))'+log(densbeta(gs1[][0]/.9,1.5,2));
decl acc=d1-d0;
decl accind=vecindex(ranu(J,1).<exp(acc));
gs0[accind][]=gs1[accind][];
gs1=gs0[][0]~(cgs1*rann(J,1)+gs0[][1]);
tmp=tmp0.*(ones(N,1)*gs0[][1]')+(1-tmp0).*(ones(N,1)*gs0[][0]');
d0=sumc(X.*log(tmp)+(1-X).*log(1-tmp))'+log(densbeta((gs0[][1]-.1)/.9,2,1.5));
tmp=tmp0.*(ones(N,1)*gs1[][1]')+(1-tmp0).*(ones(N,1)*gs1[][0]');
d1=sumc(X.*log(tmp)+(1-X).*log(1-tmp))'+log(densbeta((gs1[][1]-.1)/.9,2,1.5));
acc=d1-d0;
accind=vecindex(ranu(J,1).<exp(acc));
gs0[accind][]=gs1[accind][];
}//end draw_gs
```

The function `draw_gs` updates the item parameters. Note that the algorithm estimates $1-s_j$, rather s_j.

The function `initiate` is for initializing the current states and sufficient statistics. This function is called C times, once for each chain.

```
initiate(decl nc){
if(nc==0) {c1s=c1s2=c2s=c2s2=c3s=c3s2=c4s=c4s2=ndraws=0;
           max_sqrtR=2;}
theta0=rann(N,1)*.05+standardize(meanr(X));
decl tmp=(-3*ranu(K,1)+1);
lambda0=(1+2*ranu(1,1)*ones(K,1))~tmp;
alpha0=ranbinomial(N,K,1,.5);
gs0=ranu(J,1)*.5~.5+ranu(J,1)*.5;
stheta=sstheta=0;
salpha=ssalpha=0;
}//end initiate
```

The function `draw_dina` is called to carry out the different steps of the M-H algorithm.

The function `save` updates the sufficient statistics needed to estimate $\sqrt{R}$, $\boldsymbol{\alpha}$, and $\boldsymbol{\theta}$.

```
save(decl nc){
if(nc==0){
c1s=c1s+(lambda0[0][0]~lambda0[][1]'~vec(gs0)');
c1s2=c1s2+(lambda0[0][0]~lambda0[][1]'~vec(gs0)').^2;
ndraws=ndraws+1;
}//end nc==0
if(nc==1){
c2s=c2s+(lambda0[0][0]~lambda0[][1]'~vec(gs0)');
c2s2=c2s2+(lambda0[0][0]~lambda0[][1]'~vec(gs0)').^2;
}//end nc==1
if(nc==2){
c3s=c3s+(lambda0[0][0]~lambda0[][1]'~vec(gs0)');
c3s2=c3s2+(lambda0[0][0]~lambda0[][1]'~vec(gs0)').^2;
}//end nc==2
if(nc==3){
c4s=c4s+(lambda0[0][0]~lambda0[][1]'~vec(gs0)');
c4s2=c4s2+(lambda0[0][0]~lambda0[][1]'~vec(gs0)').^2;
}//end nc==3
stheta=stheta+theta0; sstheta=sstheta+theta0.^2;
salpha=salpha+alpha0; ssalpha=ssalpha+alpha0.^2;
}//end save
```

When the first chain is involved, `ndraws` is also incremented.

The function `compute_sqrtR` computes $\sqrt{\hat{R}_{\max}}$. It first computes the $\sqrt{\hat{R}}$ for each of the structural parameters, and picks out the maximum of these convergence statistics.

```
compute_sqrtR(){
decl B,W,V;
decl m=(c1s|c2s|c3s|c4s)/ndraws;
B=sumc(m.^2)-sumc(m).^2/Nchain;
B=ndraws/(Nchain-1)*B;
W=c1s2-c1s.^2/ndraws;
W=W+c2s2-c2s.^2/ndraws;
W=W+c3s2-c3s.^2/ndraws;
W=W+c4s2-c4s.^2/ndraws;
W=W/(Nchain*(ndraws-1));
V=(1-1/ndraws)*W+1/ndraws*B;
max_sqrtR=maxc(sqrt(V./W)');
}//end compute_sqrtR
```

```
print_est(const et){
format(1000);
decl tmp=sqrt((sstheta-(stheta.^2)/(Nchain*ndraws))/(Nchain*ndraws-1));
decl par=sprint("theta.out");
decl file=fopen(par,"w");
fprint(file,"%#M",stheta/(Nchain*ndraws)~tmp);
fclose(file);
par=sprint("alpha.out");
file=fopen(par,"w");
fprint(file,"%#M",salpha/(Nchain*ndraws).>.5);
fclose(file);
decl m=c1s+c2s+c3s+c4s;
decl se=c1s2+c2s2+c3s2+c4s2;
se=sqrt((se-m.^2/(Nchain*ndraws))/(Nchain*ndraws-1));;
m=m/(Nchain*ndraws);
print("\n","******** Higher-Order Structural Parameters ********","\n");
print("Common Discrimination", "\n", "      Est.      SE","%10.4f", m[0]~se[0]);
print("Intercepts", "\n", "      Att.      Est.      SE","%10.4f",
      range(1,K)'~m[1:K]'~se[1:K]');
print("\n","***********       Item Parameters       ***********","\n");
print("     Item        g        1-s      SE(g)    SE(1-s)","%10.4f",
      range(1,J)'~shape(m[][K+1:2*J+K],J,2)~shape(se[][K+1:2*J+K],J,2));
print("\n","Elapsed Time: ",timespan(et),"\n");
}//print_est
```

The function `print_est` computes the estimates of $\boldsymbol{\theta}$, $\boldsymbol{\alpha}$, $\boldsymbol{\lambda}$, $\boldsymbol{g}$, and $1 - \boldsymbol{s}$.

The estimates of the incidental parameters are printed to file, whereas the estimates of the structural parameters are printed as screen output.

The main function has a few components.

The first component records the starting time, and counts the total number of structural and incidental parameters.

```
ranseed(32671);
print("Starting Time: ",time(),"\n");
decl et=timer();
decl npar=(N+1)*(K+1)+2*J;//total number of parameters
```

The second component reads the Q-matrix and data files.

The next component initiates the variables and carries out the initial number of draws for each of the C chains.

After going through all the chains, $\sqrt{\hat{R}}$ is computed, and total number of iteration is updated.

```
for(decl nc=0;nc<Nchain;nc++){
initiate(nc);
for(decl j=0;j<min_it;j++){
draw_dina();
if(j>=m && imod(j,10)==0) save(nc);//saving and thinning suff. stats
}//end sampling
decl par=sprint("tmp_",nc,".out");//saving most recent values
decl file=fopen(par,"w");
fprint(file,"%#M",lambda0[0][0]~lambda0[][1]'~vec(gs0)'~theta0'~vec(alpha0)');
fclose(file);
}//end nc

compute_sqrtR();
tot_it=min_it;
print("\n","     Iteration    Max SqrtR");
print(tot_it~max_sqrtR);
```

When $\sqrt{\hat{R}}$ is greater than `crit_sqrtR` after the initial number of draws, additional draws are sampled.

After `inc` additional draws, `crit_sqrtR` is computed again to examine convergence.

The sampling process continues until $\sqrt{\hat{R}}$ is less than `crit_sqrtR`, or the maximum number of iterations is reached.

Note that we are thinning the chains by saving only every tenth draw. The function `imod` helps us to this.

```
while (max_sqrtR>crit_sqrtR && tot_it<=max_it){
for(decl nc=0;nc<Nchain;nc++){
decl tmp1;
tmp0=sprint("tmp_",nc,".out");
file0=fopen(tmp0);
fscan(file0,"%#M",1,npar,&tmp1);
fclose(file0);

lambda0[][0]=tmp1[0];
lambda0[][1]=tmp1[1:K]';
gs0=shape(tmp1[K+1:2*J+K],J,2);
theta0=tmp1[2*J+K+1:2*J+K+N]';
alpha0=shape(tmp1[2*J+K+N+1:npar-1],N,K);

for(decl j=0;j<inc;j++){
draw_dina();
if(imod(j,10)==0) save(nc);
}//end j

decl par=sprint("tmp_",nc,".out");
decl file=fopen(par,"w");
fprint(file,"%#M",lambda0[0][0]~lambda0[][1]'~vec(gs0)'~theta0'~vec(alpha0)');
fclose(file);
}//end additional draws

compute_sqrtR();
tot_it=tot_it+inc;
print(tot_it~max_sqrtR);
}//end while
```

Note that the chains are restarted right where they left off.

By saving the most recent values after each sampling chunk, we do not need to restart the chains if we they have not converged about a particular number of iterations.

Finally, when the chains have converged, or maximum number of iterations is reach, the function `print_est` is called.

Modify `min_it`, `max_it`, `m`, `inc`, and `crit_sqrtR` to see how they affect the time needed to get the estimates.

How do these changes affect the estimates and standard errors?

Additional Exercises

1. Compare the MML and MCMC estimates of the 2PL in terms of computational speed and quality of the estimates.

2. Updating α_j and β_j Separately

Modify `mcmc_2PL.ox` so that α_j and β_j are updated separately. This will require one step for sampling and updating α_j, and another step for sampling and updating β_j. However, the two steps can be carried out in the same function.

How does updating α_j and β_j separately affect the computational efficiency of the algorithm and quality of the parameter estimates?

3. Compare the MML and MCMC estimates of the HO-DINA model.

4. Modify `mcmc_hodina.ox` to use five parallel chains to evaluate convergence.

Markov Chain Monte Carlo - Part III

An MCMC Algorithm for Multi-Unidimensional Ability Estimation

Most assessments involve batteries of tests that measure highly correlated abilities. However, it is also not uncommon in practice for these tests to be scored *one test at a time*.

In so doing, only actual responses from a specific test are taken into consideration in determining examinees' scores.

In doing so, the correlation structure of the abilities are ignored.

Because available information relevant to examinees' standings on the tests is not incorporated, this method of ability estimation is deemed suboptimal.

To improve efficiency, we can estimate the abilities *simultaneously*.

For this, we would need a multidimensional model.

The extension of the 3PL model to the multidimensional context as follows:

$$P(X_{ij} = 1|\boldsymbol{\theta}_i, \boldsymbol{\alpha}_j, \beta_j, \gamma_j) = \gamma_j + (1 - \gamma_j)\frac{\exp(\boldsymbol{\alpha}_j'\boldsymbol{\theta}_i + \beta_j)}{1 + \exp(\boldsymbol{\alpha}_j'\boldsymbol{\theta}_i + \beta_j)}$$

where

- $P(X_{ij} = 1|\boldsymbol{\theta}_i, \boldsymbol{\alpha}_j, \beta_j, \gamma_j)$ is the probability of examinee i responding to item j correctly;
- X_{ij} is the response of examinee i to item j ($0 =$ incorrect, $1 =$ correct);
- $\boldsymbol{\theta}_i$ is the ability vector of the examinee;
- $\boldsymbol{\alpha}_j$ is the vector of item parameters related to the discrimination of the item;
- β_j is the parameter related to the difficulty of the item;
- γ_j is the pseudo-guessing parameter of the item;
- $i = 1, \ldots, N$ (the total number of examinees); and
- $j = 1, \ldots, J$ (the total number of items).

For the purposes of this example, we will assume a simple structure is (i.e., each item measures one dimension of ability, and thus, $\boldsymbol{\alpha}_j$ contains only one non-zero element).

Individually, the tests are unidimensional; *collectively*, they are multidimensional. Thus, what we have is a *multi-unidimensional* tests.

Under this assumption, the multidimensional 3PL model can be reexpressed as:

$$P_{ij(d)} = P(X_{j(d)} = 1|\theta_{i(d)}, \alpha_{j(d)}, \beta_{j(d)}, \gamma_{j(d)}) = \gamma_{j(d)} + (1 - \gamma_{j(d)})\frac{\exp(\alpha_{j(d)}\theta_{i(d)} + \beta_{j(d)})}{1 + \exp(\alpha_{j(d)}\theta_{i(d)} + \beta_{j(d)})}$$

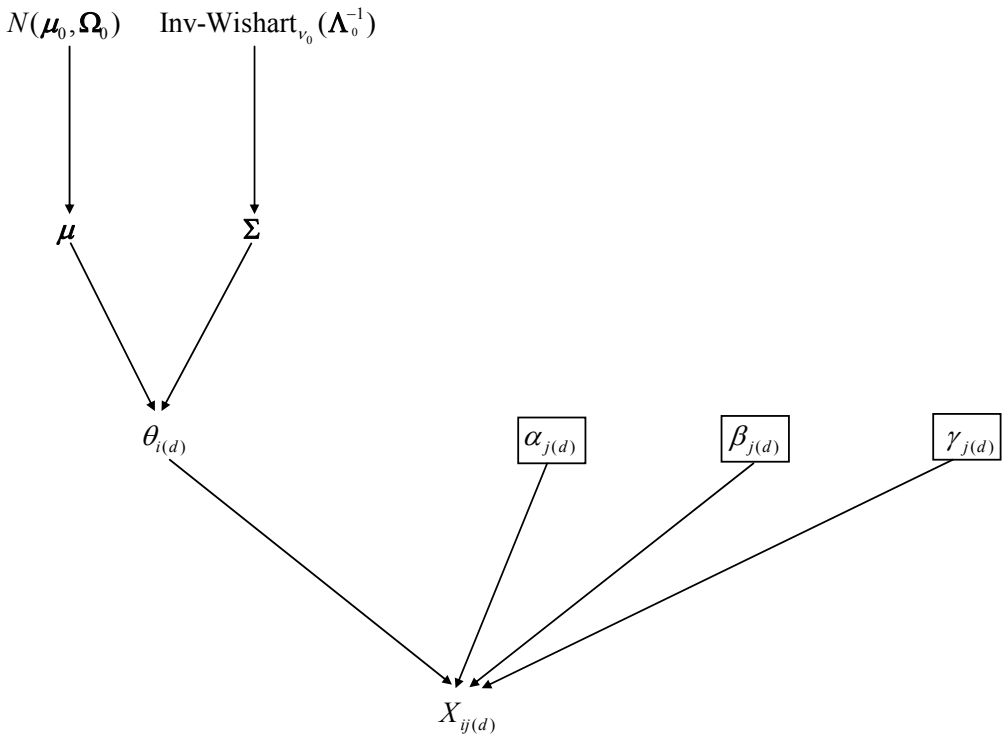
where

- $X_{ij(d)}$ is the response of examinee i to the j^{th} item of dimension d;
- $\theta_{i(d)}$ is the d^{th} component of the vector $\boldsymbol{\theta}_i$ (i.e., $\boldsymbol{\theta}_i = \{\theta_{i(d)}\}$);
- $\alpha_{j(d)}$, $\beta_{j(d)}$, and $\gamma_{j(d)}$ are the parameters of the j^{th} item of dimension d;
- $d = 1, \ldots, D$ (the number of dimensions);
- $j(d) = 1(d), \ldots, J(d)$; and
- $\sum_{d=1}^{D} J(d) = J$.

For this example, we will assume that the item parameters are known (i.e., we are only interested in scoring the responses from multiple tests),

In addition to $\boldsymbol{\Theta} = \{\boldsymbol{\theta}_i\}$, we will also estimate $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$, which represent the *hyperparameters*.

Below is a directed acyclic graph of the of the model. The variables in boxes are assumed to be known.



The likelihood of the data matrix $\mathbf{X}$ is given by

$$
\begin{aligned}
L(\mathbf{X}|\mathbf{\Theta}) &= \prod_{i=1}^{N}\prod_{d=1}^{D}\prod_{j(d)=1}^{J(d)} P_{ij(d)}^{X_{ij(d)}}(1-P_{ij(d)})^{1-X_{ij(d)}} \\
&= \prod_{i=1}^{N} L(\boldsymbol{X}_i|\boldsymbol{\theta}_i).
\end{aligned}
$$

Prior, Posterior, and Full Conditional Distributions

The the prior distribution of $\boldsymbol{\theta}_i$ is parameterized as

$$
\boldsymbol{\theta}_i|\boldsymbol{\mu},\mathbf{\Sigma} \sim N(\boldsymbol{\mu},\mathbf{\Sigma}),
$$

where

$$
\begin{aligned}
\boldsymbol{\mu} &\sim N(\boldsymbol{\mu}_0,\mathbf{\Omega}_0) \\
\mathbf{\Sigma} &\sim \text{Inv-Wishart}_{\nu_0}(\mathbf{\Lambda}_0^{-1}).
\end{aligned}
$$

A Detour: The Inverse-Wishart Distribution

The Wishart distribution is the D-dimensional generalization of the univariate χ^2 distribution.

For a $D \times D$ symmetric, positive definite matrix $\mathbf{\Lambda}$, and $\nu \geq D+2$, the $D \times D$ symmetric, positive definite matrix $\mathbf{\Sigma}^{-1} \sim \text{Wishart}_\nu(\mathbf{\Lambda})$ with probability density function

$$
p(\mathbf{\Sigma}^{-1}|\nu,\mathbf{\Lambda}) = c \cdot |\mathbf{\Sigma}|^{-\frac{v-D-1}{2}} \exp\left[-\frac{1}{2}\text{tr}(\mathbf{\Lambda}^{-1}\mathbf{\Sigma}^{-1})\right],
$$

where $c = \left[2^{\nu D/2}\pi^{D(D-1)/4}\prod_{d=1}^{D}\Gamma\left(\frac{\nu+1-d}{2}\right)\right]^{-1}|\mathbf{\Lambda}|^{-\nu/2}$ and $E[\mathbf{\Sigma}^{-1}] = \nu\mathbf{\Lambda}$.

If $\mathbf{\Sigma}^{-1} \sim \text{Wishart}_\nu(\mathbf{\Lambda})$ then $\mathbf{\Sigma} \sim \text{Inv-Wishart}_\nu(\mathbf{\Lambda}^{-1})$ with probability density function

$$
p(\mathbf{\Sigma}|\nu,\mathbf{\Lambda}^{-1}) \propto |\mathbf{\Sigma}|^{-\frac{v+D+1}{2}} \exp\left[-\frac{1}{2}\text{tr}(\mathbf{\Lambda}\mathbf{\Sigma}^{-1})\right],
$$

and the expected value is $E(\mathbf{\Sigma}) = (\nu-D-1)^{-1}\mathbf{\Lambda}$. The constant of proportionality is also c.

Let

$$
\begin{aligned}
\mathbf{\Sigma} &\sim \text{Inv-Wishart}_{\nu_0}(\mathbf{\Lambda}_0^{-1}) \\
\boldsymbol{\theta}_i|\boldsymbol{\mu}_i,\mathbf{\Sigma} &\sim N(\boldsymbol{\mu}_i,\mathbf{\Sigma})
\end{aligned}
$$

The posterior distribution of $\mathbf{\Sigma}$ is

$$
\mathbf{\Sigma}|\mathbf{\Theta},\boldsymbol{M} \sim \text{Inv-Wishart}_{\nu_N}(\mathbf{\Lambda}_N^{-1})
$$

where $\boldsymbol{M} = \{\boldsymbol{\mu}_i\}$, $\nu_N = \nu_0 + N$, and $\mathbf{\Lambda}_N = \mathbf{\Lambda}_0 + \sum_{i=1}^{N}(\boldsymbol{\theta}_i - \boldsymbol{\mu}_i)(\boldsymbol{\theta}_i - \boldsymbol{\mu}_i)'$.

The derivation of this result is shown below.

$$
\begin{aligned}
p(\boldsymbol{\Sigma}) &\propto |\boldsymbol{\Sigma}|^{-\frac{\nu+D+1}{2}} \exp\left[-\frac{1}{2}\text{tr}(\boldsymbol{\Lambda}\boldsymbol{\Sigma}^{-1})\right], \\
p(\boldsymbol{\theta}_i|\boldsymbol{\mu}_i, \boldsymbol{\Sigma}) &\propto |\boldsymbol{\Sigma}|^{-\frac{1}{2}} \exp\left[-\frac{1}{2}(\boldsymbol{\theta}_i - \boldsymbol{\mu}_i)^{'}\boldsymbol{\Sigma}^{-1}(\boldsymbol{\theta}_i - \boldsymbol{\mu}_i)\right].
\end{aligned}
$$

Using the properties of the trace of a matrix, the likelihood of $\boldsymbol{\theta}_i$ can be written as

$$
p(\boldsymbol{\theta}_i|\boldsymbol{\mu}_i, \boldsymbol{\Sigma}) \propto |\boldsymbol{\Sigma}|^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}\text{tr}\left[(\boldsymbol{\theta}_i - \boldsymbol{\mu}_i)(\boldsymbol{\theta}_i - \boldsymbol{\mu}_i)^{'}\boldsymbol{\Sigma}^{-1}\right]\right\}.
$$

The likelihood of the $\boldsymbol{\Theta}$ is

$$
\begin{aligned}
p(\boldsymbol{\Theta}|\boldsymbol{M}, \boldsymbol{\Sigma}) &\propto \prod_{i=1}^{N} p(\boldsymbol{\theta}_i|\boldsymbol{\mu}_i, \boldsymbol{\Sigma}) \\
&= |\boldsymbol{\Sigma}|^{-\frac{N}{2}} \exp\left\{-\frac{1}{2}\sum_{i=1}^{N}\text{tr}\left[(\boldsymbol{\theta}_i - \boldsymbol{\mu}_i)(\boldsymbol{\theta}_i - \boldsymbol{\mu}_i)^{'}\boldsymbol{\Sigma}^{-1}\right]\right\}.
\end{aligned}
$$

Combining the prior density and the likelihood

$$
\begin{aligned}
p(\boldsymbol{\Sigma}|\boldsymbol{\Theta}, \boldsymbol{M}) &\propto p(\boldsymbol{\Sigma}) \times p(\boldsymbol{\Theta}|\boldsymbol{M}, \boldsymbol{\Sigma}) \\
&= |\boldsymbol{\Sigma}|^{-\frac{\nu+D+1}{2}} \exp\left[-\frac{1}{2}\text{tr}(\boldsymbol{\Lambda}\boldsymbol{\Sigma}^{-1})\right] \\
&\quad |\boldsymbol{\Sigma}|^{-\frac{N}{2}} \exp\left\{-\frac{1}{2}\sum_{i=1}^{N}\text{tr}\left[(\boldsymbol{\theta}_i - \boldsymbol{\mu}_i)(\boldsymbol{\theta}_i - \boldsymbol{\mu}_i)^{'}\boldsymbol{\Sigma}^{-1}\right]\right\} \\
&= |\boldsymbol{\Sigma}|^{-\frac{\nu+N+D+1}{2}} \exp\left\{-\frac{1}{2}\text{tr}\left[(\boldsymbol{\Lambda} + \sum_{i=1}^{N}(\boldsymbol{\theta}_i - \boldsymbol{\mu}_i)(\boldsymbol{\theta}_i - \boldsymbol{\mu}_i)^{'})\boldsymbol{\Sigma}^{-1}\right]\right\} \\
&\sim \text{Inv-Wishart}_{\nu_N}(\boldsymbol{\Lambda}_N^{-1}).
\end{aligned}
$$

Below are the steps in drawing an observation from Inv-Wishart$_\nu(\boldsymbol{\Lambda}^{-1})$.

1. Draw $\boldsymbol{\theta}_i$, $i = 1, \ldots, \nu$, from $N(\mathbf{0}, \boldsymbol{\Lambda})$;
2. Find $\mathbf{S} = \sum_{i=1}^{\nu} \boldsymbol{\theta}_i\boldsymbol{\theta}_i^{'}$;
3. $\mathbf{S}^{-1}$ is a draw from Inv-Wishart$_\nu(\boldsymbol{\Lambda}^{-1})$.

End of Detour

The joint posterior distribution can be expressed as

$$
\begin{aligned}
p(\boldsymbol{\Theta}, \boldsymbol{\mu}, \boldsymbol{\Sigma}|\mathbf{X}) &\propto p(\mathbf{X}|\boldsymbol{\Theta}) \times p(\boldsymbol{\Theta}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) \times p(\boldsymbol{\mu}) \times p(\boldsymbol{\Sigma}) \\
&= \prod_{i=1}^{N} p(\boldsymbol{X}_i|\boldsymbol{\theta}_i) \times p(\boldsymbol{\theta}_i|\boldsymbol{\mu}, \boldsymbol{\Sigma}) \times p(\boldsymbol{\mu}) \times p(\boldsymbol{\Sigma}).
\end{aligned}
$$

Again, this is not a known distribution so we need to sample from the full conditional distributions.

We need the full conditional distributions for $\boldsymbol{\Sigma}$, $\boldsymbol{\mu}$, and $\boldsymbol{\theta}_i$.

The full conditional distribution of $\boldsymbol{\mu}$ is

$$\begin{aligned} p(\boldsymbol{\mu}|\mathbf{X},\boldsymbol{\Sigma},\boldsymbol{\Theta}) &\propto p(\mathbf{X}|\boldsymbol{\mu},\boldsymbol{\Sigma},\boldsymbol{\Theta}) \times p(\boldsymbol{\mu}|\boldsymbol{\Sigma},\boldsymbol{\Theta}) \\ &\propto p(\mathbf{X}|\boldsymbol{\Theta}) \times p(\boldsymbol{\Theta}|\boldsymbol{\mu},\boldsymbol{\Sigma}) \times p(\boldsymbol{\mu}|\boldsymbol{\Sigma}) \\ &\propto p(\boldsymbol{\Theta}|\boldsymbol{\mu},\boldsymbol{\Sigma}) \times p(\boldsymbol{\mu}) \end{aligned}$$

Given the prior distribution and the hyperdistributions above, the full conditional posterior distribution of $\boldsymbol{\mu}$ is

$$\begin{aligned} \boldsymbol{\mu}|\mathbf{X},\boldsymbol{\Sigma},\boldsymbol{\Theta} &\sim N(\boldsymbol{\mu}_N,\boldsymbol{\Omega}_N) \\ &= N\left(\left(\boldsymbol{\Omega}_0^{-1}+N\boldsymbol{\Sigma}^{-1}\right)^{-1}\left(\boldsymbol{\Omega}_0^{-1}\boldsymbol{\mu}_0+N\boldsymbol{\Sigma}^{-1}\bar{\boldsymbol{\theta}}\right),\left(\boldsymbol{\Omega}_0^{-1}+N\boldsymbol{\Sigma}^{-1}\right)^{-1}\right). \end{aligned}$$

This full conditional distribution is a known distribution, and can be sampled directly.

The full conditional distribution of $\boldsymbol{\Sigma}$ is

$$\begin{aligned} p(\boldsymbol{\Sigma}|\mathbf{X},\boldsymbol{\mu},\boldsymbol{\Theta}) &\propto p(\mathbf{X}|\boldsymbol{\mu},\boldsymbol{\Sigma},\boldsymbol{\Theta}) \times p(\boldsymbol{\Sigma}|\boldsymbol{\mu},\boldsymbol{\Theta}) \\ &\propto p(\mathbf{X}|\boldsymbol{\Theta}) \times p(\boldsymbol{\Theta}|\boldsymbol{\mu},\boldsymbol{\Sigma}) \times p(\boldsymbol{\Sigma}|\boldsymbol{\mu}) \\ &\propto p(\boldsymbol{\Theta}|\boldsymbol{\mu},\boldsymbol{\Sigma}) \times p(\boldsymbol{\Sigma}) \end{aligned}$$

Again, given the prior distribution and the hyperdistributions above, the full conditional posterior distribution of $\boldsymbol{\Sigma}$ is

$$\begin{aligned} \boldsymbol{\Sigma}|\mathbf{X},\boldsymbol{\mu},\boldsymbol{\Theta} &\sim \text{Inv-Wishart}_{\nu_N}(\boldsymbol{\Lambda}_N^{-1}) \\ &= \text{Inv-Wishart}_{\nu_0+N}\left(\boldsymbol{\Lambda}_0+\sum_{i=1}^{N}(\boldsymbol{\theta}_i-\boldsymbol{\mu})(\boldsymbol{\theta}_i-\boldsymbol{\mu})'\right). \end{aligned}$$

This full conditional distribution is also a known distribution, and can be sampled directly.

The full conditional distribution of $\boldsymbol{\theta}_i$ is

$$\begin{aligned} p(\boldsymbol{\theta}_i|\mathbf{X},\boldsymbol{\mu},\boldsymbol{\Sigma},\boldsymbol{\Theta}^{(-1)}) &\propto p(\mathbf{X}|\boldsymbol{\mu},\boldsymbol{\Sigma},\boldsymbol{\Theta}) \times p(\boldsymbol{\theta}_i|\boldsymbol{\mu},\boldsymbol{\Sigma},\boldsymbol{\Theta}^{(-1)}) \\ &\propto p(\mathbf{X}^{(-1)}|\boldsymbol{\Theta}^{(-1)}) \times p(\boldsymbol{X}_i|\boldsymbol{\theta}_i) \times p(\boldsymbol{\theta}_i|\boldsymbol{\mu},\boldsymbol{\Sigma}) \\ &\propto p(\boldsymbol{X}_i|\boldsymbol{\theta}_i) \times p(\boldsymbol{\theta}_i|\boldsymbol{\mu},\boldsymbol{\Sigma}) \end{aligned}$$

This full conditional distribution is NOT a known distribution so we have to use the M-H algorithm to sample from this distribution *indirectly*.

At iteration $t+1$, for $i=1,\ldots,N$, sample θ_{id}^* from $N(\theta_{id}^{(t)},\sigma_\epsilon^2)$, and accept $\boldsymbol{\theta}_i^*$ with probability

$$\alpha(\boldsymbol{\theta}_i^{(t)},\boldsymbol{\theta}_i^*) = \min\left(\frac{p(\boldsymbol{X}_i|\boldsymbol{\theta}_i^*)\times p(\boldsymbol{\theta}_i^*|\boldsymbol{\mu},\boldsymbol{\Sigma})}{p(\boldsymbol{X}_i|\boldsymbol{\theta}_i^{(t)})\times p(\boldsymbol{\theta}_i^{(t)}|\boldsymbol{\mu},\boldsymbol{\Sigma})},1\right).$$

Implementation in Ox of the MCMC Algorithm for Estimating $\boldsymbol{\Sigma}$, $\boldsymbol{\mu}$, and $\boldsymbol{\Theta}$

The program `simultaneous.ox` implements the above MCMC algorithm. This program is specifically for $D = 2$, and $J(1) = J(2) = 10$.

It starts with the basic headers.

The first set of global variables describes the characteristics of the data to be analyzed, and variables that will be holding the current values of the parameters to be estimated.

```
decl N=1000,D=2,J=20;//sample size, dimension, total test length
decl Jd=<10;10>;//test length per dimension
decl T=5000,m=2000;//iterations,burn-in
decl X;//data
decl mu0,Sigma0,theta0;//variables holding the current values
```

The next set of global variables are variables needed to generate the data. $\boldsymbol{\mu}$, σ_1^2, σ_2^2, and ρ can be specified here; `theta` will be generated given the hyperparameters, and `alpha`, `beta`, `gamma` will be read from a file.

```
decl mu=<0;-.2>;
decl Sigma;
decl S11=1,rho=.8,S22=1.2;
decl theta,alpha,beta,gamma;
```

The last set of variables are the parameters of the hyperdistributions, σ_ϵ, and the sufficient statitics.

The variable `design` is a $D \times J$ matrix that converts $\boldsymbol{\Theta}_{N\times D}$ into an $N \times J$ matrix $\boldsymbol{\Theta}^*_{N\times D}$ such that θ_{id} is in the appropriately dimension (i.e., d).

```
decl nu_0,Lambda_0;
decl mu_0,Omega_0;
decl cand_sd;
decl smu,sSigma,stheta;
decl ssmu,ssSigma,sstheta;
decl design;
```

The function `ThreePL` computes the 3PL probabilities and requires $\boldsymbol{\Theta}^*_{N\times D}$, and the item parameters.

```
ThreePL(const t,a,b,c){
decl one=ones(N,1),gam,prob;
gam=one*c';
prob=gam+(1-gam)./(1+exp(-1.7*((one*a').*t-one*(a.*b)')));
return prob;
}//ThreePL
```

The function `draw_mu` samples $\boldsymbol{\mu}$ directly from its full conditional distribution given the current values of $\boldsymbol{\Sigma}$ and $\boldsymbol{\Theta}$.

```
draw_mu(){
decl OmegaN=invert(invert(Omega_0)+N*invert(Sigma0));
decl muN=OmegaN*(invert(Omega_0)*mu_0+N*invert(Sigma0)*(meanc(theta0)'));
mu0=choleski(OmegaN)*rann(D,1)+muN;
}//end draw_mu
```

The function `draw_Sigma` samples $\boldsymbol{\Sigma}$ directly from its full conditional distribution given the current values of $\boldsymbol{\mu}$ and $\boldsymbol{\Theta}$.

```
draw_Sigma(){
decl nun=nu_0+N;
decl Lambdan=Lambda_0+((theta0-mu0') '(theta0-mu0'));
decl the=rann(nun,D)*(choleski(invert(Lambdan)))';
Sigma0=invert(the'*the);
}//draw_Sigma
```

The function `logmvd` computes $-\frac{1}{2}(\boldsymbol{\theta}_i - \boldsymbol{\mu})'\boldsymbol{\Sigma}^{-1}(\boldsymbol{\theta}_i - \boldsymbol{\mu})$ given $\boldsymbol{\theta}_i - \boldsymbol{\mu}$.

```
logmvd(decl m){
m=m*choleski(invert(Sigma0));//';
m=-.5*sumr(m.^2);
return m;
}//end logmvd
```

The function `draw_theta` updates $\boldsymbol{\theta}_i$. Note that in the sampling stage, we sample θ_{id} individually, although in the decision stage, $\boldsymbol{\theta}_i$ is accepted as a vector.

```
draw_theta(){
decl theta1=cand_sd*rann(N,D)+theta0;
decl irt0=ThreePL(theta0*design,alpha,beta,gamma);
irt0=sumr(X.*log(irt0)+(1-X).*log(1-irt0))+logmvd(theta0-mu0');
decl irt1=ThreePL(theta1*design,alpha,beta,gamma);
irt1=sumr(X.*log(irt1)+(1-X).*log(1-irt1))+logmvd(theta1-mu0');
decl acc=irt1-irt0;
acc=vecindex(ranu(N,1).<exp(acc));
theta0[acc][]=theta1[acc][];
}//end draw_theta
```

The function `initiate` assigns values to the prior parameters, and and initial values to the parameters to be estimates.

It also sets $\sigma_\epsilon = .1$.

Finally, the function also sets the sufficient statistics to zero.

```
initiate(){
nu_0=D+2;
Lambda_0=unit(D);
mu_0=zeros(D,1);
Omega_0=unit(D);
mu0=zeros(D,1);
Sigma0=unit(D);
theta0=rann(N,D);
cand_sd=.1;
smu=zeros(D,1),sSigma=zeros(D,D),stheta=zeros(N,D);
ssmu=zeros(D,1),ssSigma=zeros(D,D),sstheta=zeros(N,D);
}//end initiate
```

The function `update` updates the sufficient statistics after the burn-in. Note that running sum of squared draws are only for $\boldsymbol{\Theta}$. Thus, we won't estimate the posterior standard deviation of $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$.

```
update(){
smu=smu+mu0;
sSigma=sSigma+Sigma0;
stheta=stheta+theta0;
sstheta=sstheta+theta0.^2;
}// end update
```

The function `print_results` prints the estimates of $\boldsymbol{\mu}$, $\boldsymbol{\Sigma}$, and $\mathbf{R}$, which is the correlation matrix.

In addition, it also print $Cor(\boldsymbol{\theta}, \tilde{\boldsymbol{\theta}})$, $\sqrt{\sum \tilde{V}(\boldsymbol{\theta}_i|\boldsymbol{X}_i)/N}$, and $\sqrt{\sum(\boldsymbol{\theta}_i - \tilde{\boldsymbol{\theta}}_i)^2/N}$ to evaluate the quality of the ability estimates.

```
print_results(){
println("\n*******Hyperparameter Estimates*******");
print("\nEstimate of Mu:",smu'/T);
sSigma=sSigma/T;
print("\nEstimate of Covariance Matrix:",sSigma);
print("\nEstimate of Correlatiopn Matrix:");
decl tmp=sqrt(invert(diag(diagonal(sSigma))));
print(tmp*sSigma*tmp);
tmp=stheta/T;
println("\n*******Ability Estimate Results*******");
print("\nCorr(True,Est):");
print(diagonal(correlation(theta~tmp)[:D-1][D:]));
decl tmp2=sqrt(meanc((sstheta-T*tmp.^2)/(T-1)));
print("Mean Posterior SD:",tmp2);
print("RMSE:",sqrt(meanc((theta-tmp).^2)));
}//print_results
```

The `main` function has several components. The first component reads the ten 3PL item parameters. These parameters are replicated D times to correspond to the total test length J before they are designated as `alpha`, `beta`, and `gamma`.

The code creates a design matrix appropriate for the different test lengths across the D dimensions.

```
tmp=0;
design=zeros(D,J);
for(decl i=0;i<D;++i){
tmp=tmp~sumc(Jd[:i]);
design[i][tmp[i]:tmp[i+1]-1]=ones(1,Jd[i]);
}//end i
```

In general, the design matrix is of the following form:

$$\begin{pmatrix} \mathbf{1}_{1\times J(1)} & \mathbf{0}_{1\times J(2)} & \cdots & \mathbf{0}_{1\times J(D)} \\ \mathbf{0}_{1\times J(1)} & \mathbf{1}_{1\times J(2)} & \cdots & \mathbf{0}_{1\times J(D)} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0}_{1\times J(1)} & \mathbf{0}_{1\times J(2)} & \cdots & \mathbf{1}_{1\times J(D)} \end{pmatrix}_{D\times J}$$

The next few lines create `Sigma` from the variances and covariances, and generates `theta`, and the item response matrix `X`.

The function `initiate` is then called, after which the functions `draw_mu`, `draw_Sigma`, and `draw_theta` are called $T + m$ times. The function `update` is called only when $t > m$.

Lastly, the draws are summarized using the function `print_results`.

1. Change N, $\boldsymbol{\mu}$, σ^2, and ρ to see how they affect the estimates.

2. Double $J(d)$ to examine the effect of test length on the estimates. Which of the estimates are most affected by a longer test?.

3. What happens to the estimates if more discriminating items are involved? Use $\boldsymbol{\alpha}^* = 1.5 \times \boldsymbol{\alpha}$ to generate and analyze the data.

An MCMC Algorithm for Multi-Unidimensional Ability Estimation with Ancillary Variables

In addition to examinees' responses to the test questions, other ancillary information about the examinees is also available in many testing situations.

Examples of these ancillary information include demographic variables such as sex, age, and race, and educational variables such as grade level, courses taken, and previous test scores.

To achieve greater efficiency, in addition to the responses to other tests, we can also incorporate the ancillary information in the ability estimation.

Specifically, we will regress $\boldsymbol{\theta}_i$ on the covariates (i.e., ancillary variables) $\boldsymbol{Y_i}$.

As in the previous example, we will assume a multi-unidimensional structure.

Below is a directed acyclic graph of the model. As before the variables in boxes (i.e., item parameters and covariates) are assumed to be known.

From this model, we can see that the response $X_{ij(d)}$ is only affected by a single ability (i.e., $\theta_{i(d)}$).

However, our inference about $\theta_{i(d)}$ can be improved by taking into consideration $\boldsymbol{\Sigma}$, the covariance between abilities.

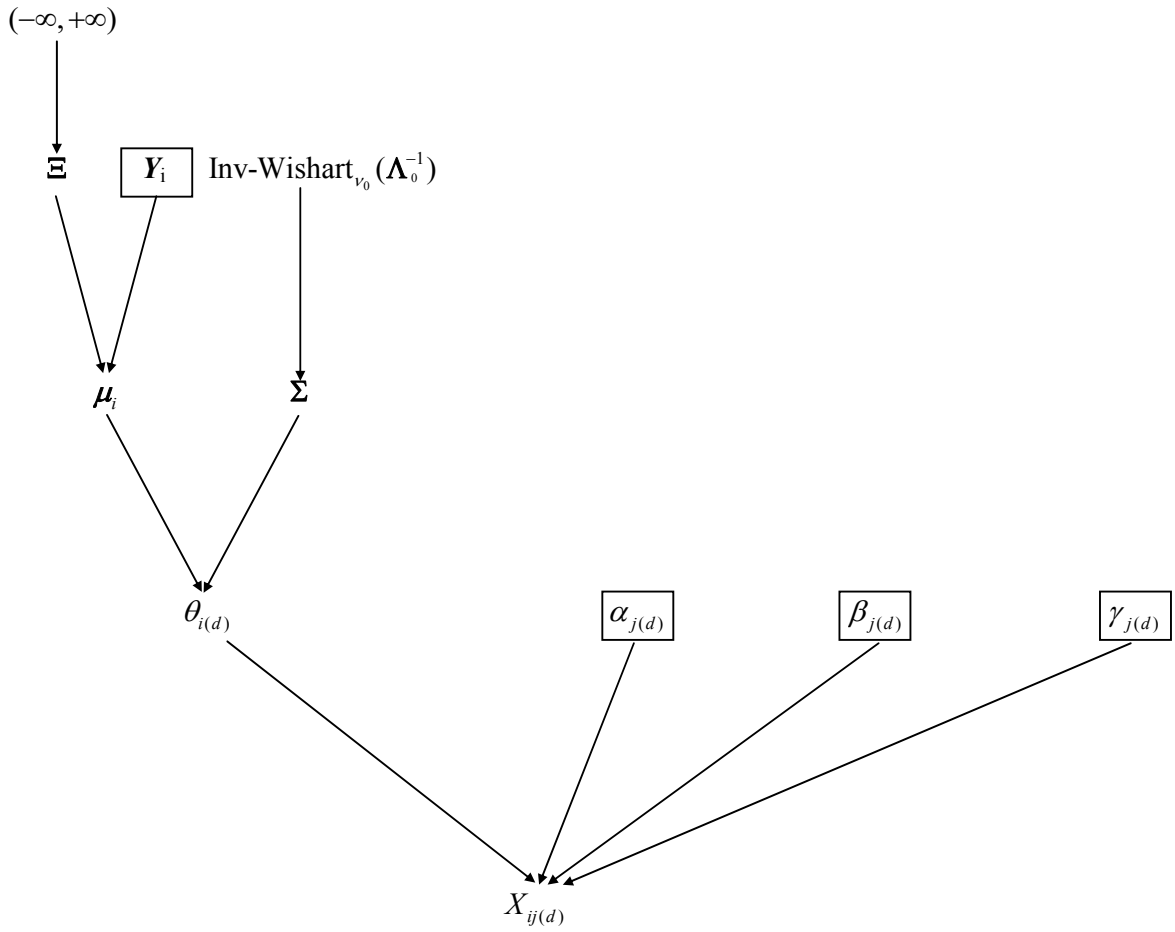
Additionally, by knowing how the abilities and covariates are related, we can further improve our inference about $\boldsymbol{\theta}$ from knowing $\boldsymbol{Y}$.

As we have done in the previous example, the correlation between the abilities are estimated, rather than assumed.

In the same way, we will also empirically determine, rather than assume, the relationship between $\boldsymbol{\theta}$ and $\boldsymbol{Y}$.

Note that, based on the graph, the covariate $\boldsymbol{Y}_i$ determines the mean of $\boldsymbol{\theta}_i$, not the ability itself.

This allows us to have individuals with identical $\boldsymbol{Y}$ to have identical $\boldsymbol{\mu}$, but actually have different $\boldsymbol{\theta}$. The only exception is when $\boldsymbol{Y}$ perfectly determines $\boldsymbol{\theta}$, which is next to impossible. In such a situation, no test needs to be administered.



As before, the likelihood of the data matrix $\mathbf{X}$ is given by

$$\begin{aligned} L(\mathbf{X}|\boldsymbol{\Theta}) &= \prod_{i=1}^{N}\prod_{d=1}^{D}\prod_{j(d)=1}^{J(d)} P_{ij(d)}^{X_{ij(d)}}(1-P_{ij(d)})^{1-X_{ij(d)}} \\ &= \prod_{i=1}^{N} L(\boldsymbol{X}_i|\boldsymbol{\theta}_i). \end{aligned}$$

Prior, Posterior, and Full Conditional Distributions

For examinee i with ability $\boldsymbol{\theta}_i$, the prior distribution is given by

$$\boldsymbol{\theta}_i|\boldsymbol{Y}_i, \boldsymbol{\Xi}, \boldsymbol{\Sigma} \sim N(\boldsymbol{\Xi}'\boldsymbol{Y}_i, \boldsymbol{\Sigma}),$$

with

$$\begin{aligned} \boldsymbol{\Sigma} &\sim \text{Inv-Wishart}_{\nu_0}(\boldsymbol{\Lambda}_0^{-1}), \\ \boldsymbol{\Xi} &\sim (-\infty, +\infty), \end{aligned}$$

where

- $\boldsymbol{Y}_i = \{Y_{i1}, Y_{i2}, \ldots, Y_{ip}\}$ is the vector the p observable covariates of examinee i;
- $\boldsymbol{\Xi} = \{\boldsymbol{\xi_1}, \boldsymbol{\xi_2}, \ldots, \boldsymbol{\xi_d}, \ldots, \boldsymbol{\xi_D}\}$ is the $p \times D$ matrix of regression parameters;
- $\boldsymbol{u}_i = \boldsymbol{\Xi}^{'}\mathbf{Y}_i$ and $\boldsymbol{\Sigma}$ are the mean vector and the covariance matrix of the multivariate normal distribution, respectively;
- ν_0 and $\boldsymbol{\Lambda}_0$ are defined as before; and
- $(-\infty, +\infty)$ represents the improper prior of $\boldsymbol{\Xi}$ to facilitate sampling from the posterior distribution.

The joint distribution of $\boldsymbol{\Xi}$, $\boldsymbol{\Sigma}$, and $\boldsymbol{\Theta}$ given $\mathbf{X}$ and $\mathbf{Y}$ is

$$\begin{aligned} p(\boldsymbol{\Xi}, \boldsymbol{\Sigma}, \boldsymbol{\Theta} | \mathbf{X}, \mathbf{Y}) &\propto p(\mathbf{X}|\mathbf{Y}, \boldsymbol{\Xi}, \boldsymbol{\Sigma}, \boldsymbol{\Theta}) \times p(\boldsymbol{\Theta}|\mathbf{Y}, \boldsymbol{\Xi}, \boldsymbol{\Sigma}) \times p(\boldsymbol{\Sigma}|\mathbf{Y}, \boldsymbol{\Xi}) \times p(\boldsymbol{\Xi}|\mathbf{Y}) \\ &= p(\mathbf{X}|\boldsymbol{\Theta}) \times p(\boldsymbol{\Theta}|\mathbf{Y}, \boldsymbol{\Xi}, \boldsymbol{\Sigma}) \times p(\boldsymbol{\Sigma}) \times p(\boldsymbol{\Xi}). \end{aligned}$$

Below are the full conditional distributions of $\boldsymbol{\Xi}$, $\boldsymbol{\Sigma}$, and $\boldsymbol{\theta}_i$.

For the regression parameters - $\boldsymbol{\Xi}|\boldsymbol{\Sigma}, \boldsymbol{\Theta}, \mathbf{X}, \mathbf{Y}$:

$$\begin{aligned} p(\boldsymbol{\Xi}|\boldsymbol{\Sigma}, \boldsymbol{\Theta}, \mathbf{X}, \mathbf{Y}) &= p(\boldsymbol{\Xi}|\boldsymbol{\Sigma}, \boldsymbol{\Theta}, \mathbf{Y}) \\ &\propto p(\boldsymbol{\Theta}|\boldsymbol{\Xi}, \boldsymbol{\Sigma}, \mathbf{Y}) \times p(\boldsymbol{\Xi}|\boldsymbol{\Sigma}, \mathbf{Y}) \\ &\propto p(\boldsymbol{\Theta}|\boldsymbol{\Xi}, \boldsymbol{\Sigma}, \mathbf{Y}) \times p(\boldsymbol{\Xi}). \end{aligned}$$

For the regression parameters, the full conditional distribution of $\boldsymbol{\Xi}$, $p(\boldsymbol{\Xi}|\boldsymbol{\Theta}, \boldsymbol{\Sigma}, \mathbf{Y})$, given an improper prior is a known distribution called the *matrix-normal*.

Specifically, this full conditional distribution is given by

$$N((\mathbf{Y}'\mathbf{Y})^{-1}\mathbf{Y}'\boldsymbol{\Theta}, \boldsymbol{\Sigma} \otimes (\mathbf{Y}'\mathbf{Y})^{-1}),$$

where $\otimes$ is the Kronecker product. We can sample *directly* from this distribution.

To sample from $p(\boldsymbol{\Xi}|\mathbf{Y}, \boldsymbol{\Sigma}, \boldsymbol{\Theta})$ in *vector* format, one can use the following algorithm:

$$\text{Vec}(\boldsymbol{\Xi}) = \mathbf{Z}_{I \times D}[\boldsymbol{\Sigma} \otimes (\mathbf{Y}'\mathbf{Y})^{-1}]^{1/2} + \text{Vec}((\mathbf{Y}'\mathbf{Y})^{-1}\mathbf{Y}'\boldsymbol{\Theta}),$$

where $\mathbf{Z} \sim \text{MVN}(\mathbf{0}, \mathbf{I})$, and $\text{Vec}(\cdot)$ stacks the vectors of the argument.

As an alternative, we can sample from this full conditional distribution without using the Kronecker product by, first, recasting the matrices as follows:

$$\boldsymbol{\Xi}^{\star}_{(D \times P) \times 1} = \text{Vec}(\boldsymbol{\Xi}),$$

$$\boldsymbol{\Theta}^{\star}_{(N\times D)\times 1} = \text{Vec}(\boldsymbol{\Theta}),$$

$$\boldsymbol{\Sigma}^{\star}_{(N\times D)\times(N\times D)} = \begin{pmatrix} \boldsymbol{\Sigma} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \boldsymbol{\Sigma} & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \boldsymbol{\Sigma} \end{pmatrix},$$

$$\mathbf{Y}^{\star}_{(N\times D)\times(D\times P)} = \begin{pmatrix} \boldsymbol{Y}'_1 & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{0} & \boldsymbol{Y}'_1 & \ldots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \ldots & \boldsymbol{Y}'_1 \\ \boldsymbol{Y}'_2 & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{0} & \boldsymbol{Y}'_2 & \ldots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \ldots & \boldsymbol{Y}'_2 \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ \boldsymbol{Y}'_N & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{0} & \boldsymbol{Y}'_N & \ldots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \ldots & \boldsymbol{Y}'_N \end{pmatrix},$$

where $\boldsymbol{Y}_i$ is the covariate vector of examinee i (i.e., the transpose of the ith row of the design matrix $\mathbf{Y}$).

Then, we can sample from

$$\boldsymbol{\Xi}^{\star} \sim \text{MVN}((\mathbf{Y}^{\star\prime}\boldsymbol{\Sigma}^{\star-1}\mathbf{Y}^{\star})^{-1}\mathbf{Y}^{\star\prime}\boldsymbol{\Sigma}^{\star-1}\boldsymbol{\Theta}^{\star}, (\mathbf{Y}^{\star\prime}\boldsymbol{\Sigma}^{\star-1}\mathbf{Y}^{\star})^{-1}).$$

For the covariance matrix - $\boldsymbol{\Sigma}|\mathbf{X}, \mathbf{Y}, \boldsymbol{\Xi}, \boldsymbol{\Theta}$:

$$\begin{aligned} p(\boldsymbol{\Sigma}|\mathbf{X}, \mathbf{Y}, \boldsymbol{\Xi}, \boldsymbol{\Theta}) &= p(\boldsymbol{\Sigma}|\mathbf{Y}, \boldsymbol{\Xi}, \boldsymbol{\Theta}) \\ &= p(\boldsymbol{\Theta}|\mathbf{Y}, \boldsymbol{\Xi}, \boldsymbol{\Sigma}) \times p(\boldsymbol{\Sigma}|\mathbf{Y}, \boldsymbol{\Xi}) \\ &\propto p(\boldsymbol{\Theta}|\mathbf{Y}, \boldsymbol{\Xi}, \boldsymbol{\Sigma}) \times p(\boldsymbol{\Sigma}). \end{aligned}$$

The full conditional distribution of $\boldsymbol{\Sigma}$, $p(\boldsymbol{\Sigma}|\boldsymbol{\Xi}, \boldsymbol{\Theta}, \mathbf{Y})$, is an Inv-Wishart$_{\nu_N}(\boldsymbol{\Lambda}_N^{-1})$, where $\nu_N = \nu_0 + N$, and $\boldsymbol{\Lambda}_N = \boldsymbol{\Lambda}_0 + \sum(\boldsymbol{\theta_i} - \boldsymbol{\Xi}^{'}\mathbf{Y})(\boldsymbol{\theta_i} - \boldsymbol{\Xi}^{'}\mathbf{Y})^{'}$.

We already know how to sample from this distribution *directly*.

For the ability parameters - $\boldsymbol{\theta}_i|\mathbf{X}, \mathbf{Y}, \boldsymbol{\Xi}, \boldsymbol{\Sigma}, \boldsymbol{\Theta}^{(-i)}$:

$$\begin{aligned} p(\boldsymbol{\theta}_i|, \mathbf{X}, \mathbf{Y}, \boldsymbol{\Xi}, \boldsymbol{\Sigma}, \boldsymbol{\Theta}^{(-i)}) &\propto p(\mathbf{X}|\mathbf{Y}, \boldsymbol{\Xi}, \boldsymbol{\Sigma}, \boldsymbol{\Theta}) \times p(\boldsymbol{\theta}_i|\mathbf{Y}, \boldsymbol{\Xi}, \boldsymbol{\Sigma}, \boldsymbol{\Theta}^{(-i)}) \\ &\propto p(\boldsymbol{X}_i|\boldsymbol{\theta}_i) \times p(\boldsymbol{\theta}_i|\boldsymbol{Y}_i, \boldsymbol{\Xi}, \boldsymbol{\Sigma}). \end{aligned}$$

Again, this full conditional distribution is NOT a known distribution so we have to use the M-H algorithm to sample from this distribution *indirectly*.

At iteration $t+1$, for $i = 1, \ldots, N$, sample θ_{id}^* from $N(\theta_{id}^{(t)}, \sigma_\epsilon^2)$, and accept $\boldsymbol{\theta}_i^*$ with probability

$$\alpha(\boldsymbol{\theta}_i^{(t-1)}, \boldsymbol{\theta}_i^*) = \min\left\{ \frac{p(\mathbf{X}_i|\boldsymbol{\theta}_i^*)p(\boldsymbol{\theta}_i^*|\mathbf{Y}_i, \boldsymbol{\Xi}^{(t)}, \boldsymbol{\Sigma}^{(t)})}{p(\mathbf{X}_i|\boldsymbol{\theta}_i^{(t)})p(\boldsymbol{\theta}_i^{(t)}|\mathbf{Y}_i, \boldsymbol{\Xi}^{(t)}, \boldsymbol{\Sigma}^{(t)})}, \ 1 \right\}.$$

The above model specification simultaneously accounts for responses to other tests and ancillary variables in scoring examinees' responses.

In the traditional method of scoring, where tests scored one at a time and no covariates are used, $\boldsymbol{\Xi} = \mathbf{0}$ and $\{\boldsymbol{\Sigma}\}_{dd'} = 0$ for $d \neq d'$.

When tests are scored simultaneously without the benefit of ancillary variables, only $\boldsymbol{\Sigma}$ is estimated whereas $\boldsymbol{\Xi} = \mathbf{0}$.

Finally, when ancillary variables are used to score one test at a time, both $\boldsymbol{\Xi}$ and $\boldsymbol{\Sigma}$ are estimated, with the constraint that $\{\boldsymbol{\Sigma}\}_{dd'} = 0$ for $d \neq d'$.

When $\boldsymbol{\Sigma}$ is treated as a diagonal matrix, the algorithm may need to be substantially altered to estimate the diagonal elements of $\boldsymbol{\Sigma}$ individually.

It should be noted that the $\boldsymbol{\Sigma}$ in this model is the covariance of $\boldsymbol{\theta}$ *conditional* on $\boldsymbol{Y}$, as in $\boldsymbol{\Sigma}_{\boldsymbol{\theta}|\boldsymbol{Y}}$.

However, we might also be interested in making inference about $\boldsymbol{\Sigma}_{\boldsymbol{\theta}\boldsymbol{\theta}}$, which is the *marginal* covariance of $\boldsymbol{\theta}$.

To do so, we just need to express $\boldsymbol{\Sigma}_{\boldsymbol{\theta}\boldsymbol{\theta}}|\mathbf{X}, \mathbf{Y}$ as $f(\mathbf{X}, \mathbf{Y}, \boldsymbol{\Xi}, \boldsymbol{\Sigma}, \boldsymbol{\Theta})$, where, at iteration $t+1$:

$$\boldsymbol{\Sigma}_{\boldsymbol{\theta}\boldsymbol{\theta}}^{(t+1)} = \frac{1}{N-1}\sum_{i=1}^{N}(\boldsymbol{\theta_i}^{(t+1)} - \boldsymbol{\mu}^{(t+1)})(\boldsymbol{\theta_i}^{(t+1)} - \boldsymbol{\mu}^{(t+1)})'.$$

From the draws, we can also make inference about $\boldsymbol{\Sigma}_{\boldsymbol{\theta}\boldsymbol{\theta}}|\mathbf{X}, \mathbf{Y}$.

A Detour: A More General Method of Generating $\boldsymbol{Y}_{P\times 1}$ and $\boldsymbol{\theta}_{D\times 1}$ Jointly

Previously, we discussed generating $\boldsymbol{Y}_i$ and $\boldsymbol{\theta}_i$ jointly such that they follow a particular structure.

One structure is that $\boldsymbol{\theta} \sim N(\mathbf{0}, \mathbf{R})$. This structure is needed if the item parameters are also to be estimated because, without such a constraint, the model will not be identified, hence, not estimable.

Another structure is the $\boldsymbol{\Sigma}_{\boldsymbol{Y}\boldsymbol{\theta}} = \{\sqrt{\psi/P}\}$, which is due to the setting $\psi_d = \psi$.

Lastly, we fixed the *conditional* covariance $\boldsymbol{\Sigma}_{\boldsymbol{\theta}|\boldsymbol{Y}}$, and allowed the *marginal* covariance $\boldsymbol{\Sigma}_{\boldsymbol{\theta}\boldsymbol{\theta}}$ to vary.

In the current method, we will allow $\boldsymbol{\Sigma}$ to not necessarily be R, ψ_d to not necessarily be ψ, and $\boldsymbol{\Sigma}_{\boldsymbol{\theta}|\boldsymbol{Y}}$ to vary while fixing $\boldsymbol{\Sigma}_{\boldsymbol{\theta}\boldsymbol{\theta}}$.

As before, we will let $\mathbf{Y} \sim N(\mathbf{0}, \mathbf{I})$. In practice, we can use standardization or principal component analysis to transform $\mathbf{Y}$.

In addition, we will let the matrix of regression coefficients $\mathbf{\Xi} = \{\xi_{dp}\} = \{\sigma_d \times \sqrt{\psi_d / P}\}$. This is but one way of splitting ψ_p into the regression weights ξ_{dp} such that

$$\frac{\sum_{d=1}^{D} \xi_{dp}^2}{\sigma_d^2} = \psi_d.$$

Following are the steps in generating N $\boldsymbol{\theta}$ observations with a *marginal* covariance of θ is $\mathbf{\Sigma}_{\boldsymbol{\theta}\boldsymbol{\theta}}$, and where the proportion variance accounted for by the covariates with respect to $\boldsymbol{\theta}_d$ is equal to ψ_d.

1. Generate a $Y_{N \times P}$ matrix of covariates, as in, N observations from $N(\mathbf{0}, \mathbf{I})$. Standardize $\mathbf{Y}$.
2. Compute $\mathbf{M} = \mathbf{Y}\mathbf{\Xi}$, the matrix of *conditional* mean vectors for the N examinees.
3. Compute $\mathbf{\Sigma}_{\boldsymbol{\theta}|\boldsymbol{Y}} = \mathbf{\Sigma}_{\boldsymbol{\theta}\boldsymbol{\theta}} - \mathbf{\Sigma}_{\boldsymbol{\theta}\mathbf{Y}}\mathbf{\Sigma}_{\mathbf{Y}\boldsymbol{\theta}}$.
4. Generate $\boldsymbol{\theta}_i$ from $N(\{\mathbf{M}\}_i', \mathbf{\Sigma}_{\boldsymbol{\theta}|\boldsymbol{Y}})$, $i = 1, 2, \ldots, N$, where $\{\mathbf{M}\}_i$ is the i^{th} row of $\mathbf{M}$.

Additional generalizations can be made. For example $\boldsymbol{\mu}_{\boldsymbol{\theta}}$ need not be equal to $\mathbf{0}$.

Implementation in Ox of the Method for Generating $\boldsymbol{Y}$ and $\boldsymbol{\theta}$ Jointly

The program `ancillary_gen.ox` can be used to generate $\boldsymbol{Y}$ and $\boldsymbol{\theta}$ that follow specific structure. In addition, it also generates the item response data that can be analyzed using the `ancillary.ox`.

The program needs to read the ten 3PL item parameters currently contained in the file `best10.txt`.

This program is specifically written for $D = 2$, and $P = 2$, and $J(1) = J(2) = 10$. The number of examinees N can be changed.

```
decl N=1000,D=2,P=2,J=20;//sample size, dimension, total test length
decl Jd=<10;10>;//test length per dimension
```

The proportion of variance account for by $\mathbf{Y}$ in each dimension (i.e., ψ_1, ψ_2) can be modified.

Similarly the variances of $\boldsymbol{\theta}$, σ_1^2 and σ_2^2, can be specified. Instead of σ_{12}, users can specify ρ.

```
decl psi1=.8,psi2=.5;
decl S11=1,rho=.8,S22=1.2;
```

In addition to the screen output, which provides statistics describing the generated $\boldsymbol{Y}$ and $\boldsymbol{\theta}$, the program will also create an output file called "ancillary.dat".

This file will contain $\mathbf{X}_{N \times 20}$, $\mathbf{Y}_{N \times 2}$ and $\mathbf{\Theta}_{N \times 2}$, in this order.

End Detour

Implementation in Ox of the MCMC Algorithm for Estimating $\boldsymbol{\Xi}$, $\boldsymbol{\Sigma}$, $\boldsymbol{\Theta}$ and $\boldsymbol{\Sigma_{\theta\theta}}$

The program `ancillary.ox` implements the above MCMC algorithm. This program is designed to be used with `ancillary_gen.ox`, and will estimate $\boldsymbol{\Xi}$, $\boldsymbol{\Sigma}$, and $\boldsymbol{\theta}_i$, and $\boldsymbol{\Sigma_{\theta\theta}}$.

The program includes the usual headers.

The first set of global variables is very similar to that in `simultaneous.ox`. However, this time, we need a variable `P` to denote the number of covariates, and `Y` to denote the vector of covariates.

We are also declaring `Xi0` to hold the current values of $\boldsymbol{\Xi}$, and `iYtY` which is equal to $(\mathbf{Y}'\mathbf{Y})^{-1}$. By declaring `iYtY`, we can compute this variable only once in the entire estimation process even though it will use it each iteration.

```
decl N=1000,D=2,P=2,J=20;//sample size, dimension, total test length
decl Jd=<10;10>;//test length per dimension
decl T=5000,m=2000;//iterations,burn-in
decl X,Y;//data, covariates
decl Xi0,Sigma0,mSigma0,theta0;//variables holding the current values
decl theta,alpha,beta,gamma;// abilities, item parameters
decl design,iYtY;//design matrix, inverse of Y'Y
```

The next set of global variables represent the priors of the hyperparameters, σ_θ^2, and the sufficient statistics. We are using an improper prior for $\boldsymbol{\Xi}$ so we do not have parameters to declare for this variable.

```
decl nu_0,Lambda_0;
decl cand_sd;
decl sXi,sSigma,smSigma,stheta;
decl ssXi,ssSigma,ssmSigma,sstheta;
```

The next few lines pertain to the function `ThreePL`, which is unaltered from before.

The function `draw_Xi` updates $\boldsymbol{\Xi}$.

We are using the first of the methods for sampling $\boldsymbol{\Xi}$ discussed above. In Ox, `**` represents $\otimes$.

```
draw_Xi(){
decl b0;
b0=(vec(rann(P,D)))'*(choleski(Sigma0**iYtY))'+(vec(iYtY*(Y'theta0)))';
Xi0=shape(b0,P,D);
}//end draw_Xi
```

The function `draw_Sigma` updates the conditional covariance $\boldsymbol{\Sigma}_{\boldsymbol{\theta}|\boldsymbol{Y}}$.

That this is identical to the `draw_Sigma` function earlier except that instead of a common $\boldsymbol{\mu}$, each individual gets its own $\boldsymbol{\mu}_i = \boldsymbol{\Xi}'\boldsymbol{Y}_i$.

```
draw_Sigma(){
decl nun=nu_0+N;
decl Lambdan=Lambda_0+(theta0-Y*Xi0)'(theta0-Y*Xi0);
decl the=rann(nun,D)*(choleski(invert(Lambdan)))';
Sigma0=invert(the'the);
}//draw_Sigma
```

The function `logmvd` is unchanged.

The function `draw_theta` is different from `draw_theta` function only in one respect: $\boldsymbol{\mu}_i$ is used in place of $\boldsymbol{\mu}$.

```
draw_theta(){
decl theta1=cand_sd*rann(N,D)+theta0;
decl irt0=ThreePL(theta0*design,alpha,beta,gamma);
irt0=sumr(X.*log(irt0)+(1-X).*log(1-irt0))+logmvd(theta0-Y*Xi0);
decl irt1=ThreePL(theta1*design,alpha,beta,gamma);
irt1=sumr(X.*log(irt1)+(1-X).*log(1-irt1))+logmvd(theta1-Y*Xi0);
decl acc=irt1-irt0;
acc=vecindex(ranu(N,1).<exp(acc));
theta0[acc][]=theta1[acc][];
}//end draw_theta
```

The function `initiate` remains largely the same, except that it has been adapted to accommodate $\boldsymbol{\Xi}$.

The function `update` updates a subset of sufficient statistics (i.e., those necessary for estimating $\boldsymbol{\Xi}$, $\boldsymbol{\Sigma}$, $\boldsymbol{\Theta}$, and $\boldsymbol{\Sigma}_{\boldsymbol{\theta}\boldsymbol{\theta}}$, and the posterior variance of $\boldsymbol{\theta}_i$.

Note that the `smSigma`, the sufficient statistic for $\boldsymbol{\Sigma}_{\boldsymbol{\theta}\boldsymbol{\theta}}$, is not sampled, only updated.

```
update(){
sXi=sXi+Xi0;
sSigma=sSigma+Sigma0;
smSigma=smSigma+theta0'theta0/(N-1);
stheta=stheta+theta0;
sstheta=sstheta+theta0.^2;
}// end update
```

The function `print_results` is largely similar to its previous definition. However, this time, we print separately for the *conditional* and *marginal* covariance and correlation matrices.

In the `main` function, we read in the ten item parameters before they are replicated, which is exactly the same process used in generating the data.

We also read in the data, which this time is composed of $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{\Theta}$. We need to ensure that the right columns are assigned to the right variables, as in, `X`, `Y` and `theta`.

The last three steps involve initiating the variables, sampling from the posterior, which include updating the sufficient statistics after the burn-in, and printing the results.

1. For $\rho = .8$ and $\sigma_1^2 = \sigma_2^2 = 1$, what values of $\psi_1 = \psi_2$ will lead to *conditional* correlation of $\mathbf{I}$? What does this tell us about the relationship between θ_1 and θ_2 vis-à-vis $\boldsymbol{Y}$?

2. Change N, ψ_d, σ^2, and ρ to see how they affect the estimates.

By comparing $\psi_d \neq 0$ or $\rho \neq 0$ to zero against $\psi_d = 0$ or $\rho = 0$, we can evaluate the impact of incorporating the covariates and responses to other tests in the estimation process.

3. Double $J(d)$ to examine the effect of test length on the estimates. Which of the estimates are most affected by a longer test?.

4. What happens to the estimates if more discriminating items are involved? Again, use $\boldsymbol{\alpha}^* = 1.5 \times \boldsymbol{\alpha}$ to generate and analyze the data.

Additional Exercises

1. Change `simultaneous.ox` to accommodate $D = 4$ dimensions.

Let $MSE_{\rho=0}$ and MSE_{ρ} be the mean squared error of the ability estimates from scoring the tests *one at a time* and scoring the tests, respectively.

The ratio $RE = MSE_{\rho=0}/MSE_{\rho}$ is the relative efficiency due to simultaneous scoring.

For example, an $RE = 1.5$ can be interpreted as gaining 50% in efficiency by taking into account responses from other tests. That is, the tests now function as if they are $1.5 \times J(d)$ long.

Find the MSE of $\boldsymbol{\theta}$ when $D = 4$, $J(d) = 10$, and $\rho = 0, .5, .9$. Average the MSE across the D dimensions.

What is the RE under this set-up?

What happens when D is reduced to 2 or when $J(d)$ is increased to 20?

2. Impact of Incorporating Ancillary Information in Ability Estimation

2ai) For a particular data set, find the EAP of $\boldsymbol{\theta}$ *one ability at a time.*

2aii) For the same data, use `simultaneous.ox`. You need to modify the code so it can to read in the same data used as above.

2aiii) Finally, use `ancillary.ox` to analyze the same data.

2b) Compare the three estimates based on $Corr(\boldsymbol{\theta}, \tilde{\boldsymbol{\theta}})$, mean posterior variance, and root-mean squared error.

Comparing 2ai and 2aii allows us to examine the advantage of *multidimensional* over *unidimensional* scoring

Comparing 2aii and 2aiii allows us to examine the advantage of using covariates on top of *multidimensional* scoring.

Comparing 2ai and 2aiii allows us to examine the impact of *multidimensional* scoring and use of covariates over *unidimensional* scoring.

3. Manipulate a few of the variables and repeat the analysis to determine under which conditions the different procedure yield similar/different results.

Optional Exercises

1. Modify `ancillary.ox` so that $\boldsymbol{\Xi}$ can be sampled without using the Kronecker product $\otimes$.

2. Simultaneous Calibration: Sampling $\mathbf{R}$

The presence of multiple tests can also be used to improve item parameter estimation. The impact of multiple abilities on the item parameters is *indirect*, but can be substantial if the sample size is not sufficiently large.

One way of ensuring that the model is determined is to constraint the ability distribution, as in, $\boldsymbol{\theta} \sim N(\mathbf{0}, \mathbf{R})$.

The full conditional distribution of $\mathbf{R}$ can be shown to be

$$p(\mathbf{R}|\mathbf{X}, \boldsymbol{\Theta}, \mathbf{B}) \propto p(\boldsymbol{\Theta}|\mathbf{R}) \times p(\mathbf{R}),$$

where $\mathbf{B}$ is the matrix of item parameters.

Because $\mathbf{R}$, we can define $p(\mathbf{R})$ so its full conditional distribution is an Inverse-Wishart.

Although we know how to sample directly from this full conditional distribution, the candidate value $\boldsymbol{\Sigma}^*$ is not guaranteed to be $\mathbf{R}$

To ensure that we only sample correlation matrices, we need to first convert $\boldsymbol{\Sigma}^* \rightarrow \mathbf{R}^*$, and then accept it with probability

$$\alpha(\mathbf{R}^{(t)}, \mathbf{R}^*) = \min\left(\exp\left[\frac{D+1}{2}\left(\log \mathbf{R}^* - \log \mathbf{R}^{(t)}\right)\right], 1\right).$$

Write a program that can simultaneously calibrate D tests. Manipulate N, D, and ρ to see their impact on the item parameter estimates.

3. Higher-Order IRT

When multiple abilities are involved, we can also posit that higher-order ability, say, ω, such that

$$p(\boldsymbol{\theta}_i|\omega_i) = \prod_{d=1}^{D} p(\theta_{id}|\omega_i),$$

where $\theta_{id} = \lambda_d \times \omega_i + \epsilon_{id}$, $|\lambda_d| \leq 1$, and $\epsilon_{id} \sim N(0, 1 - \lambda_d^2)$.

If we assume $\omega \sim N(0, 1)$, then

$$\begin{aligned} \theta_{id}|\omega_i &\sim N(\lambda_d \times \omega_i, 1 - \lambda_d^2) \\ \theta_{id} &\sim N(0, 1) \end{aligned}$$

It can be shown that ω, θ_d, and $\theta_{d'}$ are related as follows:

$$\begin{aligned} Corr(\omega, \theta_d) &= \lambda_d \\ Corr(\theta_d, \theta_{d'}) &= \lambda_d \times \lambda_{d'} \end{aligned}$$

The full conditional distribution of ω_i^* can be shown to be

$$N\left(\sigma_\omega^{2(t+1)} \times \sum_{d=1}^{D} \frac{\lambda_d^{(t)} \times \theta_{id}^{(t)}}{1 - \lambda_d^{2(t)}}, \sigma_\omega^{2(t+1)}\right),$$

where $\sigma_\omega^{2(t+1)} = \lambda_d^{2(t)} / \left[1 - \lambda_d^{2(t)}\right]$. We can sample from this distribution directly.

However, the remaining parameters (i.e., $\boldsymbol{\theta}_i$ and λ_d) need to be sampled using M-H.

Note that because the marginal distribution of θ_d is already $N(0, 1)$, we can use this model to simultaneously calibrate multiple tests.

Again, we need M-H to sample $\mathbf{B}$.

Write a program that can simultaneously calibrate D tests using the HO-IRT model. Manipulate N, D, and ρ to see their impact on the item parameter estimates.

In addition, for $D = 4$, create a correlation matrix $\mathbf{R}$ such that the correlations cannot be fully written as $\lambda_d \times \lambda_{d'}$.

Compare the quality of the estimates obtained using the HO-IRT model and the model for simultaneous estimation.