OpenMx User Guide

Release 2.6.7

Steven M. Boker **Michael C. Neale**
Hermine H. Maes **Michael J. Wilde**
Michael Spiegel **Timothy R. Brick** **Ryne Estabrook**
Timothy C. Bates **Paras Mehta** **Timo von Oertzen**
Ross J. Gore **Michael D. Hunter** **Daniel C. Hackett**
Julian Karch **Andreas Brandmaier**
Joshua N. Pritikin **Mahsa Zahery**
Robert M. Kirkpatrick

June 24, 2016

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Beginners Guide to OpenMx

This document is a basic introduction to OpenMx. It assumes that the reader has installed the R statistical programming language [http://www.r-project.org/] and the OpenMx library for R [http://openmx.psyc.virginia.edu]. Detailed introductions to R can be found on the internet. We recommend [http://faculty.washington.edu/tlumley/Rcourse] for a short course but Google search for 'introduction to R' provides many options.

The OpenMx scripts for the examples in this guide are available in the following files:

- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/OneFactorPathDemo.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/OneFactorMatrixDemo.R
- http://www.vipbg.vcu.edu/OpenMx/html/NewBeginnersGuide.R

1.1.1 Basic Introduction

The main function of OpenMx is to design a statistical model which can be used to test a hypothesis with a set of data. The way to do this is to use one of the functions written in the R language as part of the OpenMx package. The function to create an Mx model is (you guessed it) `mxModel()`. Note that i) R is case sensitive and ii) the OpenMx package must be loaded before any of the OpenMx functions are used (this only has to be done once in an R session).

```
require(OpenMx)
```

We start by building a model with the `mxModel()` function, which takes a number of arguments. We explain each of these below. Usually we save the result of applying the `mxModel()` function as an R object, here called *myModel*.

```
myModel <- mxModel()
```

This R object has a special class named MxModel. We might read the above line 'myModel gets mxModel left paren right paren'. In this case we have provided no arguments in the parentheses, but R has still created an empty MxModel object *myModel*. Obviously this model needs arguments to do anything useful, but just to get the idea of the process of 'Model Building - Model Fitting' here is how we would go about fitting this model. The *myModel* object becomes the argument of the OpenMx function `mxRun()` to fit the model. The result is stored in a new R object, *myModelRun* of the same class as the previous object but updated with the results from the model fitting.

```
myModelRun <- mxRun(myModel)
```

A model typically contains data, free and/or fixed parameters and some function to be applied to the data according to the model. Models can be build in a variety of ways, due to the flexibility of OpenMx which it inherited from classic **Mx** - for those of you who familiar with that software - and further expanded.

One possibility for building models is to create all the elements as separate R objects, which are then combined by calling them up in the `mxModel()` statement. We will refer to this approach as the **piecewise style**.

A second type is to start by creating an `mxModel()` with one element, and then taking this model as the basis for the next model where you add another element to it and so on. This also provides a good way to make sure that each element is syntactically correct before adding anything new. We will call this approach the **iterative/recursive/stepwise style**.

Another approach more closely resembles the traditional classic **Mx** approach, where you specify all the elements of the model at once, all as arguments of the `mxModel()`, separated by comma's. While this approach is often more compact and works well for scripts that run successfully, it is not the most recommended approach for debugging a new script. We will refer to this approach as the **classic style**.

1.1.2 A First mxModel

To introduce the model fitting process in OpenMx, we will present the basics of several OpenMx functions which can be used to write a simple model and view its contents.

Matrix Creation

Although `mxModel()` can have a range of arguments, we will start with the most simple one. Models are fitted to data which must be in numeric format (for continuous data) or factor format (for ordinal data). Here we consider continuous data. Numbers (data/parameter estimates) are typically put into matrices, except for fixed constants. The function created to put numbers into matrices is (unsurprisingly) `mxMatrix()`. Here we start with a basic matrix call and make use of only some of its possible arguments. All arguments are separated by comma's. To make it clear and explicit, we will include the names of the arguments, although that is optional if the arguments are included in the default order.

```
myAmatrix <- mxMatrix(type="Full", nrow=1, ncol=1, values=4, name="Amatrix")
```

The above call to the `mxMatrix()` function has five arguments. The `type` and `name` arguments are alphanumeric and therefore their values are in quotes. The `nrows`, `ncols` and `values` arguments are numeric, and refer respectively to the number of rows, the number of columns of the matrix and the value for the (in this case only one) element of the matrix.

Matrix Contents

Once you have run/executed this statement in R, a new R object has been created, namely *myAmatrix*. When you view its contents, you'll notice it has a special class of object, made by OpenMx, called an MxMatrix object. This object has a number of attributes, all of which are listed when you call up the object.

```
> myAmatrix
FullMatrix 'Amatrix'

$labels: No labels assigned.

$values
  [,1]
[1,]    4

$free: No free parameters.

$lbound: No lower bounds assigned.

$ubound: No upper bounds assigned.
```

Most of these attributes start with the $ symbol. The contents of a particular attribute can be displayed by typing the name of the R object followed by the $ symbol and the name of the attribute, for example here we're displaying the values of the matrix *myAmatrix*

```
> myAmatrix$values
      [,1]
[1,]    4
```

Note that the attribute name is part of the header of the output but is not displayed as an $ attribute. However, it does exist as one and can be seen by typing

```
> myAmatrix$name
[1] "Amatrix"
```

Wait a minute, this is confusing. The matrix has a name, here "Amatrix", and the R object to represent the matrix has a name, here "myAmatrix". Remember that when you call up *myAmatrix* you get the contents of the entire MxMatrix R object. When you call up "Amatrix", you get

```
Error: object 'Amatrix' not found
```

unless you had previously created another R object with that same name. Why do we need two names? The matrix name (here, "Amatrix") is used within OpenMx when performing an operation on this matrix using algebra (see below) or manipulating/using the matrix in any way within a model. When you want to manipulate/use/view the matrix outside of OpenMx, or build a model by building each of the elements as R objects in the 'piecewise' approach, you use the R object name (here, *myAmatrix*). Let's clarify this with an example.

Model Creation

First, we will build a model *myModel1* with just one matrix. Obviously that is not very useful but it does serve to introduce the sequence of creating a model and running it.

```
myModel1     <- mxModel( mxMatrix(type="Full", nrow=1, ncol=1, values=4, name="Amatrix") )
```

Model Execution

The mxRun() function will run a model through the optimizer. The return value of this function is an identical MxModel object, with all the free parameters - in case there are any - in the elements of the matrices of the model assigned to their final values.

```
myModel1Run  <- mxRun(myModel1)
```

Model Contents

Note that we have saved the result of applying mxRun() to *myModel1* into a new R object, called *myModel1Run* which is of the same class as *myModel1* but with values updated after fitting the model. Note that the MxModel is automatically given a name 'untitled2' as we did not specify a name argument for the mxModel() function.

```
> myModel1Run
MxModel 'untitled2'
type : default
$matrices : 'Amatrix'
$algebras :
$constraints :
$intervals :
$latentVars : none
```

```
$manifestVars : none
$data : NULL
$submodels :
$expectation : NULL
$fitfunction : NULL
$compute : NULL
$independent : FALSE
$options :
$output : TRUE
```

As you can see from viewing the contents of the new object, the current model only uses two of the arguments, namely `$matrices` and `$output`. Given the matrix was specified within the mxModel, we can explore its arguments by extending the level of detail as follows.

```
> myModel1Run$matrices
$Amatrix
FullMatrix 'Amatrix'

$labels: No labels assigned.

$values
     [,1]
[1,]    4

$free: No free parameters.

$lbound: No lower bounds assigned.

$ubound: No upper bounds assigned.
```

This lists all the matrices within the MxModel *myModel1Run*. In the current case there is only one. If we want to display just a specific argument of that matrix, we first add a dollar sign `$`, followed by the name of the matrix, and an `$` sign prior to the required argument. Thus both arguments within an object and specific elements of the same argument type are preceded by the `$` symbol.

```
> myModel1run$matrices$Amatrix$values
      [,1]
 [1,]    4
```

It is also possible to omit the `$matrices` part and use the more succinct `myModel1Run$Amatrix$values`.

Similarly, we can inspect the output which also includes the matrices in `$matrices`, but only displays the values. Furthermore, the output will list algebras (`$algebras`), model expectations (`$expectations`), status of optimization (`$status`), number of evaluations (`$evaluations`), openmx version (`$mxVersion`), and a series of time measures of which the CPU time might be most useful (`$cpuTime`).

```
> myModel1Run$output
$matrices
$matrices$untitled2.Amatrix
     [,1]
[1,]    4

....
$mxVersion
[1] "999.0.0-3297"

$frontendTime
Time difference of 0.05656791 secs
```

```
$backendTime
Time difference of 0.003615141 secs

$independentTime
Time difference of 3.385544e-05 secs

$wallTime
Time difference of 0.0602169 secs

$timestamp
[1] "2014-04-10 09:53:37 EDT"

$cpuTime
Time difference of 0.0602169 secs
```

Alternative

Now let's go back to the model *myModel1* for a minute. We specified the matrix "Amatrix" within the model. Given we had previously saved the "Amatrix" in the *myAmatrix* object, we could have just used the R object as the argument of the model as follows. Here we're adding one additional element to the `MxModel()` object, namely the `name` argument

```
myModel2     <- mxModel(myAmatrix, name="model2")
myModel2Run  <- mxRun(myModel2)
```

You can verify for yourself that the contents of *myModel2* is identical to that of *myModel1*, and the same applies to *myModel1Run* and *myModel2Run*, and as a result to the matrix contained in the model. The value of the matrix element is still 4, both in the original model and the fitted model, as we did not manipulate the matrix in any way. We refer to this alternative style of coding as **iterative**.

Algebra Creation

Now, let's take it one step further and use OpenMx to evaluate some matrix algebra. It will come as a bit of a shock to learn that the OpenMx function to specify an algebra is called `mxAlgebra()`. Its main argument is the `expression`, in other words the matrix algebra formula you want to evaluate. In this case, we're simply adding 1 to the value of the matrix element, providing a name for the matrix "Bmatrix" and then save the new matrix as *myBmatrix*. Note that the matrix we are manipulating is the "Amatrix", the name given to the matrix within OpenMx.

```
myBmatrix    <- mxAlgebra(expression=Amatrix+1, name="Bmatrix")
```

Algebra Contents

We can view the contents of this new matrix. Notice that the result has not yet computed, as we have not run the model yet.

```
> myBmatrix
mxAlgebra 'Bmatrix'
$formula:  Amatrix + 1
$result: (not yet computed) <0 x 0 matrix>
dimnames: NULL
```

Built Model

Now we can combine the two statements - one defining the matrix, and the other defining the algebra - in one model, simply by separating them by a comma, and run it to see the result of the operation.

```
myModel3     <- mxModel(myAmatrix, myBmatrix, name="model3")
myModel3Run  <- mxRun(myModel3)
```

First of all, let us view *myModel3* and more specifically the values of the matrices within that model. Note that the `$matrices` lists one matrix, "Amatrix", and that the `$algebras` lists another, "Bmatrix". To view values of matrices created with the `mxMatrix()` function, the argument is `$values`; for matrices created with the `mxAlgebra()` function, the argument is `$result`. Note that when viewing a specific matrix, you can omit the `$matrices` or the `$algebras` arguments.

```
> myModel3
MxModel 'model3'
type : default
$matrices : 'Amatrix'
$algebras : 'Bmatrix'
$constraints :
$intervals :
$latentVars : none
$manifestVars : none
$data : NULL
$submodels :
$expectation : NULL
$fitfunction : NULL
$compute : NULL
$independent : FALSE
$options :
$output : FALSE

> myModel3$Amatrix$values
     [,1]
[1,]    4

> myModel3$Bmatrix$result
<0 x 0 matrix>
```

Fitted Model

Given we're looking at the model *myModel3* before it is run, results of algebra have not been computed yet. Let us see how things change after running the model and viewing *myModel3Run*.

```
> myModel3Run
MxModel 'model3'
type : default
$matrices : 'Amatrix'
$algebras : 'Bmatrix'
$constraints :
$intervals :
$latentVars : none
$manifestVars : none
$data : NULL
$submodels :
$expectation : NULL
$fitfunction : NULL
```

```
$compute : NULL
$independent : FALSE
$options :
$output : TRUE

> myModel3Run$Amatrix$values
     [,1]
[1,]    4

> myModel3Run$Bmatrix$result
     [,1]
[1,]    5
```

You will notice that the structure of the MxModel objects is identical, the value of the "Amatrix" has not changed, as it was a fixed element. However, the value of the "Bmatrix" is now the result of the operation on the "Amatrix". Note that we're here looking at the "Bmatrix" within the MxModel object *myModel3Run*. Please verify that the original MxAlgebra objects *myBmatrix* and *myAmatrix* remain unchanged. The `mxModel()` function call has made its own internal copies of these objects, and it is only these internal copies that are being manipulated. In computer science terms, this is referred to as *pass by value*.

Pass By Value

Let us insert a mini-lecture on the R programming language. Our experience has found that this exercise will greatly increase your understanding of the OpenMx language.

As this is such a crucial concept in R (unlike many other programming languages), let us look at it in a simple R example. We will start by assigning the value 4 to the object *avariable*, and then display it. If we then add 1 to this object, and display it again, notice that the value of *avariable* has not changed.

```
> avariable <- 4
> avariable
[1] 4
> avariable +1
[1] 5
> avariable
[1] 4
```

Now we introduce a function, as OpenMx is a collection of purposely built functions. The function takes a single argument (the object *number*), adds one to the argument *number* and assigns the result to *number*, and then returns the incremented number back to the user. This function is given the name `addone()`. We then apply the function to the object *avariable*, as well as display *avariable*. Thus, the objects *addone* and *avariable* are defined. The object assigned to *addone* is a function, while the value assigned to *avariable* is the number 4.

```
> addone <- function(number) {
    number <- number + 1
    return(number)
    }

> addone(avariable)
[1] 5
> avariable
[1] 4
```

Note that it may be prudent to use the `print()` function to display the results back to the user. When R is run from a script rather than interactively, results will not be displayed unless the function `print()` is used as shown below.

```
> print(addone(avariable))
[1] 5
> print(avariable)
[1] 4
```

What is the result of executing this code? Try it. The correct results are 5 and 4. But why is the object *avariable* still 4, even after the `addone()` function was called? The answer to this question is that R uses pass-by-value function call semantics.

In order to understand pass-by-value semantics, we must understand the difference between *objects* and *values*. The *objects* declared in this example are *addone*, *avariable*, and *number*. The *values* refer to the things that are stored by the *objects*. In programming languages that use pass-by-value semantics, at the beginning of a function call it is the *values* of the argument list that are passed to the function.

The object *avariable* cannot be modified by the function `addone()`. If I wanted to update the value stored in the object, I would have needed to replace the expression as follows:

```
> print(avariable <- addone(avariable))
[1] 5
> print(avariable)
[1] 5
```

Try it. The updated example prints out 5 and 5. The lesson from this exercise is that the only way to update a object in a function call is to capture the result of the function call [1]. This lesson is sooo important that we'll repeat it:

the only way to update an object in a function call is to capture the result of the function call.

R has several built-in types of values that you are familiar with: numerics, integers, booleans, characters, lists, vectors, and matrices. In addition, R supports S4 object values to facilitate object-oriented programming. Most of the functions in the OpenMx library return S4 object values. You must always remember that R does not discriminate between built-in types and S4 object types in its call semantics. Both built-in types and S4 object types are passed by value in R (unlike many other languages).

1.1.3 Styles

In the beginning of the introduction, we discussed three styles of writing OpenMx code: the piecewise, stepwise and classic styles. Let's take the most recent model and show how it can be written in these three styles.

Piecewise Style

The style we used in *myModel3* is the piecewise style. We repeat the different statements here for clarity

```
myAmatrix    <- mxMatrix(type="Full", nrow=1, ncol=1, values=4, name="Amatrix")
myBmatrix    <- mxAlgebra(expression=Amatrix+1, name="Bmatrix")

myModel3     <- mxModel(myAmatrix, myBmatrix, name="model3")
myModel3Run  <- mxRun(myModel3)
```

Each argument of the `mxModel()` statement is defined separately first as independent R objects which are then combined in one model statement.

[1] There are a few exceptions to this rule, but you can be assured such trickery is not used in the OpenMx library.

Stepwise Style

For the stepwise style, we start with an `mxModel()` with just one argument, as we originally did with the "Amatrix" in *myModel1*, as repeated below. We could run this model to make sure it's syntactically correct.

```
myModel1     <- mxModel( mxMatrix(type="Full", nrow=1, ncol=1, values=4, name="Amatrix") )
myModel1Run  <- mxRun(myModel1)
```

Then we would build a new model starting from the first model. To do this, we invoke a special feature of the first argument of an `mxModel()`. If it is the name of a saved MxModel object, for example *myModel1*, the arguments of that model would be automatically included in the new model. These arguments can be changed (or not) and new arguments can be added. Thus, in our example, where we want to keep the "Amatrix" and add the "Bmatrix", our second model would look like this.

```
myModel4     <- mxModel(myModel1,
                mxAlgebra(expression=Amatrix+1, name="Bmatrix"),
                name="model4"
                )
myModel4Run  <- mxRun(myModel4)
```

Note that we call it "model4", by adding a `name` argument to the `mxModel()` as to not overwrite our previous "model1".

Classic Style

The final style may be reminiscent of classic Mx. Here we build all the arguments explicitly within one `mxModel()`. As a result only one R object is created prior to `mxRun()` ning the model. This style is more compact than the others but harder to debug.

```
myModel5     <- mxModel(
                mxMatrix(type="Full", nrow=1, ncol=1, values=4, name="Amatrix"),
                mxAlgebra(expression=Amatrix+1, name="Bmatrix"),
                name="model5"
                )
myModel5Run  <- mxRun(myModel5)
```

You may have seen an alternative version with the first argument in quotes. In that case, that argument refers to the name of the model and not to a previously defined model. Thus, the following specification is identical to the previous one. Note also that it is not necessary to add the 'names' of the arguments, as long as the arguments are listed in their default order, which can easily be verified by using the standard way to get help about a function (in this case `?mxMatrix()`).

```
myModel5     <- mxModel("model5",
                mxMatrix(type="Full", nrow=1, ncol=1, values=4, name="Amatrix"),
                mxAlgebra(expression=Amatrix+1, name="Bmatrix")
                )
myModel5run  <- mxRun(myModel5)
```

Note that all arguments are separated by commas. In this case, we've also separated the arguments on different lines, but that is only for clarity. No comma is needed after the last argument! If you accidentally put one in, you get the generic error message *'argument is missing, with no default'* meaning that you forgot something and R doesn't know what it should be. The bracket on the following line closes the `mxModel()` statement.

1.1.4 Data functions

Most models will be fitted to data, not just a single number. We will briefly introduce how to read data that are pre-packaged with the OpenMx library as well as reading in your own data. All standard R utilities can be used here. The critical part is to run an OpenMx model on these data, thus another OpenMx function `mxData()` is needed.

Reading Data

The `data` function can be used to read sample data that has been pre-packaged into the OpenMx library. One such sample data set is called "demoOneFactor".

```
data(demoOneFactor)
```

In order to read your own data, you will most likely use the `read.table`, `read.csv`, `read.delim` functions, or other specialized functions available from CRAN to read from 3rd party sources. We recommend you install the package **psych** which provides succinct descriptive statistics with the `describe()` function.

```
require(psych)
describe(demoOneFactor)
```

The output of this function is shown below.

```
   var   n  mean   sd median trimmed  mad   min  max range  skew kurtosis   se
x1   1 500 -0.04 0.45  -0.03   -0.04 0.46 -1.54 1.22  2.77 -0.05     0.01 0.02
x2   2 500 -0.05 0.54  -0.03   -0.04 0.55 -2.17 1.72  3.89 -0.14     0.05 0.02
x3   3 500 -0.06 0.61  -0.03   -0.05 0.58 -2.29 1.83  4.12 -0.17     0.23 0.03
x4   4 500 -0.06 0.73  -0.08   -0.05 0.75 -2.48 2.45  4.93 -0.08     0.05 0.03
x5   5 500 -0.08 0.82  -0.08   -0.07 0.89 -2.62 2.18  4.80 -0.10    -0.23 0.04
```

Now that the data are accessible in R, we need to make them readable into our OpenMx model.

Data Source

A `mxData()` function is used to construct a data source for the model. OpenMx can handle fitting models to summary statistics and to raw data.

The most commonly used **summary statistics** are covariance matrices, means and correlation matrices; information on the variances is lost/unavailable with correlation matrices, so these are usually not recommended.

These days, the standard approach for model fitting applications is to use **raw data**, which is simply a data table or rectangular file with columns representing variables and rows representing subjects. The primary benefit of this approach is that it handles datasets with missing values very conveniently and appropriately.

Covariance Matrix

We will start with an example using summary data, so we are specifying a covariance matrix by using the R function `cov` to generate a covariance matrix from the data frame. In addition to reading in the actual covariance matrix as the first (`observed`) argument, we specify the `type` (one of "cov","cor","sscp" and "raw") and the number of observations (`numObs`).

```
exampleDataCov <- mxData(observed=cov(demoOneFactor), type="cov", numObs=500)
```

We can view what *exampleDataCov* looks like for OpenMx.

```
> exampleDataCov
MxData 'data'
type : 'cov'
numObs : '500'
Data Frame or Matrix :
       x1        x2        x3        x4        x5
x1 0.1985443 0.1999953 0.2311884 0.2783865 0.3155943
x2 0.1999953 0.2916950 0.2924566 0.3515298 0.4019234
x3 0.2311884 0.2924566 0.3740354 0.4061291 0.4573587
x4 0.2783865 0.3515298 0.4061291 0.5332788 0.5610769
x5 0.3155943 0.4019234 0.4573587 0.5610769 0.6703023
Means : NA
Acov : NA
Thresholds : NA
```

Some models may include predictions for the mean(s). We could add an additional `means` argument to the `mxData` statement to read in the means as well.

```
exampleDataCovMeans <- mxData(observed=cov(demoOneFactor),
                              means=colMeans(demoOneFactor), type="cov", numObs=500)
```

The output for *exampleDataCovMeans* would have the following extra lines.

```
....
Means :
              x1          x2          x3          x4          x5
[1,] -0.04007841 -0.04583873 -0.05588236 -0.05581416 -0.07555022
```

Raw Data

Note that for most real life examples, raw data are the preferred option, except in cases where complete data are available on all variables included in the analyses. In that situation, using summary statistics is faster. To change the current example to use raw data, we would read in the data explicitly and specify the `type` as "raw". The `numObs` is no longer required as the sample size is counted automatically.

```
exampleDataRaw <- mxData(observed=demoOneFactor, type="raw")
```

Printing this MxData object would result in listing the whole data set. We show just the first few lines here:

```
> exampleData
MxData 'data'
type : 'raw'
numObs : '500'
Data Frame or Matrix :
               x1            x2           x3           x4           x5
1   -1.086832e-01 -0.4669377298 -0.177839881 -0.080931127 -0.070650263
2   -1.464765e-01 -0.2782619339 -0.273882553 -0.154120074  0.092717293
3   -6.399140e-01 -0.9295294042 -1.407963429 -1.588974090 -1.993461644
4    2.150340e-02 -0.2552252972  0.097330513 -0.117444884 -0.380906486
5     ....
```

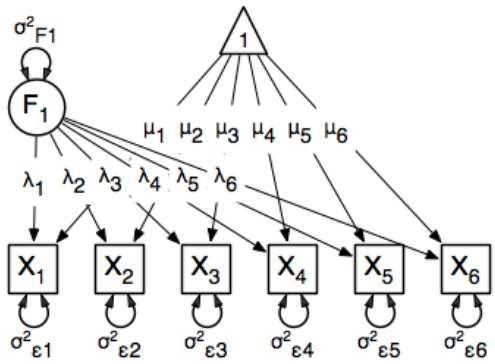
The data to be used for our example are now ready in either **covariance matrix** or **raw data** format.

1.1.5 Model functions

We introduce here several new features by building a basic factor model to real data. A useful tool to represent such a model is drawing a path diagram which is mathematically equivalent to equations describing the model. If you're not

familiar with the method of path analysis, we suggest you read one of the key reference books [LI1986].

Briefly, squares are used for observed variables; latent variables are drawn in circles. One-headed arrows are drawn to represent causal relationships. Correlations between variables are represented with two-headed arrows. Double-headed paths are also used for variances of variables. Below is a figure of a one factor model with five indicators (x1..x5). We have added a value of 1.0 to the variance of the latent variable **G** as a fixed value. All the other paths in the models are considered free parameters and are to be estimated.



Variables

To specify this path diagram in OpenMx, we need to indicate which variables are observed or manifest and which are latent. The `mxModel()` arguments `manifestVars` and `latentVars` both take a vector of variable names. In this case the manifest variables are "x1", "x2", "x3", "x4", "x5" and the latent variable is "G". The R function `c()` is used to build the vectors.

```
manifests <- c("x1","x2","x3","x4","x5")
latents <- c("G")

manifestVars = manifests
latentVars = latents
```

This could be written more succinctly as follows.

```
manifestVars = names(demoOneFactor)
latentVars = c("G")
```

because the R `names()` function call returns the vector of names that we want (the observed variables in the data frame "demoOneFactor").

Path Creation

Paths are created using the `mxPath()` function. Multiple paths can be created with a single invocation of the `mxPath()` function.

- The `from` argument specifies the path sources, and the `to` argument specifies the path sinks. If the `to` argument is missing, then it is assumed to be identical to the `from` argument.
- The `connect` argument specifies the type of the source to sink connection, which can be one of five types. For our example, we use the default "single" type in which the i^{th} element of the `from` argument is matched with the i^{th} element of the `to` argument, in order to create a path.
- The `arrows` argument specifies whether the path is unidirectional (single-headed arrow, "1") or bidirectional (double-headed arrow, "2").

- The next three arguments are vectors: `free`, is a boolean vector that specifies whether a path is free or fixed; `values` is a numeric vector that specifies the starting value of the path; `labels` is a character vector that assigns a label to each free or fixed parameter. Paths with the same labels are constrained to be equal, and OpenMx insists that paths equated in this way have the same fixed or free status; if this is not the case it will report an error.

To specify the path model above, we need to specify three different sets of paths. The first are the single-headed arrows from the latent to the manifest variables, which we will put into the R object *causalPaths* as they represent causal paths. The second set are the residuals on the manifest variables, referred to as *residualVars*. The third `mxPath()` statement fixes the variance of the latent variable to one, and is called *factorVars*.

```
causalPaths  <- mxPath(from=latents, to=manifests)
residualVars <- mxPath(from=manifests, arrows=2)
factorVars   <- mxPath(from=latents, arrows=2, free=FALSE, values=1.0)
```

Note that several arguments are optional. For example, we omitted the `free` argument for *causalPaths* and *residualVars* because the default is 'TRUE' which applies in our example. We also omitted the `connect` argument for all three paths. The default "single" type automatically generates paths from every variable back to itself for all the variances, both the *residualVars* or the *factorVars*, as neither of those statements includes the `to` argument. For the *causalPaths*, the default `connect` type will generate separate paths from the latent to each of the manifest variables. To keep things simple, we did not include `values` or `labels` arguments as they are not strictly needed for this example, but this may not be true in general. Once the variables and paths have been specified, the predicted covariance matrix will be generated from the implied path diagram in the backend of OpenMx using the RAM notation (see below).

Equations

For those more in tune with equations and matrix algebra, we can represent the model using matrix algebra rather than path specifications. For reasons that may become clear later, the expression for the expected covariances between the manifest variables is given by

$$\text{Cov}(x_{ij}) = facLoadings * facVariances * facLoadings' + resVariances$$

where *facLoadings* is a column vector of factor loadings, *facVariances* is a symmetric matrix of factor variances and *resVariances* is a diagonal matrix of residual variances. You might have noticed the correspondence between *causalPaths* and *facLoadings*, between *residualVars* and *resVariances*, and between *factorVars* and *facVariances*. To translate this model into OpenMx using the matrix specification, we will define the three matrices first using the `mxMatrix()` function, and then specify the algebra using the `mxAlgebra()` function.

Matrix Creation

The next three lines create three `MxMatrix()` objects, using the `mxMatrix()` function. The first argument declares the `type` of the matrix, the second argument declares the number of rows in the matrix (`nrow`), and the third argument declares the number of columns (`ncol`). The `free` argument specifies whether an element is a free or fixed parameter. The `values` argument specifies the starting values for the elements in the matrix, and the `name` argument specifies the name of the matrix.

```
mxFacLoadings  <-  mxMatrix(type="Full", nrow=5, ncol=1,
                            free=TRUE, values=0.2, name="facLoadings")
mxFacVariances <-  mxMatrix(type="Symm", nrow=1, ncol=1,
                            free=FALSE, values=1, name="facVariances")
mxResVariances <-  mxMatrix(type="Diag", nrow=5, ncol=5,
                            free=TRUE, values=1, name="resVariances")
```

Each `MxMatrix()` object is a container that stores five matrices of equal dimensions. The five matrices stored in a `MxMatrix()` object are: `free`, `values`, `labels`, `lbound`, and `ubound`. `Free` stores a boolean vector

that determines whether a element is free or fixed. `Values` stores the current values of each element in the matrix. `Labels` stores a character label for each element in the matrix. And `lbound` and `ubound` store the lower and upper bounds, respectively, for each element that is a free parameter. If a element has no label, lower bound, or upper bound, then an NA value is stored in the element of the respective matrix.

Algebra Creation

An `mxAlgebra()` function is used to construct an expression for any algebra, in this case the expected covariance algebra. The first argument (`expression`) is the algebra expression that will be evaluated by the numerical optimizer. The matrix operations and functions that are permitted in an MxAlgebra expression are listed in the help for the `mxAlgebra` function (obtained by `?mxAlgebra`). The algebra expression refers to entities according to `name` argument of the MxMatrix objects.

```
mxExpCov      <- mxAlgebra(expression=facLoadings %*% facVariances %*% t(facLoadings)
                           + resVariances, name="expCov")
```

You can see a direct correspondence between the formula above and the expression used to create the expected covariance matrix *myExpCov*.

1.1.6 Expectation - Fit Function

To fit a model to data, the differences between the observed covariance matrix (the data, in this case the summary statistics) and model-implied expected covariance matrix are minimized using a fit function. Fit functions are functions for which free parameter values are chosen such that the value of the fit function is minimized. Now that we have specified data objects and path or matrix/algebra objects for the predicted covariances of our model, we need to link the two and execute them which is typically done with `mxExpectation()` and `mxFitFunction()` statements. PS. These two statements replace the `mxObjective()` functions‘‘ in earlier versions of OpenMx.

RAM Expectation

When using a path specification of the model, the fit function is always `RAM` which is indicated by using the `type` argument. We don't have to specify the fit function explicitly with an `mxExpectation()` and `FitFunction()` argument, instead we simply add the following argument to the model.

```
type="RAM"
```

To gain a better understanding of the RAM principles, we recommend reading [RAM1990]

Normal Expectation

When using a matrix specification, `mxExpectationNormal()` defines how model expectations are calculated using the matrices/algebra implied by the `covariance` argument and optionally the `means`. For this example, we are specifying an expected covariance algebra (`covariance`) omitting an expected means algebra. The expected covariance algebra is referenced according to its name, i.e. the `name` argument of the MxAlgebra created above. We also need to assign `dimnames` for the rows and columns of this covariance matrix, such that a correspondence can be determined between the expected and the observed covariance matrices. Subsequently we are specifying a maximum likelihood fit function with the `mxFitFunctionML()` statement.

```
expectCov     <- mxExpectationNormal(covariance="expCov",
                                     dimnames=names(demoOneFactor))
funML         <- mxFitFunctionML()
```

The above expectation and fit function can be used when fitting to covariance matrices. A model for the predicted means is optional. However, when fitting to raw data, an expectation has to be used that specifies both a model for the means and for the covariance matrices, paired with the appropriate fit function. In the case of raw data, the `mxFitFunctionML()` function uses full-information maximum likelihood to provide maximum likelihood estimates of free parameters in the algebra defined by the `covariance` and `means` arguments. The `covariance` argument takes an `MxMatrix` or `MxAlgebra` object, which defines the expected covariance of an associated `MxData` object. Similarly, the `means` argument takes an `MxMatrix` or `MxAlgebra` object to define the expected means of an associated `MxData` object. The `dimnames` arguments takes an optional character vector. This vector is assigned to be the `dimnames` of the means vector, and the row and columns `dimnames` of the covariance matrix.

```
expectCovMeans <- mxExpectationNormal(covariance="expCov", means="expMeans",
                                      dimnames=names(demoOneFactor))
funML          <- mxFitFunctionML()
```

Raw data can come in two forms, continuous or categorical. While **continuous data** have an unlimited number of possible values, their frequencies typically form a normal distribution.

There are basically two flavors of **categorical data**. If only two response categories exist, for example Yes and No, or affected and unaffected, we are dealing with binary data. Variables with three or more ordered categories are considered ordinal.

Continuous Data

When the data to be analyzed are continuous, and models are fitted to raw data, the `mxFitFunctionML()` function will take two arguments, the `covariance` and the `means` argument, as well as `dimnames` to match them up with the observed data.

```
expectRaw      <- mxExpectationNormal(covariance="expCov", means="expMeans",
                                      dimnames=manifests)
funML          <- mxFitFunctionML()
```

If the variables to be analyzed have at least 15 possible values, we recommend to treat them as continuous data. As will be discussed later in the documentation, the power of the study is typically higher when dealing with continuous rather than categorical data.

Categorical Data

For categorical - be they binary or ordinal - data, an additional argument is needed for the `mxFitFunctionML()` function, besides the `covariance` and `means` arguments, namely the `thresholds` argument.

```
expFunOrd      <- mxExpectationNormal(covariance="expCov", means="expMeans",
                                      thresholds="expThres", dimnames=manifests)
funML          <- mxFitFunctionML()
```

For now, we will stick with the factor model example and fit it to covariance matrices, calculated from the raw continuous data.

1.1.7 Methods

We have introduced two ways to create a model. One is the **path method**, in which observed and latent variables are specified as well as the causal and correlational paths that connect the variables to form the model. This method may be more intuitive as the model maps on directly to the diagram. This of course assumes that the path diagram is drawn mathematically correct. Once the model is 'drawn' or specified correctly in this way, OpenMx translates the paths into RAM notation for the predicted covariance matrices.

Alternatively, we can specify the model using the **matrix method** by creating the necessary matrices and combining them using algebra to generate the expected covariance matrices (and optionally the mean/threshold vectors). Although less intuitive, this method provides greater flexibility for developing more complex models. Let us look at examples of both.

Path Method

We have previously generated all the pieces that go into the model, using the path method specification. As we have discussed before, the `mxModel()` function is somewhat of a swiss-army knife. The first argument to the `mxModel()` function can be an argument of type `name` (and appear in quotes), in which case it is a newly generated model, or it can be a previously defined model object. In the latter case, the new model 'inherits' all the characteristics (arguments) of the old model, which can be changed with additional arguments. An `mxModel()` can contain `mxData()`, `mxPath()`, `mxExpectation()`, `mxFitFunction` and other `mxModel()` statements as arguments.

The following `mxModel()` function is used to create the 'one-factor' model, shown on the path diagram above. The first argument is a `name`, thus we are specifying a new model, called "One Factor". By specifying the `type` argument to equal "RAM", we create a path style model. A RAM style model must include a vector of manifest variables (`manifestVars=`) and a vector of latent variables (`latentVars=`). We then include the arguments for reading the example data *exampleDataCov*, and those that specify the paths of the path model *causalPaths*, *residualVars*, and *factorVars* which we created previously.

```
factorModel1 <- mxModel(name="One Factor",
    type="RAM",
    manifestVars=manifests,
    latentVars=latents,
    exampleDataCov, causalPaths, residualVars, factorVars)
```

When we display the contents of this model, note that we now have manifest and latent variables specified. By using `type``="RAM" we automatically use the expectation ``mxExpectationRAM` which translates the path model into RAM specification [RAM1990] as reflected in the matrices **A**, **S** and **F**, and the function `mxFitFunctionML()`. Briefly, the **A** matrix contains the asymmetric paths, which are the unidirectional paths in the *causalPaths* object, and represent the factor loadings from the latent variable onto the manifest variables. The **S** matrix contains the symmetric paths which include both the bidirectional paths in *residualVars* and in *factorVars*. The **F** matrix is the filter matrix.

The formula $F(I-A)^{-}1*S*(I-A)^{-}1'F'$, where I is an identity matrix, $^{-}1$ denotes the inverse and ' the transpose, generates the expected covariance matrix.

```
> factorModel1
MxModel 'One Factor'
type : RAM
$matrices : 'A', 'S', and 'F'
$algebras :
$constraints :
$intervals :
$latentVars : 'G'
$manifestVars : 'x1', 'x2', 'x3', 'x4', and 'x5'
$data : 5 x 5
$data means : NA
$data type: 'cov'
$submodels :
$expectation : MxExpectationRAM
$fitfunction : MxFitFunctionML
$compute : NULL
$independent : FALSE
$options :
$output : FALSE
```

You can verify that after running the model, the new R object *factorFit* has similar arguments, except that they now contain the estimates from the model rather than the starting values. For example, we can look at the values in the **A** matrix in the built model *factorModel*, and in the fitted model *factorFit*. We will get back to this later. Note also that from here on out, we use the convention the R object containing the built model will end with *Model* while the R object containing the fitted model will end with *Fit*.

```
factorFit1 <- mxRun(factorModel1)
```

We can inspect the values of the **A** matrix in *factorModel1* and *factorFit1* respectively as follows.

```
> factorModel1$A$values
   x1 x2 x3 x4 x5 G
x1  0  0  0  0  0 0
x2  0  0  0  0  0 0
x3  0  0  0  0  0 0
x4  0  0  0  0  0 0
x5  0  0  0  0  0 0
G   0  0  0  0  0 0

> factorFit1$A$values
   x1 x2 x3 x4 x5         G
x1  0  0  0  0  0 0.3971521
x2  0  0  0  0  0 0.5036611
x3  0  0  0  0  0 0.5772414
x4  0  0  0  0  0 0.7027737
x5  0  0  0  0  0 0.7962500
G   0  0  0  0  0 0.0000000
```

We can also specify all the arguments directly within the `mxModel()` function, using the **classical** style, as follows. The script reads data from disk, creates the one factor model, fits the model to the observed covariances, and prints a summary of the results.

```
data(demoOneFactor)
manifests <- names(demoOneFactor)
latents   <- c("G")

factorModel1 <- mxModel(name="One Factor",
    type="RAM",
    manifestVars=manifests,
    latentVars=latents,
    mxPath(from=latents, to=manifests),
    mxPath(from=manifests, arrows=2),
    mxPath(from=latents, arrows=2, free=FALSE, values=1.0),
    mxData(observed=cov(demoOneFactor), type="cov", numObs=500)
)

factorFit1 <- mxRun(factorModel1)
summary(factorFit1)
```

For more details about the summary and alternative options to display model results, see below.

Matrix Method

We will now re-create the model from the previous section, but this time we will use a matrix specification technique. The script reads data from disk, creates the one factor model, fits the model to the observed covariances, and prints a summary of the results.

We have already created separate objects for each of the parts of the model, which can then be combined in an

`mxModel()` statement at the end. To repeat ourselves, the name of an OpenMx entity bears no relation to the R object that is used to identify the entity. In our example, the object "mxFacLoadings" stores a value that is a MxMatrix object with the name "facLoadings". Note, however, that it is not necessary to use different names for the name within the `mxMatrix` object and the name of the R object generated with the statement. For more complicated models, using the same name for both rather different entities, may make it easier to keep track of the various pieces. For now, we will use different names to highlight which one should be used in which context.

```
data(demoOneFactor)

factorModel2 <- mxModel(name="One Factor",
    exampleDataCov, mxFacLoadings, mxFacVariances, mxResVariances,
    mxExpCov, expectCov, funML)
factorFit2 <- mxRun(factorModel2)
summary(factorFit2)
```

Alternatively, we can write the script in the **classical** style and specify all the matrices, algebras, objective function and data as arguments to the `mxModel()`.

```
data(demoOneFactor)

factorModel2 <- mxModel(name="One Factor",
    mxMatrix(type="Full", nrow=5, ncol=1, free=TRUE, values=0.2, name="facLoadings"),
    mxMatrix(type="Symm", nrow=1, ncol=1, free=FALSE, values=1, name="facVariances"),
    mxMatrix(type="Diag", nrow=5, ncol=5, free=TRUE, values=1, name="resVariances"),
    mxAlgebra(expression=facLoadings %*% facVariances %*% t(facLoadings)
                        + resVariances, name="expCov"),
    mxExpectationNormal(covariance="expCov", dimnames=names(demoOneFactor)),
    mxFitFunctionML(),
    mxData(observed=cov(demoOneFactor), type="cov", numObs=500)
)

factorFit2 <- mxRun(factorModel2)
summary(factorFit2)
```

Now that we've specified the model with both methods, we can run both examples and verify that they indeed provide the same answer by inspecting the two fitted R objects *factorFit1* and *factorFit2*.

1.1.8 Output

We can generate output in a variety of ways. As you might expect, the **summary** function summarizes the model, including data, model parameters, goodness-of-fit and run statistics.

Note that the fitted model is an R object that can be further manipulated, for example, to output specific parts of the model or to use it as a basis for developing an alternative model.

Model Summary

The summary function (`summary(modelname)`) is a convenient method for displaying the highlights of a model after it has been executed. Many R functions have an associated `summary()` function which summarizes all key aspects of the model. In the case of OpenMx, the `summary(model)` includes a summary of the data, a list of all the free parameters with their name, matrix element locators, parameter estimate and standard error, as well as lower and upper bounds if those were assigned. Currently the list of goodness-of-fit statistics printed include the number of observed statistics, the number of estimated parameters, the degrees of freedom, minus twice the log-likelihood of the data, the number of observations, the chi-square and associated p-value and several information criteria. Various time-stamps and the OpenMx version number are also displayed.

```
> summary(factorFit1)
data:
$`One Factor.data`
$`One Factor.data`$cov
           x1        x2        x3        x4        x5
x1 0.1985443 0.1999953 0.2311884 0.2783865 0.3155943
x2 0.1999953 0.2916950 0.2924566 0.3515298 0.4019234
x3 0.2311884 0.2924566 0.3740354 0.4061291 0.4573587
x4 0.2783865 0.3515298 0.4061291 0.5332788 0.5610769
x5 0.3155943 0.4019234 0.4573587 0.5610769 0.6703023


free parameters:
  name matrix row col   Estimate    Std.Error Std.Estimate      Std.SE lbound ubound
1  One Factor.A[1,6]      A  x1   G 0.39715182 0.015549708   0.89130932 0.034897484
2  One Factor.A[2,6]      A  x2   G 0.50366066 0.018232433   0.93255458 0.033758321
3  One Factor.A[3,6]      A  x3   G 0.57724092 0.020448313   0.94384664 0.033435037
4  One Factor.A[4,6]      A  x4   G 0.70277323 0.024011318   0.96236250 0.032880581
5  One Factor.A[5,6]      A  x5   G 0.79624935 0.026669339   0.97255562 0.032574489
6  One Factor.S[1,1]      S  x1  x1 0.04081418 0.002812716   0.20556770 0.014166734
7  One Factor.S[2,2]      S  x2  x2 0.03801997 0.002805791   0.13034196 0.009618951
8  One Factor.S[3,3]      S  x3  x3 0.04082716 0.003152305   0.10915353 0.008427851
9  One Factor.S[4,4]      S  x4  x4 0.03938701 0.003408870   0.07385841 0.006392303
10 One Factor.S[5,5]      S  x5  x5 0.03628708 0.003678556   0.05413557 0.005487924

observed statistics:  15
estimated parameters:  10
degrees of freedom:  5
-2 log likelihood:  -3648.281
saturated -2 log likelihood:  -3655.665
number of observations:  500
chi-square:  7.384002
p:  0.1936117
Information Criteria:
     df Penalty Parameters Penalty Sample-Size Adjusted
AIC:  -2.615998           27.38400                   NA
BIC: -23.689038           69.53008             37.78947
CFI: 0.9993583
TLI: 0.9987166
RMSEA:  0.03088043
timestamp: 2014-04-10 10:23:07
frontend time: 0.02934313 secs
backend time: 0.005492926 secs
independent submodels time: 1.907349e-05 secs
wall clock time: 0.03485513 secs
cpu time: 0.03485513 secs
openmx version number: 999.0.0
```

The table of free parameters requires a little more explanation. First, <NA> is given for the name of elements that were not assigned a label. Second, the columns 'row' and 'col' display the variables at the tail of the paths and the variables at the head of the paths respectively. Third, standard errors are calculated. We will discuss the use of standard errors versus confidence intervals later on.

Model Evaluation

The `mxEval()` function should be your primary tool for observing and manipulating the final values stored within a MxModel object. The simplest form of the `mxEval()` function takes two arguments: an `expression` and a

`model`. The expression can be **any** arbitrary expresssion to be evaluated in R. That expression is evaluated, but the catch is that any named entities or parameter names are replaced with their current values from the model. The model can be either a built or a fitted model.

```
myModel6      <- mxModel('topmodel',
    mxMatrix('Full', 1, 1, values=1, free=TRUE, labels='p1', name='A'),
    mxModel('submodel',
        mxMatrix('Full', 1, 1, values=2, free=FALSE, labels='p2', name='B')
    )
)
myModel6Run   <- mxRun(myModel6)
```

The example above has a model ("submodel") embedded in another model ("topmodel"). Note that the name of the arguments can be omitted if they are used in the default order (`type`, `nrow` and `ncol`).

The `expression` of the `mxEval` statement can include both matrices, algebras as well as matrix element labels, each taking on the value of the model specified in the `model` argument. To reinforce an earlier point, it is not necessary to restrict the expression only to valid MxAlgebra expressions. In the following example, we use the `harmonic.mean()` function from the `psych` package.

```
mxEval(A + submodel.B + p1 + p2, myModel6)        # initial values
mxEval(A + submodel.B + p1 + p2, myModel6Run)     # final values

library(psych)
nVars <- 4
mxEval(nVars * harmonic.mean(c(A, submodel.B)), myModel6)
```

When the name of an entity in a model collides with the name of a built-in or user-defined function in R, the named entity will supercede the function. We strongly advice against naming entities with the same name as the predefined functions or values in R, such as *c*, *T*, and *F* among others.

The `mxEval()` function allows the user to inspect the values of named entities without explicitly poking at the internals of the components of a model. We encourage the use of `mxEval()` to look at the state of a model either before the execution of a model or after model execution.

Indexing Operator

MxModel objects support the `$` operator, also known as the list indexing operator, to access all the components contained within a model. Here is an example collection of models that will help explain the uses of the `$` operator:

```
myModel7 <-
    mxModel('topmodel',
        mxMatrix(type='Full', nrow=1, ncol=1, name='A'),
        mxAlgebra(A, name='B'),
        mxModel('submodel1',
            mxConstraint(topmodel1.A == topmodel1.B, name = 'C'),
            mxModel('undersub1', mxData(diag(3), type='cov', numObs=10)
        )
    ),
    mxModel('submodel2',
        mxFitFunctionAlgebra('topmodel1.A')
    )
)
```

Access Elements

The first useful trick is entering the string `model$` in the R interpreter and then pressing the TAB key. You should see a list of all the named entities contained within the `model` object.

```
> model$
model$A
model$B
model$submodel1
model$submodel2
model$submodel1.C
model$undersub1
model$undersub1.data
model$submodel2.fitfunction
```

The named entities of the model are displayed in one of three modes.

1. All of the submodels contained within the parent model are accessed by using their unique model name (`submodel1`, `submodel2`, and `undersub1`).
2. All of the named entities contained within the parent model are displayed by their names (`A` and `B`).
3. All of the named entities contained by the submodels are displayed in the `modelname.entityname` format (`submodel1.C`, `submodel2.objective`, and `undersub1.data`).

Modify Elements

The list indexing operator can also be used to modify the components of an existing model. There are three modes of using the list indexing operator to perform modifications, and they correspond to the three modes for accessing elements.

In the first mode, a submodel can be replaced using the unique name of the submodel or even eliminated.

```
# replace 'submodel1' with the contents of the mxModel() expression
model$submodel1 <- mxModel(...)
# eliminate 'undersub1' and all children models
model$undersub1 <- NULL
```

In the second mode, the named entities of the parent model are modified using their names. Existing matrices can be eliminated or new matrices can be created.

```
# eliminate matrix 'A'
model$A <- NULL
# create matrix 'D'
model$D <- mxMatrix(...)
```

In the third mode, named entities of a submodel can be modified using the `modelname.entityname` format. Again existing elements can be eliminated or new elements can be created.

```
# eliminate constraint 'C' from submodel1
model$submodel1.C <- NULL
# create algebra 'D' in undersub1
model$undersub1.D <- mxAlgebra(...)
# create 'undersub2' as a child model of submodel1
model$submodel1.undersub2 <- mxModel(...)
```

Keep in mind that when using the list indexing operator to modify a named entity within a model, the name of the created or modified entity is always the name on the left-hand side of the <− operator. This feature can be convenient, as it avoids the need to specify a name of the entity on the right-hand side of the <− operator.

1.1.9 Classes

We have introduced a number of OpenMx functions which correspond to specific classes which are summarized below. The basic unit of abstraction in the OpenMx library is the model. A model serves as a container for a collection of matrices, algebras, constraints, expectation, fit functions, data sources, and nested sub-models. In the parlance of R, a model is a value that belongs to the class MxModel that has been defined by the OpenMx library. The following table indicates what classes are defined by the OpenMx library.

entity	S4 class
model	MxModel
data source	MxData
matrix	MxMatrix
algebra	MxAlgebra
expectation	MxExpectationRAM MxExpectationNormal
fit function	MxFitFunctionML
constraint	MxConstraint

All of the entities listed in the table are identified by the OpenMx library by the name assigned to them. A name is any character string that does not contain the ”.” character. In the parlance of the OpenMx library, a model is a container of named entities. The name of an OpenMx entity bears no relation to the R object that is used to identify the entity. In our example, the object `factorModel` is created with the `mxModel()` function and stores a value that is an “MxModel” object with the name ‘One Factor’.

1.2 Alternative Approaches

In the Beginner’s Guide, we introduced the OpenMx framework for building and fitting models to data. Here we will discuss some basic examples in more detail, providing a parallel treatment of the different formats and types of data and their associated functions to apply them to and the different methods and styles to build and fit models. We will start with a single variable - univariate - example and generate data of different type and in different formats. Three types of data can be used:

“(1)” continuous data, “(2)” ordinal data or “(3)” binary data,

and the two supported data formats are

“(i)” summary statistics, i.e. covariance matrices and possibly means, and “(ii)” raw data format.

We will illustrate all of them, as arguments of functions may differ. There are currently two supported methods for specifying models:

“(a)” path specification and “(b)” matrix specification.

Each example is presented using both approaches, so you can get a sense of their advantages/ disadvantages, and see which best fits your style. In the ‘path specification’ model style you specify a model in terms of paths in a path diagram; the ‘matrix specification’ model style relies on matrices and matrix algebra to produce OpenMx code. Furthermore, we will introduce the different styles available to specify models and data, namely

“(A)” the piecewise style, “(B)” the stepwise style and “(C)” the classic style.

This will be done for a bivariate extension of the first example.

The code used to run the examples in this chapter is available here, and you may wish to access it while working through this manual.

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/AlternativeApproaches.R

1.2.1 Introduction

Our first example fits a model to one variable - and is thus considered a univariate model. The simplest model is one in which we estimate the variance of the variable. Additionally we can estimate both the mean and the variance - a so called saturated model. You might wonder why we use a modeling framework to estimate the mean and variance of a variable which we can do with anything from a calculator to using any basic spreadsheet or statistical program. That is a fair question if you're just interested in the value of the mean and/or variance. However, if you want to test, for example the mean of your variable is equal to a certain fixed value or to the mean of another variable or the same variable in another group, the modeling framework becomes relevant.

1.2.2 Data Handling

You probably have your favorite data set ready to go, but before reading in data from an external file, we simulate a simple dataset directly in R, and use some of R's great capabilities. As this is not an R manual, we just provide the code here with minimal explanation. There are several helpful sites for learning R, for instance http://www.statmethods.net/

```
# Simulate Data
set.seed(100)
x <- rnorm (1000, 0, 1)
univData <- as.matrix(x)
dimnames(univData) <- list(NULL, "X")
summary(univData)
mean(univData)
var(univData)
```

The first line is a comment (starting with a #). We set a seed for the simulation so that we generate the same data each time and get a reproducible answer. We then create a variable *x* for 1000 subjects, with a mean of 0 and a variance of 1, using R's normal distribution function `rnorm`. We read the data in as a matrix into an object *univData* and give the variable a name *"X"* using the `dimnames` command. We can easily produce some descriptive statistics in R using built-in functions `summary`, `mean` and `var`, just to make sure the data look like what we expect. The output generated looks like this:

```
summary(univData)
       X
Min.   :-3.32078
1st Qu.:-0.64970
Median : 0.03690
Mean   : 0.01681
3rd Qu.: 0.70959
Max.   : 3.30415
> mean(univData)
[1] 0.01680509
> var(univData)
          X
X 1.062112
```

For our second example, we will be fitting models to two variables which may be correlated. The data used for the second example were generated using the multivariate normal function (`mvrnorm()` in the R package MASS). The `mvrnorm()` has three arguments: (i) sample size, (ii) vector of means, and (iii) covariance matrix. We are simulating data on two variables named *X* and *Y* for 1000 individuals with means of zero, variances of one and a covariance of 0.5 using the following R code, and saved is as *bivData*. Note that we can now use the R function `colMeans()` to generate the predicted means for the columns of our data frame and the `cov()` to generate the observed covariance matrix.

```
# Simulate Data
require(MASS)
```

```
set.seed(200)
bivData <- mvrnorm (1000, c(0,0), matrix(c(1,.5,.5,1),2,2))
dimnames(bivData) <- list(NULL, c('X','Y'))
summary(bivData)
colMeans(bivData)
cov(bivData)
```

Notice that the simulated data are close to what we expected.

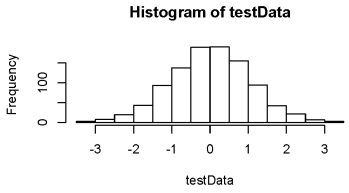
```
> summary(bivData)
       X                    Y
Min.   :-3.296159   Min.   :-2.942561
1st Qu.:-0.596177   1st Qu.:-0.633711
Median :-0.010538   Median :-0.004139
Mean   :-0.004884   Mean   : 0.032116
3rd Qu.: 0.598326   3rd Qu.: 0.739236
Max.   : 4.006771   Max.   : 4.173841
> colMeans(bivData)
         X            Y
-0.004883811  0.032116480
> cov(bivData)
          X         Y
X 0.9945328 0.4818317
Y 0.4818317 1.0102951
```

Data Types

Continuous Data

The data we simulated are continuous in nature and follow a normal distribution. This can easily be verified by R's excellent graphical capabilities. Here we show the R code and a basic histogram of the *univData* we generated.

```
hist(univData)
```



This is the ideal type of data, as many of the models we fit to them assume that the data are normally distributed. However, reality is often different and it might be necessary to apply a transformation to the original data to better approximate a normal distribution. When there are 15 or more possible values for the variable of interest, it is appropriate to treat them as continuous. Note that although the simulated data have many more than 15 different values, values are binned to simplify the graph.

Continuous data can be summarized by their mean and standard deviation. Two or more variables are summarized by a vector of means and a covariance matrix which provides information on the variance of each of the variables as well as the covariances between the variables.

Categorical Data

A lot of variables, however, are not measured on a continuous scale, but using a limited number of categories. If the categories are ordered in a logical way, we refer to them as **ordinal** variables and often assume that the underlying construct follows a normal distribution. This assumption can actually be tested for any ordinal variable with a minimum of three categories, when more than one variable is available or the same variable is measured in related individuals or over time.

Categorical data contain less information than continuous data, and are summarized by thresholds which predict the proportion of individuals in a specific category. As the sum of the proportions of each of the categories adds up to one, there is no information about the variance. The relationship between two or more variables can be summarized in a correlation matrix. Rather than estimating two (or more) thresholds and a correlation(s), one could fix the first threshold to zero and the second to one and estimate the means and covariance matrices instead, which can be interpreted in the same way as for continuous variables. The estimated proportion in each of the categories can then be calculated by rescaling the statistics.

Often, unfortunately, variables are only measured with two categories (Yes/No, affected/unaffected, etc.) and called **binary** variables. The only statistic to be estimated in the univariate case is the threshold and no information is available about the variance. With two or more variables, their relationship is also summarized in a correlation matrix.

The power of your study is directly related to the type of variable analyzed, and is typically higher for continuous variables compared to categorical variables, with ordinal variables providing more power than binary variables. Whenever possible, use continuous variables or at least ordinal variables.

As a lot of real data are only available as categorical variables, we will generate both an ordinal and a binary variable from the simulated continuous variable in *univData*. The code below uses the `cut` and `breaks` commands to cut the continuous variable into 5 ordered categories.

```
univDataOrd <- data.frame(X=cut(univData[,1], breaks=5, ordered_result=T,
                          labels=c(0,1,2,3,4)) )
table(univDataOrd)
```

A summary of the resulting data set looks as follows:

```
> table(univDataOrd)
univDataOrd
  0   1   2   3   4
 28 216 483 244  29
```

A similar approach could be used to create a binary variable. However, here we show an alternative way to generate a binary variable using a specific cutoff using the `ifelse` command. We will assign a value of 1 when the value of our original standardized continuous variable is above 0.5; otherwise a value of 0 will be assigned.

```
univDataBin <- data.frame(X=ifelse(univData[,1] >.5,1,0))
table(univDataBin)
```

The resulting data set table looks as follows:

```
> table(univDataBin)
univDataBin
  0   1
680 320
```

We will go through the same steps to generate ordinal and binary data from the simulated bivariate data. Given we need to repeat the same statement for the two variables, we employ a `for` statement.

```
bivDataOrd <- data.frame(bivData)
for (i in 1:2) { bivDataOrd[,i] <- cut(bivData[,i], breaks=5, ordered_result=T,
                                       labels=c(0,1,2,3,4)) }
```

```
   table(bivDataOrd[,1],bivDataOrd[,2])
bivDataBin <- data.frame(bivData)
for (i in 1:2) { bivDataBin[,i] <- ifelse(bivData[,i] >.5,1,0) }
   table(bivDataBin[,1],bivDataBin[,2])
```

Data Formats

Raw Data

To make these data available for statistical modeling in OpenMx, we need to create an "MxData" object which is accomplished with the `mxData` function. Remember to load the OpenMx package first.

```
require(OpenMx)
obsRawData <- mxData( observed=univData, type="raw" )
selVars <- "X"
```

First, we read the data matrix in with the `observed` argument. Then, we tell OpenMx what format or type the data is in, in this case we're reading in the raw data. We save this MxData object as *obsRawData*. As later on, we need to be able to map our data onto the model, we typically create a vector with the variable labels of the variable(s) we are analyzing. To make our scripts more readable, we use consistent names for objects - something you can decide to copy or change according to your preferences - and we use *selVars* for the variables we select for analysis. In this example, it is a single variable *X*.

```
> obsRawData
MxData 'data'
type : 'raw'
numObs : '1000'
Data Frame or Matrix :
                      X
   [1,] -5.021924e-01
   [2,]  1.315312e-01
   ....
   [1000,] -2.141428e+00
   Means : NA
   Acov : NA
   Thresholds : NA
```

A look at this newly created object shows that it was given the `name` *data*, which is done by default. It has the `type` that we specified, and `numObs` are automatically counted for us. The actual data for the variable *X* are then listed; we only show the first two values.

In a similar manner we create a MxData object for the second example. We read in the `observed` *bivData*, and indicate the `type` as raw. We refer to this object as *obsBivData*.

```
obsBivData <- mxData( observed=bivData, type="raw" )
```

If we want to fit models to categorical data, we need to read in the ordinal or binary data. However, when your data are ordinal or binary, OpenMx expects them to be 'ordered factors'. To ensure that your data have the appropriate format, it is recommended/required to apply the `mxFactor` command to the categorical variables, where the `x` argument reads in a vector of data or a data.frame, and `levels` expects a vector of possible values for those data. We save the resulting objects as *univDataOrdF* and *univDataBinF*, or *bivDataOrdF* and *bivDataBinF* for the corresponding data in the second example.

```
univDataOrdF <- mxFactor( x=univDataOrd, levels=c(0:4) )
univDataBinF <- mxFactor( x=univDataBin, levels=c(0,1) )
bivDataOrdF  <- mxFactor( x=bivDataOrd, levels=c(0:4) )
bivDataBinF  <- mxFactor( x=bivDataBin, levels=c(0,1) )
```

Next, we generate the corresponding MxData objects.

```
obsRawDataOrd <- mxData( observed=univDataOrdF, type="raw" )
obsRawDataBin <- mxData( observed=univDataBinF, type="raw" )
obsBivDataOrd <- mxData( observed=bivDataOrdF, type="raw" )
obsBivDataBin <- mxData( observed=bivDataBinF, type="raw" )
```

Summary Stats

Covariances While analyzing raw data is the standard in most statistical modeling these days, this was not the case in a previous generation of computers, which could only deal with summary statistics. As fitting models to summary statistics still is much faster then using raw data (unless your data set is small), it is sometimes useful for didactic purposes. Furthermore, sometimes one has access only to the summary statistics. In the case where the dataset is complete, in other words there are no missing data, there is no advantage to using raw data. For our example, we can easily create a covariance matrix based on our data set by using R's `var()` function, in the case of analyzing a single variable, or `cov()` function, when analyzing more than one variable. This can be done prior to or directly when creating the MxData object. Its first argument, `observed`, reads in the data from an R matrix or data.frame, with the `type` given in the second argument, followed by the `numObs` argument which is necessary when reading in summary statistics.

```
univDataCov <- var(univData)
obsCovData  <- mxData( observed=univDataCov, type="cov", numObs=1000 )
```

or

```
obsCovData  <- mxData( observed=var(univData), type="cov", numObs=1000 )
```

Given our first example has only one variable, we use the `var()` function (as there is no covariance for a single variable). When summary statistics are used as input, the number of observations (`numObs`) needs to be supplied. The resulting MxData object looks as follows:

```
> obsCovData
MxData 'data'
type : 'cov'
numObs : '1000'
Data Frame or Matrix :
          X
X 1.062112
Means : NA
Acov : NA
Thresholds : NA
```

The differences with the previous data objects are that the type is now 'cov' and the actual data frame is now a single value, the variance of the 1000 data points.

Covariances + Means In addition to the observed covariance matrix, a fourth argument `means` can be added for the vector of observed means from the data, calculated using the R `colMeans` command.

```
obsCovMeanData <- mxData( observed=var(univData),  type="cov", numObs=1000,
                          means=colMeans(univData) )
```

You can verify that the new *obsCovMeanData* object now has a value for the observed means as well.

For the second, bivariate example the only change we'd have to make - besides reading in the *bivData* - is the use of `cov` instead of `var` to generate the object for the observed covariance matrix.

Correlations To analyze categorical data, we can also fit the models to summary statistics, in this case, correlation matrices, as indicated by using the `cor()` R command to generate them and by the `type`=cor, which also requires the `numObs` argument to indicate how many observations (data records) are in the dataset.

```
obsOrdData <- mxData( observed=cor(univDataOrdF), type="cor", numObs=1000 )
```

We will start by fitting a simple univariate model to the continuous data and then show which changes have to be made when dealing with ordinal or binary variables. For the continuous data example, we will start with fitting the model to the summary statistics prior to fitting to raw data and show their equivalence (in the absence of missing data).

1.2.3 Model Handling

Path Method

Summary Stats

If we have data on a single variable *X* summarized in its variance, the basic univariate model will simply estimate the variance of the variable *X*. We call this model saturated because there is a free parameter corresponding to each and every observed statistic. Here we have covariance matrix input only, so we can estimate one variance. This model can be represented by the following path diagram:

Model Building When using the path specification, it is easiest to work from the path diagram. Assuming you are familiar with path analysis (*for those who are not, there are several excellent introductions, see [LI1986]*), we have a box for the observed/manifest variable *X*, and one double headed arrow, labeled σ_x^2. To indicate which variable we are analyzing, we use the `manifestVars` argument, which takes a vector of labels. In this example, we are selecting one variable, which we pre-specified in the *selVars* object.

```
selVars    <- c("X")
manifestVars=selVars
```

We have already built the MxData object above, so here we will build the model by specifying the relevant paths. Our first model only has one path which has two arrows and goes from the variable *X* to the variable *X*. That path represents the variance of *X* which we aim to estimate. Let's see how this translates into the `mxPath` object.

The `mxPath` command indicates where the path originates (`from`) and where it ends (`to`). If the `to` argument is omitted, the path ends at the same variable where it started. The `arrows` argument distinguishes one-headed arrows (if `arrows`=1) from two-headed arrows (if `arrows`=2). The `free` command is used to specify which elements are free or fixed with a `TRUE` or `FALSE` option. If the `mxPath` command creates more than one path, a single "T" implies that all paths created here are free. If some of the paths are free and others fixed, a list is expected. The same applies for the `values` command which is used to assign starting values or fixed final values, depending on the corresponding 'free' status. Optionally, lower and upper bounds can be specified (using `lbound` and `ubound`, again generally for all the paths or specifically for each path). Labels can also be assigned using the `labels` command which expects as many labels (in quotes) as there are elements. Thus for our example, we specify only a `from` argument, as the double-headed arrow (`arrows`=2) goes back to *X*. This path is estimated (`free`=TRUE), and given a start value of

1 (values=1) and has to be positive (lbound=.01). Finally we assign it a label (labels="vX"). The generated MxPath object is called *expVariance*.

```
expVariance <- mxPath(
    from=c("X"), arrows=2,
    free=TRUE,
    values=1,
    lbound=.01,
    labels="vX"
    )
```

Note that all arguments could be listed on one (or two) lines; in either case they are separated by comma's:

```
expVariance <- mxPath( from=c("X"), arrows=2,
                       free=TRUE, values=1, lbound=.01, labels="vX" )
```

The resulting MxPath object looks as follows:

```
> expVariance
mxPath
$from:  'X'
$to:  'X'
$arrows:  2
$values:  1
$free:  TRUE
$labels:  vX
$lbound:  0.01
$ubound:  NA
$connect:  single
```

To evaluate the model that we have built, we need an expectation and a fit function that obtain the best solution for the model given the data. When using the path specification, both are automatically generated by invoking the type="RAM" argument in the model. The 'RAM' objective function has a predefined structure.

```
type="RAM"
```

Internally, OpenMx translates the paths into RAM notation in the form of the matrices **A**, **S**, and **F** [see RAM1990]. Before we can 'run' the model through the optimizer, we need to put all the arguments into an MxModel using the mxModel command. Its first argument is a name, and therefore is in quotes. We then add all the arguments we have built so far, including the list of variables to be analyzed in manifestVars, the MxData object, and the predicted model specified using paths.

```
univSatModel1 <- mxModel("univSat1", manifestVars=selVars, obsCovData,
                         expVariance, type="RAM" )
```

We can inspect the MxModel object generated by this statement.

```
> univSatModel1
MxModel 'univSat1'
type : RAM
$matrices : 'A', 'S', and 'F'
$algebras :
$constraints :
$intervals :
$latentVars : none
$manifestVars : 'X'
$data : 1 x 1
$data means : NA
$data type: 'cov'
$submodels :
```

```
$expectation : MxExpectationRAM
$fitfunction : MxFitFunctionML
$compute : NULL
$independent : FALSE
$options :
$output : FALSE
```

Note that only the relevant arguments have been updated, and that the path information has been stored in the **A**, **S**, and **F** matrices. The free parameter for the variance "vX" ends up in the **S** matrix which holds the symmetric (double-headed) paths. Here we print the details for this **S** matrix:

```
> univSatModel1$matrices$S
SymmMatrix 'S'

$labels
  X
X "vX"

$values
  X
X 1

$free
     X
X TRUE

$lbound
     X
X 0.01

$ubound: No upper bounds assigned.
```

Model Fitting So far, we have specified the model, but nothing has been evaluated. We have 'saved' the specification in the object *univSatModel1*. This object is evaluated when we invoke the `mxRun` command with the MxModel object as its argument.

```
univSatFit1 <- mxRun(univSatModel1)
```

You can verify that the arguments of the *univSatModel1* and *univSatFit1* look mostly identical. What we expect to be updated with the estimated value of variance is the element of the **S** matrix, which we can output as follows:

```
> univSatFit1$matrices$S$values
         X
X 1.062112
```

An alternative form of extracting values from a matrix is:

```
> univSatFit1[['S']]$values
         X
X 1.062112
```

There are actually a variety of ways to generate output. We will promote the use of the `mxEval` command, which takes two arguments: an `expression` and a `model` object. The `expression` can be a matrix or algebra defined in the model, new calculations using any of these matrices/algebras of the model, the objective function, etc. Here we use `mxEval` to simply list the values of the **S** matrix, which formats the output slightly differently as a typical R matrix object, and call it *EC1*, short for the expected covariance:

```
EC1 <- mxEval(S, univSatFit1)
>         EC1
              X
    X 1.062112
```

We can then use any regular R function in the `mxEval` command to generate derived fit statistics, some of which are built in as standard. When fitting to covariance matrices, the saturated likelihood can be easily obtained and subtracted from the likelihood of the data to obtain a Chi-square goodness-of-fit. The saturated likelihood, here named 'SL1' is obtained from the `$output$Saturated` argument of the fitted object *univSatFit1* which contains a range of statistics. We get the likelihood of the data, here referred to as *LL1*, from the `$output$fit` argument of the fitted object *univSatFit1*.

```
SL1  <- univSatFit1$output$Saturated
LL1  <- univSatFit1$output$fit
Chi1 <- LL1-SL1
```

The output of these objects like as follows

```
> SL1
[1] 1059.199
> LL1
[1] 1059.199
> Chi1
[1] 0
```

An alternative to requesting specific output is to generate the default summary of the model, which can be done with the `summary` function, and can also be saved in another R object, i.e. *univSatSumm1*.

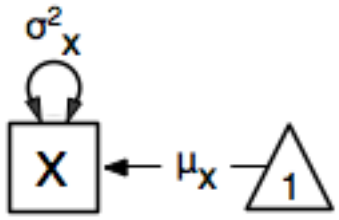
```
summary(univSatFit1)
univSatSumm1 <- summary(univSatFit1)
```

This output includes a summary of the data (if available), a list of all the free parameters with their estimates (if the model contains free parameters), their confidence intervals (if requested), a list of goodness-of-fit statistics, and a list of job statistics (timestamps and OpenMx version).

```
> univSatSumm1
data:
$univSat1.data
$univSat1.data$cov
          X
X 1.062112

free parameters:
  name matrix row col Estimate  Std.Error Std.Estimate     Std.SE lbound ubound
1   vX      S   X   X 1.062112 0.04752282            1 0.04474372   0.01

observed statistics:  1
estimated parameters:  1
degrees of freedom:  0
-2 log likelihood:  1059.199
saturated -2 log likelihood:  1059.199
number of observations:  1000
chi-square:  0
p:  1
Information Criteria:
    df Penalty Parameters Penalty Sample-Size Adjusted
AIC          0           2.000000                   NA
BIC          0           6.907755             3.731699
CFI: NaN
```

```
TLI: NaN
RMSEA:  NA
timestamp: 2014-04-02 18:41:35
frontend time: 0.09399414 secs
backend time: 0.007524967 secs
independent submodels time: 5.602837e-05 secs
wall clock time: 0.1015751 secs
cpu time: 0.1015751 secs
openmx version number: 999.0.0-3160
```

In addition to providing a covariance matrix as input data, we could add a means vector. As this requires a few minor changes, let's highlight those. The path diagram for this model, now including means (path from triangle of value 1) is as follows:



We have to specify one additional `mxPath` command for the means. In the path diagram, the means are specified by a triangle which has a fixed value of one, reflected in the `from="one"` argument, with the `to` argument referring to the variable whose mean is estimated. Note that paths for means are always single headed. We will save this path as the R object *expMean*.

```
expMean <- mxPath(from="one", to="X", arrows=1, free=TRUE, values=0, labels="mX")
```

This new path adds one additional parameter, called 'mX'.

```
>  expMean
mxPath
$from:  'one'
$to:  'X'
$arrows:  1
$values:  0
$free:  TRUE
$labels:  mX
$lbound:  NA
$ubound:  NA
$connect:  single
```

The other required change is in the `mxData` command, which now takes a fourth argument `means` for the vector of observed means from the data, calculated using the R `colMeans` command.

```
obsCovMeanData <- mxData( observed=var(univData), type="cov", numObs=1000,
                          means=colMeans(univData) )
```

As this new object will simply be added to the previous model, we can build onto our existing model. Therefore, instead of using the first argument for the name, we use it in its other capacity, namely as the name of a previously defined MxModel object that is being modified. In this case, we start with the previous model *univSatModel1*, which becomes the first argument of our new model *univSatModel1M*. To change the name of the object, we add a `name` argument. Note that the default order of arguments can be changed by adding the argument's syntax name. We then add the new argument for the expected means, as well as the modified MxData object.

```
univSatModel1M <- mxModel(univSatModel1, name="univSat1M", expMean, obsCovMeanData )
```

Note the following changes in the modified MxModel below. First, the name is changed to 'univSat1M'. Second, an additional matrix **M** was generated for the expected means vector. Third, observed means were added, here referred to as '$data means'.

```
>   univSatModel1M
MxModel 'univSat1M'
type : RAM
$matrices : 'A', 'S', 'F', and 'M'
$algebras :
$constraints :
$intervals :
$latentVars : none
$manifestVars : 'X'
$data : 1 x 1
$data means : 1 x 1
$data type: 'cov'
$submodels :
$expectation : MxExpectationRAM
$fitfunction : MxFitFunctionML
$compute : NULL
$independent : FALSE
$options :
$output : FALSE
```

When a mean vector is supplied and a parameter added for the estimated mean, the RAM matrices **A**, **S** and **F** are augmented with an **M** matrix which can be extracted from the output in a similar way as the expected variance before, and is called *EM1*, short for expected mean.

```
univSatFit1M  <- mxRun(univSatModel1M)
EM1M          <- mxEval(M, univSatFit1M)
univSatSumm1M <- summary(univSatFit1M)
```

The new summary object *univSatSumm1M* is different from the previous one in the following ways: the observed data means were added, an extra free parameter is listed and estimated, thus the fit statistics are updated. Notice, however, that the likelihood of both models is the same. (We have cut part of the summary that is not relevant here.)

```
> univSatSumm1M
data:
$univSat1M.data
$univSat1M.data$cov
          X
X 1.062112

$univSat1M.data$means
               X
[1,] 0.01680509


free parameters:
  name matrix row col   Estimate  Std.Error Std.Estimate     Std.SE lbound ubound
1   vX      S   X   X 1.06211141 0.04752281            1 0.04474372   0.01
2   mX      M   1   X 0.01680503 0.03259006           NA         NA

observed statistics:  2
estimated parameters:  2
degrees of freedom:  0
-2 log likelihood:  1059.199
```

```
saturated -2 log likelihood:  1059.199
number of observations:  1000
chi-square:  8.867573e-12
p:  0
Information Criteria:
      df Penalty Parameters Penalty Sample-Size Adjusted
AIC 8.867573e-12            4.00000                   NA
BIC 8.867573e-12           13.81551             7.463399
```

Raw Data

Instead of fitting models to summary statistics, it is now popular to fit models directly to the raw data and using full information maximum likelihood (FIML). Doing so requires specifying not only a model for the covariances, but also one for the means, just as in the case of fitting to covariance matrices and mean vectors described above.

The only change required is in the MxData object, *obsRawData* defined above, which reads the raw data in directly from an R matrix or a data.frame into the `observed` first argument, and has `type="raw"` as its second argument. A nice feature of OpenMx is that existing models can be easily modified. Here we will start from the saturated model estimating covariances and means from summary statistics, namely *univSatModel1M*, as both expected means and covariances have to be modeled when fitting to raw data.

```
univSatModel2 <- mxModel(univSatModel1M, obsRawData )
```

The resulting model can be run as usual using `mxRun`:

```
univSatFit2  <- mxRun(univSatModel2)
univSatSumm2 <- summary(univSatFit2)
EM2          <- mxEval(M, univSatFit2)
EC2          <- mxEval(S, univSatFit2)
LL2          <- univSatFit2$output$fit
```

Note that the estimates for the expected means, as well as the expected covariance matrix are exactly the same as before, as we have no missing data.

```
>         EM2
               X
[1,] 0.01680499
>         EC2
          X
X 1.061049
>         LL2
[1] 2897.135
```

The estimates for the predicted mean and covariance matrix are exactly the same as those obtained when fitting to summary statistics. The likelihood, however, is different.??

```
> univSatSumm2
data:
$univSat1M.data
        X
Min.   :-3.32078
1st Qu.:-0.64970
Median : 0.03690
Mean   : 0.01681
3rd Qu.: 0.70959
Max.   : 3.30415
```

```
free parameters:
 name matrix row col   Estimate  Std.Error Std.Estimate     Std.SE lbound ubound
1   vX      S   X   X 1.06104923 0.04745170            1 0.04472149   0.01
2   mX      M   1   X 0.01680499 0.03257418           NA         NA

observed statistics:  1000
estimated parameters:  2
degrees of freedom:  998
-2 log likelihood:  2897.135
saturated -2 log likelihood:  NA
number of observations:  1000
chi-square:  NA
p:  NA
Information Criteria:
   df Penalty Parameters Penalty Sample-Size Adjusted
AIC   901.1355           2901.135                  NA
BIC -3996.8043           2910.951            2904.599
```

Matrix Method

The next example replicates these models using matrix-style coding. In addition to the `mxData` and `mxModel` commands which were introduced before, the code to specify the model includes three new commands, (i) `mxMatrix`, and (ii) `mxExpectationNormal` and `mxFitFunctionML()`.

Summary Stats

Covariances Starting with the model fitted to the summary covariance matrix, the `mxData` is identical to that used in path style models, as is the case for all the corresponding models specified using paths or matrices.

To specify the model, we now create a matrix for the expected covariance matrix using the `mxMatrix` command. The first argument is its `type`, symmetric for a covariance matrix. The second and third arguments are the number of rows (`nrow`) and columns (`ncol`) – one each for a univariate model. The `free` and `values` parameters work as in the path specification. If only one element is given, it is applied to all elements of the matrix. Alternatively, each element can be assigned its free/fixed status and starting value with a list command. Note that in the current example, the matrix is a simple **1x1** matrix, but that will change rapidly in the later examples.

```
expCovMat <- mxMatrix( type="Symm", nrow=1, ncol=1,
                       free=TRUE, values=1, name="expCov" )
```

The resulting MxMatrix object *expCovMat* looks as follows. Note that the starting value for the free parameter is 1 and that optionally labels can be assigned for the rows and columns of the matrix and lower and upper bounds can be assigned to limit the parameter space for the estimation:

```
> expCovMat
SymmMatrix 'expCov'

$labels: No labels assigned.

$values
     [,1]
[1,]    1

$free
     [,1]
[1,] TRUE
```

```
$lbound: No lower bounds assigned.

$ubound: No upper bounds assigned.
```

To link the model for the covariance matrix to the data, an `mxExpectation` needs to be specified which will be evaluated with an `mxFitFunctionML`. The `mxExpectationNormal` command takes two arguments, `covariance` to hold the expected covariance matrix (which we named "expCov" above using the `mxMatrix` command), and `dimnames` which allow the mapping of the observed data to the expected covariance matrix, i.e. the model. `mxFitFunctionML()` will invoke the maximum likelihood ('ML'), to obtain the best estimates for the free parameters.

```
expectCov     <- mxExpectationNormal( covariance="expCov", dimnames=selVars )
funML         <- mxFitFunctionML()
```

The internal name of an MxExpectationNormal object is by default *expectation* and that for an MxFitFunctionML object is by default *fitfunction*. We can thus inspect these two objects by using the names of the resulting objects, here *expCovFun* and *ML* as shown below. The result of applying the fit function is not yet computed and thus reported as *<0 x 0 matrix>*; its arguments will change after running the model successfully.

```
> expectCov
MxExpectationNormal 'expectation'
$covariance : 'expCov'
$means : NA
$dims : 'X'
$thresholds : NA
$threshnames : 'X'

> funML
MxFitFunctionML 'fitfunction'
$vector : FALSE
<0 x 0 matrix>
```

We can then simply combine the appropriate elements into a new model and fit it in the usual way to the data. Please note that within the `mxExpectationNormal` function, we refer to the expected covariance matrix by its name within the `mxMatrix` function that created the matrix, namely *expCov*. However when we combine the arguments into the `mxModel` function, we use the name of the MxMatrix and MxMLObjective objects, respectively *expCovMat*, *expCovFun* and *ML*, as shown below.

```
univSatModel3 <- mxModel("univSat3", obsCovData, expCovMat, expectCov, funML)
univSatFit3   <- mxRun(univSatModel3)
univSatSumm3  <- summary(univSatFit3)
```

Note that the estimates for the free parameters and the goodness-of-fit statistics are exactly the same for the matrix method as they were for the path method.

```
> univSatSumm3
data:
$univSat3.data
$univSat3.data$cov
          X
X 1.062112

free parameters:
  name matrix row col Estimate  Std.Error lbound ubound
1 <NA> expCov   X   X 1.062112 0.04752287

observed statistics:  1
estimated parameters:  1
```

```
degrees of freedom:  0
-2 log likelihood:  1059.199
saturated -2 log likelihood:  1059.199
number of observations:  1000
chi-square:  0
p:  1
Information Criteria:
    df Penalty Parameters Penalty Sample-Size Adjusted
AIC          0           2.000000                   NA
BIC          0           6.907755             3.731699
```

We can also obtain the values of the likelihood by accessing the fitted object with the default name for the fit function, here *univSatFit4$fitfunction*. Note the the expectation part of the fitted object has not changed.

```
> univSatFit3$expectation
MxExpectationNormal 'expectation'
$covariance : 'expCov'
$means : NA
$dims : 'X'
$thresholds : NA
$threshnames : 'X'

> univSatFit3$fitfunction
MxFitFunctionML 'fitfunction'
$vector : FALSE
        [,1]
[1,] 1059.199
attr(,"expCov")
         [,1]
[1,] 1.062112
attr(,"expMean")
<0 x 0 matrix>
attr(,"gradients")
<0 x 0 matrix>
attr(,"SaturatedLikelihood")
[1] 1059.199
attr(,"IndependenceLikelihood")
[1] 1059.199
```

Covariances + Means A means vector can also be added to the observed data as the fourth argument of the `mxData` command. When means are requested to be modeled, a second `mxMatrix` command is required to specify the vector of expected means. In this case a matrix of `type` ="Full", with one row and one column, is assigned `free` =TRUE with start value zero, and the name *expMean*. The object is saved as *expMeanMat*.

```
expMeanMat <- mxMatrix( type="Full", nrow=1, ncol=1,
                        free=TRUE, values=0, name="expMean" )
```

When we inspect this MxMatrix object, note that it looks rather similar to the *expCovMat* object, except for the name and type and start value. Its estimate depends entirely on which argument of the expectation function it is supposed to represent. As soon as we move to an example with more than one variable, the difference becomes more obvious as the expected means will be a vector while the expected covariance matrix will always be a symmetric matrix.

```
> exMeanMat
SymmMatrix 'expMean'

$labels: No labels assigned.
```

```
$values
     [,1]
[1,]    0

$free
     [,1]
[1,] TRUE

$lbound: No lower bounds assigned.

$ubound: No upper bounds assigned.
```

The second change is adding an additional argument `means` to the `mxExpectationNormal` function for the expected mean, here *expMean*.

```
expextCovMean <- mxExpectationNormal( covariance="expCov", means="expMean",
                                      dimnames=selVars )
```

We now create a new model based on the old one, give it a new name, read in the MxData object with covariance and mean, add the MxMatrix object for the means and change the expectation function to the one created above.

```
univSatModel3M <- mxModel(univSatModel3, name="univSat3M", obsCovMeanData,
                          expMeanMat, expextCovMean, funML )
univSatFit3M   <- mxRun(univSatModel3M)
univSatSumm3M  <- summary(univSatFit3M)
```

You can verify that the only changes to the output are the addition of the means to the data and estimates, resulting in two observed statistics and two estimated parameters rather than one. As a result the values AIC and BIC criteria have changed although the value for the likelihood is exactly the same as before.

Raw Data

Finally, if we want to use the matrix specification with raw data, no changes are needed to the matrices for the means and covariances, or to the expectation which combines the two. Instead of summary statistics, we now fit the model to the raw data, saved in the MxData object *obsRawData*. The fit function is still the same `mxFitFunctionML()` but now uses FIML (Full Information Maximum Likelihood), appropriate for raw data to evaluate the likelihood of the data .

The MxModel object for the saturated model applied to raw data has a name *univSat4*, a MxData object *obsRawData*, a MxMatrix object for the expected covariance matrix *expCovMat*, a MxMatrix object for the expected means vector *expMeanMat*, a mxExpectationNormal object *expCovMeanFun*, and a mxFitFunction object *ML*.

```
univSatModel4 <- mxModel("univSat4", obsRawData,
                          expCovMat, expMeanMat, expectCovMean, funML )
univSatFit4   <- mxRun(univSatModel4)
univSatSumm4  <- summary(univSatFit4)
```

The output looks like this:

```
> univSatSumm4
data:
$univSat4.data
       X
Min.   :-3.32078
1st Qu.:-0.64970
Median : 0.03690
Mean   : 0.01681
3rd Qu.: 0.70959
```

```
Max.   : 3.30415

free parameters:
  name  matrix row col   Estimate  Std.Error lbound ubound
1 <NA>  expCov   X   X 1.06104925 0.04745032
2 <NA> expMean   1   X 0.01680499 0.03257294

observed statistics:  1000
estimated parameters:  2
degrees of freedom:  998
-2 log likelihood:  2897.135
saturated -2 log likelihood:  NA
number of observations:  1000
chi-square:  NA
p:  NA
Information Criteria:
    df Penalty Parameters Penalty Sample-Size Adjusted
AIC   901.1355           2901.135                  NA
BIC -3996.8043           2910.951             2904.599
```

Note that the output generated for the paths and matrices specification are again completely equivalent, regardless of whether the model was fitted to summary statistics or raw data. In each of the four versions of the model fitted to the same data, the data objects were generated from the continuous data. Similar models can be fit to categorical data, with one or more thresholds delineating the proportion of individual in each of the two or more categories, based on the assumption of an underlying (multi)normal probability density function.

Threshold Model

Binary Data We will show below - only for the version using the matrix method to build a model to be fitted to the raw data - which changes are required when the input data is categorical. We'll start with a binary example, followed by an ordinal one.

First, we read in the binary data, *obsRawDataBin* created earlier. Then we turn the symmetric predicted covariance matrix into a standardized matrix with the variance of categorical variables (on the diagonal) fixed to one. To estimate the thresholds, we need to fix the mean to zero, by changing the `type` argument to 'Zero'. The one new object that is required is a matrix for the thresholds which will be estimated. For binary data, the threshold matrix is similar to the means matrix before.

```
expCovMatBin  <- mxMatrix( type="Stand", nrow=1, ncol=1,
                           free=TRUE, values=.5, name="expCov" )
expMeanMatBin <- mxMatrix( type="Zero", nrow=1, ncol=1, name="expMean" )
expThreMatBin <- mxMatrix( type="Full", nrow=1, ncol=1,
                           free=TRUE, values=0, name="expThre" )
```

Let's inspect the latter matrix.

```
> expThreMatBin
FullMatrix 'expThre'

$labels: No labels assigned.

$values
     [,1]
[1,]    0

$free
     [,1]
```

```
[1,] TRUE

$lbound: No lower bounds assigned.

$ubound: No upper bounds assigned.
```

The final change is adding an additional `threshold` argument to the `mxExpectationNormal` function for the expected threshold, here "expThre".

```
expectBin <- mxExpectationNormal( covariance="expCov", means="expMean",
                                  threshold="expThre", dimnames=selVars )
```

We then include all these objects into a model *univSat5* and fit it to the data.

```
univSatModel5 <- mxModel("univSat5", obsRawDataBin,
                         expCovMatBin, expMeanMatBin, expThreMatBin, expectBin, funML )
univSatFit5   <- mxRun(univSatModel5)
univSatSumm5  <- summary(univSatFit5)
```

The summary of the univariate model fitted to binary data includes a summary of the data. Given binary data have no variance, it is fixed to one while the threshold is estimated.

```
> univSatSumm5
data:
$univSat5.data
 X
 0:680
 1:320

free parameters:
  name  matrix row col  Estimate  Std.Error lbound ubound
1 <NA> expThre   1   X 0.4676989 0.04124951

observed statistics:  1000
estimated parameters:  1
degrees of freedom:  999
-2 log likelihood:  1253.739
saturated -2 log likelihood:  NA
number of observations:  1000
chi-square:  NA
p:  NA
Information Criteria:
    df Penalty Parameters Penalty Sample-Size Adjusted
AIC  -744.2611           1255.739                   NA
BIC -5647.1086           1260.647             1257.471
CFI: NA
TLI: NA
RMSEA:  NA
timestamp: 2012-02-24 00:32:39
frontend time: 0.1296248 secs
backend time: 0.007578135 secs
independent submodels time: 5.102158e-05 secs
wall clock time: 0.137254 secs
cpu time: 0.137254 secs
openmx version number: 999.0.0-1661
```

Ordinal Data Next, we will show how to adapt the model to analyze an ordinal variable. As the number of thresholds depends on the variable, we specify it first, by creating a number of thresholds *nth* object. The matrices for the expected

covariance matrices and expected means are the same as in the binary case. The matrix for the thresholds, however, now has as many rows as there are thresholds. Furthermore, start values should be increasing. Here, we estimate the thresholds directly though.

```
nth <- 4
expCovMatOrd  <- mxMatrix( type="Stand", nrow=1, ncol=1,
                           free=TRUE, values=.5, name="expCov" )
expMeanMatOrd <- mxMatrix( type="Zero", nrow=1, ncol=1, name="expMean" )
expThreMatOrd <- mxMatrix( type="Full", nrow=nth, ncol=1,
                           free=TRUE, values=c(-1.5,-.5,.5,1.5), name="expThre" )
```

Here we print the matrix of thresholds:

```
> expThreMatOrd
FullMatrix 'expThre'

$labels: No labels assigned.

$values
     [,1]
[1,] -1.5
[2,] -0.5
[3,]  0.5
[4,]  1.5

$free
     [,1]
[1,] TRUE
[2,] TRUE
[3,] TRUE
[4,] TRUE

$lbound: No lower bounds assigned.

$ubound: No upper bounds assigned.
```

The remainder of the model statements is almost identical to those of the binary model, except for replacing 'Bin' with 'Ord'.

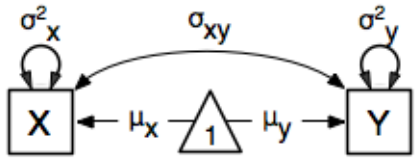
```
expFunOrd     <- mxExpectationNormal( covariance="expCov", means="expMean",
                                      threshold="expThre", dimnames=selVars )
univSatModel6 <- mxModel("univSat6", obsRawDataOrd,
                         expCovMatOrd, expMeanMatOrd, expThreMatOrd, expectOrd, funML )
univSatFit6   <- mxRun(univSatModel6)
univSatSumm6  <- summary(univSatFit6)
```

Thresholds An alternative approach to ensure that the thresholds are increasing can be enforced through multiplying the threshold matrix with a lower triangular matrix of 'Ones' and bounding all threshold increments except the first to be positive. The first threshold will be estimated as before. The remaining thresholds are estimated as increments from the previous thresholds. To generalize this, we specify a start value for the lower threshold ('svLTh') and for the increments ('svITh'), and then create a vector of start values to match the number of thresholds ('svTh'). Similarly, a vector of lower bounds is defined with all thresholds, except the first bounded to be positive ('lbTh'). These start values and lower bounds are read in to a MxMatrix object, of size *nth x 1*, similar to the threshold matrix in the previous example. Then, we create a lower triangular matrix of ones which will be pre-multiplied with the threshold matrix to generate the expected threshold matrix *expThreMatOrd*. The rest of the model is not changed, except that all the intermediate matrices, named *threG* and *inc* also have to be included in the MxModel object *univSatModel6I*.

```
svLTh      <- -1.5                                  # start value for first threshold
svITh      <- 1                                     # start value for increments
svTh       <- (c(svLTh,(rep(svITh,nth-1))))         # start value for thresholds
lbTh       <- c(-3,(rep(0.001,nth-1)))              # lower bounds for thresholds

threG          <- mxMatrix( type="Full", nrow=nth, ncol=1,
                            free=TRUE, values=svTh, lbound=lbTh, name="Thre" )
inc            <- mxMatrix( type="Lower", nrow=nth, ncol=nth,
                            free=FALSE, values=1, name="Inc" )
expThreMatOrd  <- mxAlgebra( expression= Inc %*% Thre, name="expThre" )
expectOrd      <- mxExpectationNormal( covariance="expCov", means="expMean",
                                       threshold="expThre", dimnames=selVars )
univSatModel6I <- mxModel("univSat6", obsRawDataOrd,
                          expCovMatOrd, expMeanMatOrd,
                          Inc, Thre, expThreMatOrd, expectOrd, funML )
univSatFit6I   <- mxRun(univSatModel6I, unsafe=T)
univSatSumm6I  <- summary(univSatFit6I)
```

1.2.4 Approaches

Rarely will we analyze a single variable. As soon as a second variable is added, not only can we estimate both means and variances, but also a covariance between the two variables, as shown in the following path diagram:



The path diagram for our bivariate example includes two boxes for the observed variables **X** and **Y**, each with a two-headed arrow for the variance of each of the variables. We also estimate a covariance between the two variables with the two-headed arrow connecting the two boxes. The optional means are represented as single-headed arrows from a triangle to the two boxes.

As raw data are now standard for data analysis, we will focus this example on fitting directly to the raw data. We will present the example in both the path and the matrix specification, and furthermore show not only the piecewise style but also the stepwise and the classic style of writing OpenMx scripts.

Piecewise Style

Here we will illustrate the various approaches with the bivariate example. For the piecewise approach, we'll show both the path specification and the matrix specification. The other two approaches, stepwise and classic, will just be shown for the matrix example as specifying models using matrix algebra allows for greater flexibility and variety of models to be built.

Path Method

In the path specification, we will use three `mxPath` commands to specify (i) the variance paths, (ii) the covariance path, and (iii) the mean paths. We first specify the number of variables *nv* and which variables are selected for analysis *selVars*.

```
nv      <- 2
selVars <- c('X','Y')
```

We start with the two-headed paths for the variances and covariances. The first one specifies two-headed arrows from *X* and *Y* to themselves - the `to` argument is omitted - to represent the variances. This command now generates two free parameters, each with start value of 1 and lower bound of .01, but with a different label indicating that these are separate free parameters. Note that we could test whether the variances are equal by specifying a model with the same label for the two variances and comparing it with the current model. The second `mxPath` command specifies a two-headed arrow from *X* to *Y* for the covariance, which is also assigned 'free' and given a start value of .2 and a label.

```
expVars <- mxPath( from=c("X", "Y"), arrows=2,
                   free=TRUE, values=1, lbound=.01, labels=c("varX","varY") )
expCovs <- mxPath( from="X", to="Y", arrows=2,
                   free=TRUE, values=.2, lbound=.01, labels="covXY" )
```

The resulting MxPath objects 'expVars' and 'expCovs' are as follows:

```
> mxPath( from=c("X", "Y"), arrows=2,
          free=TRUE, values=1, lbound=.01, labels=c("varX","varY") )
mxPath
$from:  'X' and 'Y'
$to:  'X' and 'Y'
$arrows:  2
$values:  1
$free:  TRUE
$labels:  varX varY
$lbound:  0.01
$ubound:  NA
>  mxPath( from="X", to="Y", arrows=2,
           free=TRUE, values=.2, lbound=.01, labels="covXY" )
mxPath
$from:  'X'
$to:  'Y'
$arrows:  2
$values:  0.2
$free:  TRUE
$labels:  covXY
$lbound:  0.01
$ubound:  NA
```

When observed means are included in addition to the observed covariance matrix, as is necessary when fitting to raw data, we add an `mxPath` command with single-headed arrows from `one` to the variables to represent the two means.

```
expMeans <- mxPath( from="one", to=c("X", "Y"), arrows=1,
                    free=TRUE, values=.01, labels=c("meanX","meanY") )
```

The "one" argument in the `from` argument is used exclusively for means objects, here called *expMeans*.

```
> mxPath( from="one", to=c("X", "Y"), arrows=1,
          free=TRUE, values=.01, labels=c("meanX","meanY") )
mxPath
$from:  'one'
$to:  'X' and 'Y'
$arrows:  1
$values:  0.01
$free:  TRUE
$labels:  meanX meanY
$lbound:  NA
$ubound:  NA
```

To fit this bivariate model to the simulated data, we have to combine the data and model statements in a MxModel objects.

```
bivSatModel1 <- mxModel("bivSat1", manifestVars=selVars, obsBivData,
                        expVars, expCovs, expMeans, type="RAM" )
bivSatFit1   <- mxRun(bivSatModel1)
bivSatSumm1  <- summary(bivSatFit1)
```

As you can see below, the maximum likelihood (ML) estimates are very close to the summary statistics of the simulated data.

```
> bivSatSumm1
data:
$bivSat1.data
       X                    Y
Min.   :-3.296159   Min.   :-2.942561
1st Qu.:-0.596177   1st Qu.:-0.633711
Median :-0.010538   Median :-0.004139
Mean   :-0.004884   Mean   : 0.032116
3rd Qu.: 0.598326   3rd Qu.: 0.739236
Max.   : 4.006771   Max.   : 4.173841

free parameters:
   name matrix row col     Estimate  Std.Error Std.Estimate     Std.SE lbound ubound
1  varX      S   X   X  0.993537344 0.04443221    1.0000000 0.04472123   0.01
2 covXY      S   X   Y  0.481348846 0.03513471    0.4806856 0.03508630   0.01
3  varY      S   Y   Y  1.009283953 0.04513849    1.0000000 0.04472328   0.01
4 meanX      M   1   X -0.004884421 0.03152067           NA         NA
5 meanY      M   1   Y  0.032116307 0.03177008           NA         NA

observed statistics:  0
estimated parameters:  5
degrees of freedom:  -5
-2 log likelihood:  5415.772
saturated -2 log likelihood:  -2
number of observations:  1000
chi-square:  5417.772
p:  NaN
Information Criteria:
     df Penalty Parameters Penalty Sample-Size Adjusted
AIC:   5425.772           5425.772                  NA
BIC:   5450.311           5450.311            5434.431
```

Matrix Method

If we use matrices instead of paths to specify the bivariate model, we need to generate matrices to represent the expected covariance matrix and the means. The `mxMatrix` command for the expected covariance matrix now specifies a **2x2** symmetric matrix with all elements free. Start values have to be given only for the unique elements (diagonal elements plus upper or lower diagonal elements), in this case we provide a list with values of 1 for the variances and 0.5 for the covariance.

```
expCovM <- mxMatrix( type="Symm", nrow=2, ncol=2,
                     free=TRUE, values=c(1,0.5,1),
                     labels=c('V1','Cov','V2'), name="expCov" )
```

By specifying labels, we can tell that the two covariance elements, expCovM[1,2] and expCovM[2,1] are constrained to be equal - implied by the fact that the label is the same. This of course automatically happens when you specify the matrix to be symmetric.

```
> expCovM
SymmMatrix 'expCov'

$labels
     [,1]  [,2]
[1,] "V1"  "Cov"
[2,] "Cov" "V2"

$values
     [,1] [,2]
[1,]  1.0  0.5
[2,]  0.5  1.0

$free
     [,1] [,2]
[1,] TRUE TRUE
[2,] TRUE TRUE

$lbound: No lower bounds assigned.

$ubound: No upper bounds assigned.
```

When fitting to raw data, we also use a `mxMatrix` command to specify the expected means as **1x2** row vector with two free parameters, each given a 0 start value.

```
expMeanM <- mxMatrix( type="Full", nrow=1, ncol=2,
                      free=TRUE, values=c(0,0), labels=c('M1','M2'), name="expMean" )
```

Similarly to above, the elements in this matrix can also be given labels, although this is entirely optional for both matrices. However, as soon as we want to change the model to e.g. test equality of means or variances, the most efficient way to do this is by using labels. Given fitting alternative models to test hypotheses is very common, we highly recommend to use labels at all times. Note that we truncated the output below as no bounds had been assigned.

```
> expMeanM
FullMatrix 'expMean'

$labels
     [,1] [,2]
[1,] "M1" "M2"

$values
     [,1] [,2]
[1,]    0    0

$free
     [,1] [,2]
[1,] TRUE TRUE
```

So far, we have specified the expected covariance matrix directly as a symmetric matrix. However, this may cause optimization problems as the matrix could become not positive-definite which would prevent the likelihood to be evaluated. To overcome this problem, we can use a Cholesky decomposition of the expected covariance matrix instead, by multiplying a lower triangular matrix with its transpose. To obtain this, we use a `mxMatrix` command and define a **2x2** lower triangular matrix using `type`="Lower", declare all elements free with 0.5 starting values. We name this matrix "Chol" and save the object as *lowerTriM*.

```
lowerTriM <- mxMatrix( type="Lower", nrow=2, ncol=2,
                       free=TRUE, values=.5, name="Chol" )
```

Given we specified the matrix as lower triangular, the start values and free assignments are only applied to the elements

on the diagonal and below the diagonal.

```
> lowerTriM
LowerMatrix 'Chol'

$labels: No labels assigned.

$values
     [,1] [,2]
[1,]  0.5  0.0
[2,]  0.5  0.5

$free
     [,1]  [,2]
[1,] TRUE FALSE
[2,] TRUE  TRUE
```

We then use an `mxAlgebra` command to multiply this matrix with its transpose (R function `t()`). The `mxAlgebra` command is a very useful command to apply any operation or function to matrices. It only has two arguments, the first for the `expression` you intend to generate, the second the name of the resulting matrix. Note that although the matrix object for the lower triangular matrix was saved as 'lowerTwiM', the matrices in the `expression` are referred to by the name given to them within the MxMatrix object. This is similar to referring to the names of the expected covariance matrices and means when they are needed in the arguments of the `mxExpectationNormal` function.

```
expCovMA <- mxAlgebra( expression=Chol %*% t(Chol), name="expCov" )
```

So far, we've only specified the algebra, but not computed it yet as shown when we look at the *expCovMA* object. We need to combine all elements in an `mxModel` prior to `mxRun` ning the model to compute the algebra.

```
> expCovMA
mxAlgebra 'expCov'
$formula:  Chol %*% t(Chol)
$result: (not yet computed) <0 x 0 matrix>
dimnames: NULL
```

Given we used the same names for the resulting matrices for the expected covariances and means as in the univariate example, the `mxExpectationNormal` command looks identical. Note that we have redefined *selVars* when starting the bivariate examples. When you use the piecewise style and you're running more than one job, make sure you're not accidentally using an object from a previous job, especially if you've made an error in a newly specified object with the same name.

```
expectBiv <- mxExpectationNormal( covariance="expCov", means="expMean",
                                  dimnames=selVars )
```

Combining these two `mxMatrix` and the `mxAlgebra` objects with the raw data, specified in the `mxData` object *obsBivData* created earlier and the `mxExpectationNormal` command with the appropriate arguments is all that is needed to fit a saturated bivariate model.

```
bivSatModel2 <- mxModel("bivSat2", obsBivData, lowerTriM,
                        expCovMA, expMeanM, expectBiv, funML )
bivSatFit2   <- mxRun(bivSatModel2)
bivSatSumm2  <- summary(bivSatFit2)
```

The goodness-of-fit statistics in the output from the path and matrix specification appear identical. However, in the latter model, we do not model the variances and covariance directly but parameterize them using a Cholesky decomposition.

```
> bivSatSumm2
data:
$bivSat2.data
       X                    Y
Min.   :-3.296159   Min.   :-2.942561
1st Qu.:-0.596177   1st Qu.:-0.633711
Median :-0.010538   Median :-0.004139
Mean   :-0.004884   Mean   : 0.032116
3rd Qu.: 0.598326   3rd Qu.: 0.739236
Max.   : 4.006771   Max.   : 4.173841

free parameters:
               name  matrix row col     Estimate  Std.Error lbound ubound
1 bivSat2.Chol[1,1]    Chol   1   1  0.996763911 0.02228823
2 bivSat2.Chol[2,1]    Chol   2   1  0.482912544 0.02987720
3 bivSat2.Chol[2,2]    Chol   2   2  0.880954091 0.01969863
4                M1 expMean   1   X -0.004883967 0.03151918
5                M2 expMean   1   Y  0.032116277 0.03176869

observed statistics:  2000
estimated parameters:  5
degrees of freedom:  1995
-2 log likelihood:  5415.772
saturated -2 log likelihood:  -2
number of observations:  1000
chi-square:  5417.772
p:  2.595415e-313
Information Criteria:
     df Penalty Parameters Penalty Sample-Size Adjusted
AIC:   1425.772           5425.772                   NA
BIC:  -8365.200           5450.311             5434.431
```

We can obtain the predicted variances and covariances by printing the *expCov* matrix which can be done with the `mxEval` command - either by recalculating or by just printing the calculated algebra, or by grabbing the predicted covariance matrix from the fitted object *bivSatFit2*

```
mxEval(Chol %*% t(Chol), bivSatFit2 )
mxEval(expCov, bivSatFit2 )
bivSatFit2$expCov$result
```

So far, we have presented the bivariate model (path or matrix method) using the piecewise approach. As a result, we end up with a series of OpenMx objects, each of which we can check for syntax correctness. As such, this is a great way to build new models. An alternative approach is to start the mxModel with one argument, and then add another argument step by step. We will show the various steps of building the bivariate model with matrices (and algebras).

Here, we simply repeat all the lines that make up the model.

```
obsBivData   <- mxData( observed=bivData, type="raw" )
expMeanM     <- mxMatrix( type="Full", nrow=1, ncol=2,
                          free=TRUE, values=0, labels=c('M1','M2'), name="expMean" )
lowerTriM    <- mxMatrix( type="Lower", nrow=2, ncol=2,
                                free=TRUE, values=.5, name="Chol" )
expCovMA     <- mxAlgebra( expression=Chol %*% t(Chol), name="expCov" )
expectBiv    <- mxExpectationNormal( covariance="expCov", means="expMean",
                                     dimnames=selVars )
funML        <- mxFitFunctionML()
bivSatModel2 <- mxModel("bivSat2", obsBivData, lowerTriM,
                           expCovMA, expMeanM, expectBiv, funML )
bivSatFit2   <- mxRun(bivSatModel2)
```

```
bivSatSumm2  <- summary(bivSatFit2)
```

Stepwise Style

Looking back at the MxModel we just built (*bivSatModel2*), the first argument after the name (in quotes) was the MxData object. Let's now build a new model that just has a new name and the data object to start, specified from scratch - assuming we had not built the object before. Note we need to close both the `mxData` command which resides within the `mxModel` command and the `mxModel` command itself. We can execute it in R, to check for syntax errors.

```
bivSatModel3 <- mxModel("bivSat3", mxData( observed=bivData, type="raw" ) )
```

If it checks out to be syntactically correct, we can add another argument, i.e. the `mxMatrix` command to define the lower triangular matrix. Given we now want to build upon the previous model, we use that as the first argument (without quotes). As the previous model already has a name argument we don't need to include that and can go straight to the new argument.

```
bivSatModel3 <- mxModel(bivSatModel3,
                        mxMatrix( type="Lower", nrow=2, ncol=2,
                                  free=TRUE, values=.5, name="Chol" ) )
```

Note that we used the same name for the MxModel object to be generated, thus it will overwrite the previous one. One might choose a new name if uncertain about the syntax of the new model, to avoid having to rerun the previous step to correct the original model. As everything checks out OK after step two, let us add another argument, this time the `mxAlgebra` object for the expected covariance matrix.

```
bivSatModel3 <- mxModel(bivSatModel3,
                        mxAlgebra( expression=Chol %*% t(Chol), name="expCov" ) )
```

As everything still appears OK, we continue to add arguments. It's not necessary to do them one at the time, but if you're just learning to build a model, it might be the safest bet. Next, we add the `mxMatrix` command for the expected means.

```
bivSatModel3 <- mxModel(bivSatModel3,
                        mxMatrix( type="Full", nrow=1, ncol=2,
                                  free=TRUE, values=0, name="expMean" ) )
```

The only argument left to add is the `mxExpectationNormal` and the `mxFitFunctionML()` to specify what function to use to test the fit between covariances and means predicted by the observed data and those expected by the model.

```
bivSatModel3 <- mxModel(bivSatModel3,
                        mxExpectationNormal( covariance="expCov", means="expMean",
                                             dimnames=selVars ) )
bivSatModel3 <- mxModel(bivSatModel3, mxFitFunctionML() )
```

Now that we have the complete model built, we can evaluate it using the `mxRun` command.

```
bivSatFit3  <- mxRun(bivSatModel3)
bivSatSumm3 <- summary(bivSatFit3)
```

You can verify for yourself that the results (goodness-of-fit statistics and parameter estimates) are entirely equivalent between this stepwise approach and the piecewise approach.

We here combine all the separate lines to see the full picture.

```
bivSatModel3 <- mxModel("bivSat3", mxData( observed=bivData, type="raw" ) )
bivSatModel3 <- mxModel(bivSatModel3,
                        mxMatrix( type="Lower", nrow=2, ncol=2,
                                  free=TRUE, values=.5, name="Chol" ) )
bivSatModel3 <- mxModel(bivSatModel3,
                        mxAlgebra( expression=Chol %*% t(Chol), name="expCov" ) )
bivSatModel3 <- mxModel(bivSatModel3,
                        mxMatrix( type="Full", nrow=1, ncol=2,
                                  free=TRUE, values=0, name="expMean" )
bivSatModel3 <- mxModel(bivSatModel3,
                        mxExpectationNormal( covariance="expCov", means="expMean",
                                             dimnames=selVars ) )
bivSatFit3   <- mxRun(bivSatModel3)
bivSatSumm3  <- summary(bivSatFit3)
```

Classic Style

If you are fairly confident that you can specify each of the arguments of the model syntactically correct, there is no need to build the objects one by one and combine them, as we did using the piecewise approach, or to build models step by step by adding one argument at a time, as we did in the stepwise approach. Instead, we can generate the complete syntax at once. As a result, this will be the most concise way to write and run models. The disadvantage, however, is that if you make changes to the model, and they include a syntax error, it is less evident to find the error. The advantage, on the other hand, is that some models do not require any changes, but maybe you just want to apply them to different data sets in which case this approach works fine.

Here, we present the complete bivariate saturated model, with each argument printed on a different line for clarity of presentation. To not overwrite the previous objects, we'll start with a new name for the MxModel object. Remember that arguments have to be separated by comma's, and that we need a double bracket after the last argument to close both that argument and the full model.

```
bivSatModel4 <- mxModel("bivSat4",
                        mxData( observed=bivData, type="raw" ),
                        mxMatrix( type="Lower", nrow=2, ncol=2,
                                  free=TRUE, values=.5, name="Chol" ),
                        mxAlgebra( expression=Chol %*% t(Chol), name="expCov" ),
                        mxMatrix( type="Full", nrow=1, ncol=2,
                                  free=TRUE, values=c(0,0), name="expMean" ),
                        mxExpectationNormal( covariance="expCov", means="expMean",
                                             dimnames=selVars )
                        mxFitFunctionML() )
bivSatFit4 <- mxRun(bivSatModel4)
bivSatSumm4 <- summary(bivSatFit4)
```

Again, as you might expect by now, the output of this model run will be identical to that of both the piecewise and the stepwise approach. Given their equivalence, it is really up to the OpenMx user to decide which method (path or matrix) and which approach (piecewise, stepwise or classic) is preferred. It is also not necessary to pick just one of these approaches, as they can be 'mixed and matched'. For didactic purposes, we recommend the piecewise approach, which we will use in the majority of this documentation. We will, however, provide some parallel classic scripts. Furthermore, given some people do better with path diagrams and others with matrix algebra, we present some models both ways, in so far that this is doable.

The following sections describe OpenMx examples in detail beginning with regression, factor analysis, time series analysis, multiple group models, including twin models, and analysis using definition variables. Each is presented in both path and matrix styles and where relevant, contrasting data input from covariance matrices versus raw data input are also illustrated. Additional examples will be added as they are implemented in OpenMx.

CHAPTER

TWO

EXAMPLES, PATH SPECIFICATION

2.1 Regression, Path Specification

This example will show how regression can be carried out from a path-centric structural modeling perspective. This example is in three parts; a simple regression, a multiple regression, and multivariate regression. There are two versions of each example available; one where the data is supplied as a covariance matrix and vector of means, and one with raw data. These examples are available in the following files:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/SimpleRegression_PathCov.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/SimpleRegression_PathRaw.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/MultipleRegression_PathCov.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/MultipleRegression_PathRaw.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/MultivariateRegression_PathCov.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/MultivariateRegression_PathRaw.R

Parallel versions of these examples, using matrix specification of models rather than paths, can be found here:

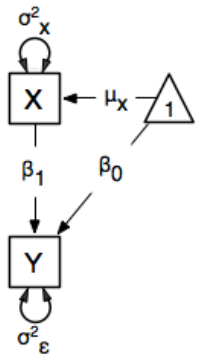
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/SimpleRegression_MatrixCov.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/SimpleRegression_MatrixRaw.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/MultipleRegression_MatrixCov.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/MultipleRegression_MatrixRaw.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/MultivariateRegression_MatrixCov.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/MultivariateRegression_MatrixRaw.R

and are discussed here (*Regression, Matrix Specification*).

2.1.1 Simple Regression

We begin with a single dependent variable (y) and a single independent variable (x). The relationship between these variables takes the following form:

$$y = \beta_0 + \beta_1 * x + \epsilon$$



In this model, the mean of y is dependent on both regression coefficients (and by extension, the mean of x). The variance of y depends on both the residual variance (σ^2_ϵ) and the product of the regression slope (β_1) and the variance of x (σ^2_x). This model contains five parameters from a structural modeling perspective β_0, β_1, σ^2_ϵ, and the mean and variance of x, μ^2_x and σ^2_x. We are modeling a covariance matrix with three degrees of freedom (two variances and one covariance) and a means vector with two degrees of freedom (two means). Because the model has as many parameters (5) as the data have degrees of freedom, this model is fully saturated.

Data

Our first step to running this model is to include the data to be analyzed. The data must first be placed in a variable or object. For raw data, this can be done with the `read.table` function. The data provided has a header row, indicating the names of the variables.

```
data(myRegDataRaw)
```

The names of the variables provided by the header row can be displayed with the `names()` function.

```
names(myRegDataRaw)
```

As you can see, our data has four variables in it. However, our model only contains two variables, x and y. To use only them, we will select only the variables we want and place them back into our data object. That can be done with the R code below.

```
SimpleDataRaw <- myRegDataRaw[,c("x","y")]
```

For covariance data, we do something very similar. We create an object to house our data. Instead of reading in raw data from an external file, we can include a covariance matrix. This requires the `matrix()` function, which needs to know what values are in the covariance matrix, how big it is, and what the row and column names are. As our model also references means, we will include a vector of means in a separate object. Data is selected in the same way as before.

```
myRegDataCov <- matrix(
    c(0.808,-0.110, 0.089, 0.361,
     -0.110, 1.116, 0.539, 0.289,
      0.089, 0.539, 0.933, 0.312,
      0.361, 0.289, 0.312, 0.836),
    nrow=4, dimnames=list( c("w","x","y","z"), c("w","x","y","z")) )

SimpleDataCov <- myRegDataCov[c("x","y"),c("x","y")]

myRegDataMeans <- c(2.582, 0.054, 2.574, 4.061)
names(myRegDataMeans) <- c("w","x","y","z")

SimpleDataMeans <- myRegDataMeans[c(2,3)]
```

Model Specification

The following code contains all of the components of our model. Before running a model, the OpenMx library must be loaded into R using either the `require()` or `library()` function. All objects required for estimation (data, paths, and a model type) are included in their own arguments or functions. This code uses the `mxModel` function to create an `MxModel` object, which we will then run. Note the difference in capitalization for the first letter.

```
require(OpenMx)

dataRaw      <- mxData( observed=SimpleDataRaw,  type="raw" )
# variance paths
varPaths     <- mxPath( from=c("x","y"), arrows=2,
                        free=TRUE, values = c(1,1), labels=c("varx","residual") )
# regression weights
regPaths     <- mxPath( from="x", to="y", arrows=1,
                        free=TRUE, values=1, labels="beta1" )
# means and intercepts
means        <- mxPath( from="one", to=c("x","y"), arrows=1,
                        free=TRUE, values=c(1,1), labels=c("meanx","beta0") )

uniRegModel  <- mxModel(model="Simple Regression Path Specification", type="RAM",
                        dataRaw, manifestVars=c("x","y"), varPaths, regPaths, means)
```

We are presenting the code here in the piecewise style and thus will create several of the pieces up front before putting them together in the `mxModel` statement. We will pre-specify the MxData object *dataRaw*, and the various MxPath objects to define the variance paths *varPaths*, regression weights *regPaths* and the means and intercepts in *means*. These are then included as arguments of the MxModel object.

This `mxModel` function can be split into several parts. First, we give the model a title. The first argument in an `mxModel` function has a special function. If an object or variable containing an `MxModel` object is placed here, then `mxModel` adds to or removes pieces from that model. If a character string (as indicated by double quotes) is placed first, then that becomes the name of the model. Models may also be named by including a `name` argument. This model is named "Simple Regression Path Specification".

The next part of our code is the `type` argument. By setting `type="RAM"`, we tell OpenMx that we are specifying a RAM model for covariances and means, and that we are doing so using the `mxPath` function. With this setting, OpenMx uses the specified paths to define the expected covariance and means of our data.

The third component of our code creates an `MxData` object. The example above, reproduced here in parts, first references the object where our data is, then uses the `type` argument to specify that this is raw data.

```
dataRaw      <- mxData( observed=SimpleDataRaw, type="raw" )
```

If we were to use a covariance matrix and vector of means as data, we would replace the existing `mxData` function with this one:

```
dataCov      <- mxData( observed=SimpleDataCov, type="cov", numObs=100,
                        means=SimpleDataMeans )
```

We must also specify the list of observed variables using the `manifestVars` argument. In the code below, we include a list of both observed variables, *x* and *y*.

```
manifestVars=c("x", "y")
```

The last features of our code are three `mxPath` functions, which describe the relationships between variables. Each function first describes the variables involved in any path. Paths go from the variables listed in the `from` argument, and to the variables listed in the `to` argument. When `arrows` is set to `1`, then one-headed arrows (regressions) are drawn from the `from` variables to the `to` variables. When `arrows` is set to `2`, two headed arrows (variances or

covariances) are drawn from the the `from` variables to the `to` variables. If `arrows` is set to `2`, then the `to` argument may be omitted to draw paths both to and from the list of `from` variables.

The variance terms of our model (that is, the variance of *x* and the residual variance of *y*) are created with the following `mxPath` function. We want two headed arrows from *x* to *x*, and from *y* to *y*. These paths should be freely estimated (`free=TRUE`), have starting values of `1`, and be labeled `"varx"` and `"residual"`, respectively.

```
# variance paths
varPaths     <- mxPath( from=c("x","y"), arrows=2,
                        free=TRUE, values = c(1,1), labels=c("varx","residual") )
```

The regression term of our model (that is, the regression of *y* on *x*) is created with the following `mxPath` function. We want a single one-headed arrow from *x* to *y*. This path should be freely estimated (`free=TRUE`), have a starting value of `1`, and be labeled `"beta1"`.

```
# regression weights
regPaths     <- mxPath( from="x", to="y", arrows=1,
                        free=TRUE, values=1, labels="beta1" )
```

We also need means and intercepts in our model. Exogenous or independent variables have means, while endogenous or dependent variables have intercepts. These can be included by regressing both *x* and *y* on a constant, which can be refered to in OpenMx by `"one"`. The intercept terms of our model are created with the following `mxPath` function. We want single one-headed arrows from the constant to both *x* and *y*. These paths should be freely estimated (`free=TRUE`), have a starting value of `1`, and be labeled `meanx` and `"beta1"`, respectively.

```
# means and intercepts
means        <- mxPath( from="one", to=c("x","y"), arrows=1,
                        free=TRUE, values=c(1,1), labels=c("meanx","beta0") )
```

Our model is now complete!

Model Fitting

We've created an `MxModel` object, and placed it into an object or variable named *uniRegModel*. We can run this model by using the `mxRun` function, and the output is placed in the object *uniRegFit* in the code below. We then view the output by referencing the `output` slot, as shown here.

```
uniRegFit <- mxRun(uniRegModel)
```

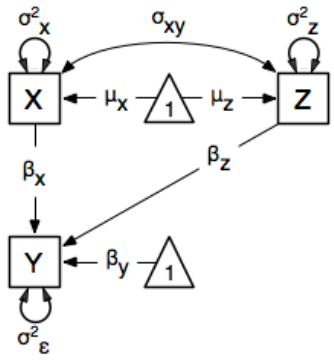
The `output` slot contains a great deal of information, including parameter estimates and information about the matrix operations underlying our model. A more parsimonious report on the results of our model can be viewed using the `summary` function, as shown here.

```
uniRegFit$output
summary(uniRegFit)
```

2.1.2 Multiple Regression

In the next part of this demonstration, we move to multiple regression. The regression equation for our model looks like this:

$$y = \beta_0 + \beta_x * x + \beta_z * z + \epsilon$$



Our dependent variable y is now predicted from two independent variables, x and z. Our model includes 3 regression parameters (β_0, β_x, β_z), a residual variance (σ^2_ϵ) and the observed means, variances and covariance of x and z, for a total of 9 parameters. Just as with our simple regression, this model is fully saturated.

We prepare our data the same way as before, selecting three variables instead of two.

```
MultipleDataRaw <- myRegDataRaw[,c("x","y","z")]

MultipleDataCov <- myRegDataCov[c("x","y","z"),c("x","y","z")]

MultipleDataMeans <- myRegDataMeans[c(2,3,4)]
```

Now, we can move on to our code. It is identical in structure to our simple regression code, but contains additional paths for the new parts of our model.

```
require(OpenMx)

dataCov      <- mxData( observed=MultipleDataCov,  type="cov", numObs=100,
                        means=MultipleDataMeans )
# variance paths
varPaths     <- mxPath( from=c("x","y","z"),  arrows=2,
                        free=TRUE, values = c(1,1,1), labels=c("varx","res","varz") )
# covariance of x and z
covPaths     <- mxPath( from="x", to="z", arrows=2,
                        free=TRUE, values=0.5, labels="covxz" )
# regression weights
regPaths     <- mxPath( from=c("x","z"), to="y", arrows=1,
                        free=TRUE, values=1, labels=c("betax","betaz") )
# means and intercepts
means        <- mxPath( from="one", to=c("x","y","z"), arrows=1,
                        free=TRUE, values=c(1,1), labels=c("meanx","beta0","meanz") )

multiRegModel <- mxModel("Multiple Regression Path Specification", type="RAM",
                        dataCov, manifestVars=c("x","y","z"),
                        varPaths, covPaths, regPaths, means)

multiRegFit <- mxRun(multiRegModel)

multiRegFit$output
summary(multiRegFit)
```

As the code should look more or less familiar, we will focus on the parts that are new or changed. As I'm sure you know by now, `require(OpenMx)` makes sure the OpenMx library is loaded into R. This only needs to be done at the first model of any R session. Note that we will discuss the various objects of the piecewise style script as they are included in the `mxModel` statement.

First, the title is changed to reflect the purpose of this model. The `type="RAM"` argument is identical. The `mxData` function references our multiple regression data, which contains one more variable than our simple regression data, and is saved in the *dataCov* object. Similarly, our `manifestVars` list contains an extra label, `"z"`.

The `mxPath` functions work just as before. Our first function defines the variances of our variables. Whereas our simple regression included just the variance of x and the residual variance of y, our multiple regression includes the variance of z as well.

Our second `mxPath` function specifies a two-headed arrow (covariance) between x and z. We've omitted the `to` argument from two-headed arrows up until now, as we have only required variances. Covariances may be specified by using both the `from` and `to` arguments. This path is freely estimated, has a starting value of 0.5, and is labeled `covxz`.

```
# covariance of x and z
covPaths     <- mxPath( from="x", to="z", arrows=2,
                        free=TRUE, values=0.5, labels="covxz" )
```

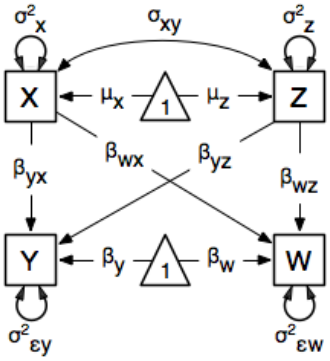
The third and fourth `mxPath` functions mirror the second and third `mxPath` functions from our simple regression, defining the regressions of y on both x and z as well as the means and intercepts of our model.

The model is run and output is viewed just as before, using the `mxRun` function, and `$output` and the `summary` function to run, view and summarize the completed model.

2.1.3 Multivariate Regression

The structural modeling approach allows for the inclusion of not only multiple independent variables (i.e., multiple regression), but multiple dependent variables as well (i.e., multivariate regression). Versions of multivariate regression are sometimes fit under the heading of path analysis. This model will extend the simple and multiple regression frameworks we've discussed above, adding a second dependent variable w.

$$y = \beta_y + \beta_{yx} * x + \beta_{yz} * z + \epsilon_y$$
$$w = \beta_w + \beta_{wx} * x + \beta_{wz} * z + \epsilon_w$$



We now have twice as many regression parameters, a second residual variance, and the same means, variances and covariances of our independent variables. As with all of our other examples, this is a fully saturated model.

Data import for this analysis will actually be slightly simpler than before. The data we imported for the previous examples contains only the four variables we need for this model. We can use `myRegDataRaw`, `myRegDataCov`, and `myRegDataMeans` in our models.

```
data(myRegDataRaw)

myRegDataCov <- matrix(
    c(0.808,-0.110, 0.089, 0.361,
```

```
     -0.110, 1.116, 0.539, 0.289,
      0.089, 0.539, 0.933, 0.312,
      0.361, 0.289, 0.312, 0.836),
    nrow=4, dimnames=list( c("w","x","y","z"), c("w","x","y","z")) )

myRegDataMeans <- c(2.582, 0.054, 2.574, 4.061)
```

Our code should look very similar to our previous two models. It includes the same `type` argument, `mxData` function, and `manifestVars` argument as previous models, with a different version of the data and additional variables in the latter two components.

```
dataRaw      <- mxData( observed=myRegDataRaw, type="raw" )
# variance paths
varPaths     <- mxPath( from=c("w","x","y","z"), arrows=2,
                        free=TRUE, values=1,
                        labels=c("residualw","varx","residualy","varz") )
# covariance of x and z
covPaths     <- mxPath( from="x", to="z", arrows=2,
                        free=TRUE, values=0.5, labels="covxz" )
# regression weights for y
regPathsY    <- mxPath( from=c("x","z"), to="y", arrows=1,
                        free=TRUE, values=1, labels=c("betayx","betayz") )
# regression weights for w
regPathsW    <- mxPath( from=c("x","z"), to="w", arrows=1,
                        free=TRUE, values=1, labels=c("betawx","betawz") )
# means and intercepts
means        <- mxPath( from="one", to=c("w","x","y","z"), arrows=1,
                        free=TRUE, values=c(1, 1),
                        labels=c("betaw","meanx","betay","meanz") )

multivariateRegModel <- mxModel("MultiVariate Regression Path Specification",
                        type="RAM", dataRaw, manifestVars=c("w","x","y","z"),
                        varPaths, covPaths, regPathsY, regPathsW, means )

multivariateRegFit <- mxRun(multivariateRegModel)

multivariateRegFit$output
summary(multivariateRegFit)
```

The only additional components to our `mxPath` functions are the inclusion of the *w* variable and the additional set of regression coefficients for *w*. Running the model and viewing output works exactly as before.

These models may also be specified using matrices instead of paths. See *Regression, Matrix Specification* for matrix specification of these models.

2.2 Factor Analysis, Path Specification

This example will demonstrate latent variable modeling via the common factor model using path-centric model specification. We'll walk through two applications of this approach: one with a single latent variable, and one with two latent variables. As with previous examples, these two applications are split into four files, with each application represented separately with raw and covariance data. These examples can be found in the following files:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/OneFactorModel_PathCov.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/OneFactorModel_PathRaw.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/TwoFactorModel_PathCov.R

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/TwoFactorModel_PathRaw.R

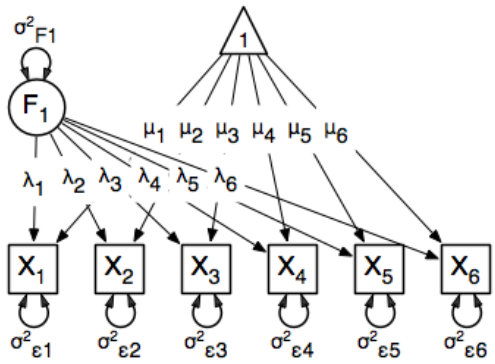
Parallel versions of this example, using matrix specification of models rather than paths, can be found here:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/OneFactorModel_MatrixCov.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/OneFactorModel_MatrixRaw.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/TwoFactorModel_MatrixCov.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/TwoFactorModel_MatrixRaw.R

2.2.1 Common Factor Model

The common factor model is a method for modeling the relationships between observed variables believed to measure or indicate the same latent variable. While there are a number of exploratory approaches to extracting latent factor(s), this example uses structural modeling to fit confirmatory factor models. The model for any person and path diagram of the common factor model for a set of variables x_1-x_6 are given below.

$$x_{ij} = \mu_j + \lambda_j * \eta_i + \epsilon_{ij}$$



While 19 parameters are displayed in the equation and path diagram above (six manifest variances, six manifest means, six factor loadings and one factor variance), we must constrain either the factor variance or one factor loading to a constant to identify the model and scale the latent variable. As such, this model contains 18 parameters. Unlike the manifest variable examples we've run up until now, this model is not fully saturated. The means and covariance matrix for six observed variables contain 27 degrees of freedom, and thus our model contains 9 degrees of freedom.

Data

Our first step to running this model is to include the data to be analyzed. The data for this example contain nine variables. We'll select the six we want for this model using the selection operators used in previous examples. Both raw and covariance data are included below, but only one is required for any model.

```
data(myFADataRaw)
names(myFADataRaw)

oneFactorRaw <- myFADataRaw[,c("x1", "x2", "x3", "x4", "x5", "x6")]

myFADataCov <- matrix(
    c(0.997, 0.642, 0.611, 0.672, 0.637, 0.677, 0.342, 0.299, 0.337,
      0.642, 1.025, 0.608, 0.668, 0.643, 0.676, 0.273, 0.282, 0.287,
      0.611, 0.608, 0.984, 0.633, 0.657, 0.626, 0.286, 0.287, 0.264,
      0.672, 0.668, 0.633, 1.003, 0.676, 0.665, 0.330, 0.290, 0.274,
      0.637, 0.643, 0.657, 0.676, 1.028, 0.654, 0.328, 0.317, 0.331,
```

```
      0.677, 0.676, 0.626, 0.665, 0.654, 1.020, 0.323, 0.341, 0.349,
      0.342, 0.273, 0.286, 0.330, 0.328, 0.323, 0.993, 0.472, 0.467,
      0.299, 0.282, 0.287, 0.290, 0.317, 0.341, 0.472, 0.978, 0.507,
      0.337, 0.287, 0.264, 0.274, 0.331, 0.349, 0.467, 0.507, 1.059), nrow=9,
dimnames=list( c("x1","x2","x3","x4","x5","x6","y1","y2","y3"),
               c("x1","x2","x3","x4","x5","x6","y1","y2","y3")) )

oneFactorCov <- myFADataCov[c("x1","x2","x3","x4","x5","x6"),
                            c("x1","x2","x3","x4","x5","x6")]

myFADataMeans <- c(2.988, 3.011, 2.986, 3.053, 3.016, 3.010, 2.955, 2.956, 2.967)
names(myFADataMeans) <- c("x1","x2","x3","x4","x5","x6","y1","y2","y3")

oneFactorMeans <- myFADataMeans[1:6]
```

Model Specification

Creating a path-centric factor model will use many of the same functions and arguments used in previous path-centric examples. However, the inclusion of latent variables adds a few extra pieces to our model. Before running a model, the OpenMx library must be loaded into R using either the `require()` or `library()` function. All objects required for estimation (data, paths, and a model type) are included in their own arguments or functions. This code uses the `mxModel` function to create an `MxModel` object, which we will then run.

```
require(OpenMx)

dataRaw      <- mxData( observed=myFADataRaw, type="raw" )
# residual variances
resVars      <- mxPath( from=c("x1","x2","x3","x4","x5","x6"), arrows=2,
                        free=TRUE, values=c(1,1,1,1,1,1),
                        labels=c("e1","e2","e3","e4","e5","e6") )
# latent variance
latVar       <- mxPath( from="F1", arrows=2,
                        free=TRUE, values=1, labels ="varF1" )
# factor loadings
facLoads     <- mxPath( from="F1", to=c("x1","x2","x3","x4","x5","x6"), arrows=1,
                        free=c(FALSE,TRUE,TRUE,TRUE,TRUE,TRUE), values=c(1,1,1,1,1,1),
                        labels =c("l1","l2","l3","l4","l5","l6") )
# means
means        <- mxPath( from="one", to=c("x1","x2","x3","x4","x5","x6","F1"), arrows=1,
                        free=c(T,T,T,T,T,T,FALSE), values=c(1,1,1,1,1,1,0),
                        labels =c("meanx1","meanx2","meanx3",
                                  "meanx4","meanx5","meanx6",NA) )

oneFactorModel <- mxModel("Common Factor Model Path Specification", type="RAM",
                        manifestVars=c("x1","x2","x3","x4","x5","x6"), latentVars="F1",
                        dataRaw, resVars, latVar, facLoads, means)
```

As with previous examples, this model begins with a name ("Common Factor Model Path Specification") for the model and a `type="RAM"` argument. The name for the model may be omitted, or may be specified in any other place in the model using the `name` argument. Including `type="RAM"` allows the `mxModel` function to interpret the `mxPath` functions that follow and turn those paths into an expected covariance matrix and means vector for the ensuing data. The `mxData` function works just as in previous examples, and the following raw data specification is included in the code:

```
dataRaw      <- mxData( observed=myFADataRaw, type="raw" )
```

can be replaced with a covariance matrix and means, like so:

```
dataCov      <- mxData( observed=oneFactorCov, type="cov", numObs=500,
                        means=oneFactorMeans )
```

The first departure from our previous examples can be found in the addition of the `latentVars` argument after the `manifestVars` argument. The `manifestVars` argument includes the six variables in our observed data. The `latentVars` argument provides names for the latent variables (here just one), so that it may be referenced in `mxPath` functions.

```
manifestVars=c("x1","x2","x3","x4","x5","x6")
latentVars="F1"
```

Our model is defined by four `mxPath` functions. The first defines the residual variance terms for our six observed variables. The `to` argument is not required, as we are specifiying two headed arrows both from and to the same variables, as specified in the `from` argument. These six variances are all freely estimated, have starting values of 1, and are labeled `e1` through `e6`.

```
# residual variances
resVars      <- mxPath( from=c("x1","x2","x3","x4","x5","x6"), arrows=2,
                        free=TRUE, values=c(1,1,1,1,1,1),
                        labels=c("e1","e2","e3","e4","e5","e6") )
```

We also must specify the variance of our latent variable. This code is identical to our residual variance code above, with the latent variable `"F1"` replacing our six manifest variables. Alternatively, both could be combined.

```
# latent variance
latVar       <- mxPath( from="F1", arrows=2,
                        free=TRUE, values=1, labels ="varF1" )
```

Next come the factor loadings. These are specified as asymmetric paths (regressions) of the manifest variables on the latent variable `"F1"`. As we have to scale the latent variable, the first factor loading has been given a fixed value of one by setting the first elements of the `free` and `values` arguments to `FALSE` and `1`, respectively. Alternatively, the latent variable could have been scaled by fixing the factor variance to 1 in the previous `mxPath` function and freely estimating all factor loadings. The five factor loadings that are freely estimated are all given starting values of 1 and labels `l2` through `l6`.

```
# factor loadings
facLoads     <- mxPath( from="F1", to=c("x1","x2","x3","x4","x5","x6"), arrows=1,
                        free=c(FALSE,TRUE,TRUE,TRUE,TRUE,TRUE), values=c(1,1,1,1,1,1),
                        labels =c("l1","l2","l3","l4","l5","l6") )
```

Lastly, we must specify the mean structure for this model. As there are a total of seven variables in this model (six manifest and one latent), we have the potential for seven means. However, we must constrain at least one mean to a constant value, as there is not sufficient information to yield seven mean and intercept estimates from the six observed means. The six observed variables receive freely estimated intercepts, while the factor mean is fixed to a value of zero in the code below.

```
# means
means        <- mxPath( from="one", to=c("x1","x2","x3","x4","x5","x6","F1"), arrows=1,
                        free=c(T,T,T,T,T,T,FALSE), values=c(1,1,1,1,1,1,0),
                        labels =c("meanx1","meanx2","meanx3","meanx4","meanx5","meanx6",NA) )
```

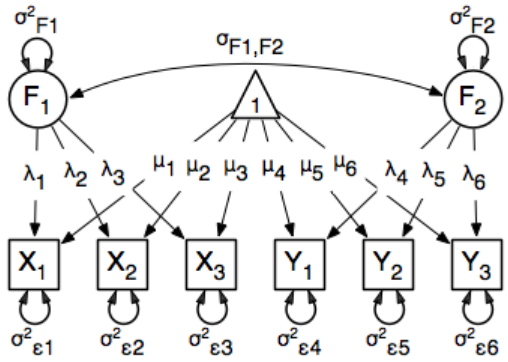
The model can now be run using the `mxRun` function, and the output of the model can be accessed from the `output` slot of the resulting model. A summary of the output can be reached using `summary()`.

```
oneFactorFit <- mxRun(oneFactorModel)

oneFactorFit$output
summary(oneFactorFit)
```

2.2.2 Two Factor Model

The common factor model can be extended to include multiple latent variables. The model for any person and path diagram of the common factor model for a set of variables x_1-x_3 and y_1-y_3 are given below.

$$x_{ij} = \mu_j + \lambda_j * \eta_{1i} + \epsilon_{ij}$$
$$y_{ij} = \mu_j + \lambda_j * \eta_{2i} + \epsilon_{ij}$$



Our model contains 21 parameters (six manifest variances, six manifest means, six factor loadings, two factor variances and one factor covariance), but each factor requires one identification constraint. Like in the common factor model above, we will constrain one factor loading for each factor to a value of one. As such, this model contains 19 parameters. The means and covariance matrix for six observed variables contain 27 degrees of freedom, and thus our model contains 8 degrees of freedom.

The data for the two factor model can be found in the `myFAData` files introduced in the common factor model. For this model, we will select three *x* variables (`x1-x3`) and three *y* variables (`y1-y3`).

```
twoFactorRaw <- myFADataRaw[,c("x1","x2","x3","y1","y2","y3")]

twoFactorCov <- myFADataCov[c("x1","x2","x3","y1","y2","y3"),
                            c("x1","x2","x3","y1","y2","y3")]

twoFactorMeans <- myFADataMeans[c(1:3,7:9)]
```

Specifying the two factor model is virtually identical to the single factor case. The last three variables of our `manifestVars` argument have changed from `"x4","x5","x6"` to `"y1","y2","y3"`, which is carried through references to the variables in later `mxPath` functions.

```
dataRaw      <- mxData( observed=twoFactorRaw, type="raw" )
# residual variances
resVars      <- mxPath( from=c("x1", "x2", "x3", "y1", "y2", "y3"), arrows=2,
                        free=TRUE, values=c(1,1,1,1,1,1),
                        labels=c("e1","e2","e3","e4","e5","e6") )
# latent variances and covariance
latVars      <- mxPath( from=c("F1","F2"), arrows=2, connect="unique.pairs",
                        free=TRUE, values=c(1,.5,1), labels=c("varF1","cov","varF2") )
# factor loadings for x variables
facLoadsX    <- mxPath( from="F1", to=c("x1","x2","x3"), arrows=1,
                        free=c(F,T,T), values=c(1,1,1), labels=c("l1","l2","l3") )
# factor loadings for y variables
facLoadsY    <- mxPath( from="F2", to=c("y1","y2","y3"), arrows=1,
                        free=c(F,T,T), values=c(1,1,1), labels=c("l4","l5","l6") )
# means
means        <- mxPath( from="one", to=c("x1","x2","x3","y1","y2","y3","F1","F2"),
```

```
                        arrows=1,
                        free=c(T,T,T,T,T,T,F,F), values=c(1,1,1,1,1,1,0,0),
                        labels=c("meanx1","meanx2","meanx3",
                                 "meany1","meany2","meany3",NA,NA) )

twoFactorModel <- mxModel("Two Factor Model Path Specification", type="RAM",
                        manifestVars=c("x1", "x2", "x3", "y1", "y2", "y3"),
                        latentVars=c("F1","F2"),
                        dataRaw, resVars, latVars, facLoadsX, facLoadsY, means)
```

We've covered the `type` argument, `mxData` function and `manifestVars` and `latentVars` arguments previously, so now we will focus on the changes this model makes to the `mxPath` functions. The first and last `mxPath` functions, which detail residual variances and intercepts, accomodate the changes in manifest and latent variables but carry out identical functions to the common factor model.

```
# residual variances
resVars      <- mxPath( from=c("x1", "x2", "x3", "y1", "y2", "y3"), arrows=2,
                        free=TRUE, values=c(1,1,1,1,1,1),
                        labels=c("e1","e2","e3","e4","e5","e6") )
# means
means        <- mxPath( from="one", to=c("x1","x2","x3","y1","y2","y3","F1","F2"),
                        arrows=1,
                        free=c(T,T,T,T,T,T,F,F), values=c(1,1,1,1,1,1,0,0),
                        labels=c("meanx1","meanx2","meanx3",
                                 "meany1","meany2","meany3",NA,NA) )
```

The second, third and fourth `mxPath` functions provide some changes to the model. The second `mxPath` function specifies the variances and covariance of the two latent variables. Like previous examples, we've omitted the `to` argument for this set of two-headed paths. Unlike previous examples, we've set the `connect` argument to `unique.pairs`, which creates all unique paths between the variables. As omitting the `to` argument is identical to putting identical variables in the `from` and `to` arguments, we are creating all unique paths from and to our two latent variables. This results in three paths: from F1 to F1 (the variance of F1), from F1 to F2 (the covariance of the latent variables), and from F2 to F2 (the variance of F2).

```
# latent variances and covariance
latVars      <- mxPath( from=c("F1","F2"), arrows=2, connect="unique.pairs",
                        free=TRUE, values=c(1,.5,1), labels=c("varF1","cov","varF2") )
```

The third and fourth `mxPath` functions define the factor loadings for each of the latent variables. We've split these loadings into two functions, one for each latent variable. The first loading for each latent variable is fixed to a value of one, just as in the previous example.

```
# factor loadings for x variables
facLoadsX    <- mxPath( from="F1", to=c("x1","x2","x3"), arrows=1,
                        free=c(F,T,T), values=c(1,1,1), labels=c("l1","l2","l3") )
# factor loadings for y variables
facLoadsY    <- mxPath( from="F2", to=c("y1","y2","y3"), arrows=1,
                        free=c(F,T,T), values=c(1,1,1), labels=c("l4","l5","l6") )
```

The model can now be run using the `mxRun` function, and the output of the model can be accessed from the `$output` slot of the resulting model. A summary of the output can be reached using `summary()`.

```
oneFactorFit <- mxRun(oneFactorModel)

oneFactorFit$output
summary(oneFactorFit)
```

These models may also be specified using matrices instead of paths. See *Factor Analysis, Matrix Specification* for matrix specification of these models.

2.3 Time Series, Path Specification

This example will demonstrate a growth curve model using path-centric specification. As with previous examples, this application is split into two files, one each for raw and covariance data. These examples can be found in the following files:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/LatentGrowthCurveModel_PathRaw.R

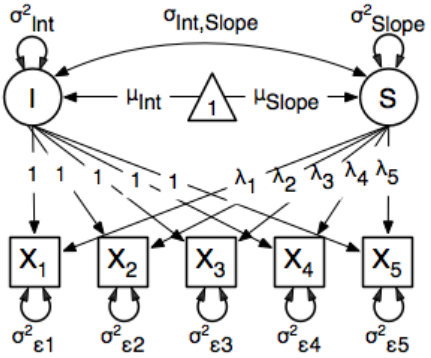
Parallel versions of this example, using matrix specification of models rather than paths, can be found here:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/LatentGrowthCurveModel_MatrixRaw.R

2.3.1 Latent Growth Curve Model

The latent growth curve model is a variation of the factor model for repeated measurements. For a set of manifest variables x_{i1} - x_{i5} measured at five discrete times for people indexed by the letter i, the growth curve model can be expressed both algebraically and via a path diagram as shown here:

$$x_{ij} = Intercept_i + \lambda_j * Slope_i + \epsilon_i$$



The values and specification of the λ parameters allow for alterations to the growth curve model. This example will utilize a linear growth curve model, so we will specify λ to increase linearly with time. If the observations occur at regular intervals in time, then λ can be specified with any values increasing at a constant rate. For this example, we will use [0, 1, 2, 3, 4] so that the intercept represents scores at the first measurement occasion, and the slope represents the rate of change per measurement occasion. Any linear transformation of these values can be used for linear growth curve models.

Our model for any number of variables contains six free parameters; two factor means, two factor variances, a factor covariance and a (constant) residual variance for the manifest variables. Our data contains five manifest variables, and so the covariance matrix and means vector contain 20 degrees of freedom. Thus, the linear growth curve model fit to these data has 14 degrees of freedom.

Data

The first step to running our model is to import data. The code below is used to import both raw data and a covariance matrix and means vector, either of which can be used for our growth curve model. This data contains five variables, which are repeated measurements of the same variable *x*. As growth curve models make specific hypotheses about the variances of the manifest variables, correlation matrices generally aren't used as data for this model.

```
data(myLongitudinalData)

myLongitudinalDataCov<-matrix(
```

```
    c(6.362, 4.344, 4.915,  5.045,  5.966,
      4.344, 7.241, 5.825,  6.181,  7.252,
      4.915, 5.825, 9.348,  7.727,  8.968,
      5.045, 6.181, 7.727, 10.821, 10.135,
      5.966, 7.252, 8.968, 10.135, 14.220), nrow=5,
    dimnames=list( c("x1","x2","x3","x4","x5"), c("x1","x2","x3","x4","x5")) )

myLongitudinalDataMeans <- c(9.864, 11.812, 13.612, 15.317, 17.178)
names(myLongitudinalDataMeans) <- c("x1","x2","x3","x4","x5")
```

Model Specification

We'll create a path-centric factor model with the same functions and arguments used in previous path-centric examples. This model is a special type of two-factor model, with fixed factor loadings, constant residual variance and manifest means dependent on latent means.

Before running a model, the OpenMx library must be loaded into R using either the `require()` or `library()` function. This code uses the `mxModel` function to create an `MxModel` object, which we will then run.

```
require(OpenMx)

dataRaw      <- mxData( observed=myLongitudinalData, type="raw" )
# residual variances
resVars      <- mxPath( from=c("x1","x2","x3","x4","x5"), arrows=2,
                        free=TRUE,  values = c(1,1,1,1,1),
                        labels=c("residual","residual","residual","residual","residual") )
# latent variances and covariance
latVars      <- mxPath( from=c("intercept","slope"), arrows=2, connect="unique.pairs",
                        free=TRUE, values=c(1,1,1), labels=c("vari","cov","vars") )
# intercept loadings
intLoads     <- mxPath( from="intercept", to=c("x1","x2","x3","x4","x5"), arrows=1,
                        free=FALSE, values=c(1,1,1,1,1) )
# slope loadings
sloLoads     <- mxPath( from="slope", to=c("x1","x2","x3","x4","x5"), arrows=1,
                        free=FALSE, values=c(0,1,2,3,4) )
# manifest means
manMeans     <- mxPath( from="one", to=c("x1","x2","x3","x4","x5"), arrows=1,
                        free=FALSE, values=c(0,0,0,0,0) )
# latent means
latMeans     <- mxPath( from="one", to=c("intercept", "slope"), arrows=1,
                        free=TRUE, values=c(1,1), labels=c("meani","means") )

growthCurveModel <- mxModel("Linear Growth Curve Model Path Specification", type="RAM",
                            manifestVars=c("x1","x2","x3","x4","x5"),
                            latentVars=c("intercept","slope"),
                            dataRaw, resVars, latVars, intLoads, sloLoads,
                            manMeans, latMeans)
```

The model begins with a name, in this case "Linear Growth Curve Model Path Specification". If the first argument is an object containing an `MxModel` object, then the model created by the `mxModel` function will contain all of the named entities in the referenced model object. The `type="RAM"` argument specifies a RAM model, allowing the `mxModel` to define an expected covariance matrix from the paths we supply.

Data is supplied with the `mxData` function. This example uses raw data, but the `mxData` function in the code above could be replaced with the function below to include covariance data.

```
dataCov      <- mxData( myLongitudinalDataCov, type="cov", numObs=500,
                        means=myLongitudinalDataMeans )
```

Next, the manifest and latent variables are specified with the `manifestVars` and `latentVars` arguments. The two latent variables in this model are named `"intercept"` and `"slope"`.

There are six `mxPath` functions in this model. The first two specify the variances of the manifest and latent variables, respectively. The manifest variables are specified below, which take the form of residual variances. The `to` argument is omitted, as it is not required to specify two-headed arrows. The residual variances are freely estimated, but held to a constant value across the five measurement occasions by giving all five variances the same label, `residual`.

```
# residual variances
resVars      <- mxPath( from=c("x1","x2","x3","x4","x5"), arrows=2,
                        free=TRUE,  values = c(1,1,1,1,1),
                        labels=c("residual","residual","residual","residual","residual") )
```

Next are the variances and covariance of the two latent variables. Like the last function, we've omitted the `to` argument for this set of two-headed paths. However, we've set the `connect` argument to `unique`, which creates all unique paths between the variables. As omitting the `to` argument is identical to putting identical variables in the `from` and `to` arguments, we are creating all unique paths from and to our two latent variables. This results in three paths: from intercept to intercept (the variance of the intercepts), from intercept to slope (the covariance of the latent variables), and from slope to slope (the variance of the slopes).

```
# latent variances and covariance
latVars      <- mxPath( from=c("intercept","slope"), arrows=2, connect="unique.pairs",
                        free=TRUE, values=c(1,1,1), labels=c("vari","cov","vars") )
```

The third and fourth `mxPath` functions specify the factor loadings. As these are defined to be a constant value of 1 for the intercept factor and the set [0, 1, 2, 3, 4] for the slope factor, these functions have no free parameters.

```
# intercept loadings
intLoads     <- mxPath( from="intercept", to=c("x1","x2","x3","x4","x5"), arrows=1,
                        free=FALSE, values=c(1,1,1,1,1) )
# slope loadings
sloLoads     <- mxPath( from="slope", to=c("x1","x2","x3","x4","x5"), arrows=1,
                        free=FALSE, values=c(0,1,2,3,4) )
```

The last two `mxPath` functions specify the means. The manifest variables are not regressed on the constant, and thus have intercepts of zero. The observed means are entirely functions of the means of the intercept and slope. To specify this, the manifest variables are regressed on the constant (denoted `"one"`) with a fixed value of zero, and the regressions of the latent variables on the constant are estimated as free parameters.

```
# manifest means
manMeans     <- mxPath( from="one", to=c("x1","x2","x3","x4","x5"), arrows=1,
                        free=FALSE, values=c(0,0,0,0,0) )
# latent means
latMeans     <- mxPath( from="one", to=c("intercept", "slope"), arrows=1,
                        free=TRUE, values=c(1,1), labels=c("meani","means") )
```

The model is now ready to run using the `mxRun` function, and the output of the model can be accessed from the `output` slot of the resulting model. A summary of the output can be reached using `summary()`.

```
growthCurveFit <- mxRun(growthCurveModel)

summary(growthCurveFit)
```

These models may also be specified using matrices instead of paths. See *Time Series, Matrix Specification* for matrix specification of these models.

2.4 Multiple Groups, Path Specification

An important aspect of structural equation modeling is the use of multiple groups to compare means and covariances structures between any two (or more) data groups, for example males and females, different ethnic groups, ages etc. Other examples include groups which have different expected covariances matrices as a function of parameters in the model, and need to be evaluated together for the parameters to be identified.

The example includes the heterogeneity model as well as its submodel, the homogeneity model, and is available in the following file:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/BivariateHeterogeneity_PathRaw.R

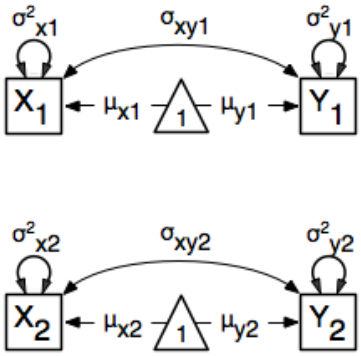
A parallel version of this example, using matrix specification of models rather than paths, can be found here:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/BivariateHeterogeneity_MatrixRaw.R

2.4.1 Heterogeneity Model

We will start with a basic example here, building on modeling means and variances in a saturated model. Assume we have two groups and we want to test whether they have the same mean and covariance structure.

The path diagram of the heterogeneity model for a set of variables x and y are shown below.



Data

For this example we simulated two datasets (*xy1* and *xy2*) each with zero means and unit variances, one with a correlation of 0.5, and the other with a correlation of 0.4 with 1000 subjects each. We use the `mvrnorm` function in the `MASS` package, which takes three arguments: `Sample Size`, `Means`, `Covariance Matrix`. We check the means and covariance matrix in R and provide `dimnames` for the dataframe. See attached R code for simulation and data summary.

```
#Simulate Data
require(MASS)
#group 1
set.seed(200)
xy1 <- mvrnorm (1000, c(0,0), matrix(c(1,.5,.5,1),2,2))
#group 2
set.seed(200)
xy2 <- mvrnorm (1000, c(0,0), matrix(c(1,.4,.4,1),2,2))

#Print Descriptive Statistics
selVars <- c('X','Y')
summary(xy1)
```

```
cov(xy1)
dimnames(xy1) <- list(NULL, selVars)
summary(xy2)
cov(xy2)
dimnames(xy2) <- list(NULL, selVars)
```

Model Specification

As before, we include the OpenMx package using a `require` statement. We first fit a heterogeneity model, allowing differences in both the mean and covariance structure of the two groups. As we are interested whether the two structures can be equated, we have to specify the models for the two groups, named `group1` and `group2` within another model, named `bivHet`. The structure of the job thus looks as follows, with two `mxModel` commands as arguments of another `mxModel` command. Note that `mxModel` commands are unlimited in the number of arguments.

```
require(OpenMx)

bivHetModel <- mxModel("bivHet",
    mxModel("group1"),
    mxModel("group2"),
    mxFitFunctionMultigroup(c("group1.fitfunction", "group2.fitfunction")) )
```

For each of the groups, we fit a saturated model, by specifying paths with free parameters for the variances and the covariance using two-headed arrows to generate the expected covariance matrix. Single-headed arrows from the constant `one` to the manifest variables contain the free parameters for the expected means. Note that we have specified different labels for all the free elements, in the two `mxModel` statements. The type is RAM by default.

```
dataRaw1     <- mxData( observed=xy1, type="raw")
variances1   <- mxPath( from=selVars, arrows=2,
                        free=T, values=1, lbound=.01, labels=c("vX1","vY1") )
covariance1  <- mxPath( from="X", to="Y", arrows=2,
                        free=T, values=.2, lbound=.01, labels="cXY1")
means1       <- mxPath( from="one", to=selVars, arrows=1,
                        free=T, values=c(0.1,-0.1), ubound=c(NA,0), lbound=c(0,NA),
                        labels=c("mX1","mY1") )
model1       <- mxModel("group1", type="RAM", manifestVars=selVars,
                         dataRaw1, variances1, covariance1, means1)

dataRaw2     <- mxData( observed=xy2, type="raw")
variances2   <- mxPath( from=selVars, arrows=2,
                        free=T, values=1, lbound=.01, labels=c("vX2","vY2") )
covariance2  <- mxPath( from="X", to="Y", arrows=2,
                        free=T, values=.2, lbound=.01, labels="cXY2")
means2       <- mxPath( from="one", to=selVars, arrows=1,
                        free=T, values=c(0.1,-0.1), ubound=c(NA,0), lbound=c(0,NA),
                        labels=c("mX2","mY2") )
model2       <- mxModel("group2", type="RAM", manifestVars=selVars,
                         dataRaw2, variances2, covariance2, means2)
```

We estimate five parameters (two means, two variances, one covariance) per group for a total of 10 free parameters. We cut the `Labels matrix:` parts from the output generated with `bivHetModel$group1$matrices` and `bivHetModel$group2$matrices`.

```
in group1
    $S
          X      Y
    X   "vX1" "cXY1"
    Y  "cXY1"  "vY1"
```

```
in group2
    $S
          X      Y
    X   "vX2" "cXY2"
    Y  "cXY2"  "vY2"
```

```
$M                                  $M
     X       Y                           X       Y
[1,] "mX1" "mY1"                    [1,] "mX2" "mY2"
```

To evaluate both models together, we use an `mxFitFunctionMultigroup` command that adds up the values of the fit functions of the two groups.

```
fun           <- mxFitFunctionMultigroup(c("group1.fitfunction", "group2.fitfunction"))

bivHetModel   <- mxModel("bivariate Heterogeneity Path Specification",
                         model1, model2, fun )
```

Model Fitting

The `mxRun` command is required to actually evaluate the model. Note that we have adopted the following notation of the objects. The result of the `mxModel` command ends in `Model`; the result of the `mxRun` command ends in `Fit`. Of course, these are just suggested naming conventions.

```
bivHetFit <- mxRun(bivHetModel)
```

A variety of output can be printed. We chose here to print the expected means and covariance matrices, which the RAM objective function generates based on the path specification, respectively in the matrices **M** and **S** for the two groups. OpenMx also puts the values for the expected means and covariances in the `means` and `covariance` objects. We also print the likelihood of the data given the model.

```
EM1Het <- bivHetFit$group1.fitfunction$info$expMean
EM2Het <- bivHetFit$group2.fitfunction$info$expMean
EC1Het <- bivHetFit$group1.fitfunction$info$expCov
EC2Het <- bivHetFit$group2.fitfunction$info$expCov
LLHet <- bivHetFit$output$fit
```

2.4.2 Homogeneity Model: a Submodel

Next, we fit a model in which the mean and covariance structure of the two groups are equated to one another, to test whether there are significant differences between the groups. As this model is nested within the previous one, the data are the same.

Model Specification

Rather than having to specify the entire model again, we copy the previous model `bivHetModel` into a new model `bivHomModel` to represent homogeneous structures.

```
#Fit Homogeneity Model
bivHomModel <- bivHetModel
```

As the free parameters of the paths are translated into RAM matrices, and matrix elements can be equated by assigning the same label, we now have to equate the labels of the free parameters in group1 to the labels of the corresponding elements in group2. This can be done by referring to the relevant matrices using the `ModelName$MatrixName` syntax, followed by `$labels`. Note that in the same way, one can refer to other arguments of the objects in the model. Here we assign the labels from group1 to the labels of group2, separately for the 'covariance' matrices (in **S**) used for the expected covariance matrices and the 'means' matrices (in **M**) for the expected mean vectors.

```
bivHomModel$group2.S$labels <- bivHomModel$group1.S$labels
bivHomModel$group2.M$labels <- bivHomModel$group1.M$labels
```

The specification for the submodel is reflected in the names of the labels which are now equal for the corresponding elements of the mean and covariance matrices, as below.

```
in group1
    $S
           X      Y
    X   "vX1" "cXY1"
    Y  "cXY1"  "vY1"

    $M
           X      Y
    [1,] "mX1" "mY1"
```

```
in group2
    $S
           X      Y
    X   "vX1" "cXY1"
    Y  "cXY1"  "vY1"

    $M
           X      Y
    [1,] "mX1" "mY1"
```

Model Fitting

We can produce similar output for the submodel, i.e. expected means and covariances and likelihood, the only difference in the code being the model name. Note that as a result of equating the labels, the expected means and covariances of the two groups should be the same, and a total of 5 parameters is estimated.

```
bivHomFit <- mxRun(bivHomModel)
EM1Hom <- bivHomFit$group1.fitfunction$info$expMean
EM2Hom <- bivHomFit$group2.fitfunction$info$expMean
EC1Hom <- bivHomFit$group1.fitfunction$info$expCov
EC2Hom <- bivHomFit$group2.fitfunction$info$expCov
LLHom <- bivHomFit$output$fit
```

Finally, to evaluate which model fits the data best, we generate a likelihood ratio test from the difference between -2 times the log-likelihood of the homogeneity model and -2 times the log-likelihood of the heterogeneity model. This statistic is asymptotically distributed as a Chi-square, which can be interpreted with the difference in degrees of freedom of the two models, in this case 5 df.

```
Chi <- LLHom-LLHet
LRT <- rbind(LLHet,LLHom,Chi)
LRT
```

These models may also be specified using matrices instead of paths. See *Multiple Groups, Matrix Specification* for matrix specification of these models.

2.5 Genetic Epidemiology, Path Specification

Mx was and OpenMx is probably the most popular statistical modeling package in the behavior genetics field, as it was conceived with genetic models in mind, which rely heavily on multiple groups. We introduce here an OpenMx script for the basic genetic model in genetic epidemiologic research, the ACE model. This model assumes that the variability in a phenotype, or observed variable, can be explained by differences in genetic and environmental factors, with **A** representing additive genetic factors, **C** shared/common environmental factors and **E** unique/specific environmental factors (see Neale & Cardon 1992, for a detailed treatment). To estimate these three sources of variance, data have to be collected on relatives with different levels of genetic and environmental similarity to provide sufficient information to identify the parameters. One such design is the classical twin study, which compares the similarity of identical (monozygotic, MZ) and fraternal (dizygotic, DZ) twins to infer the role of **A**, **C** and **E**.

The example starts with the ACE model and includes one submodel, the AE model. It is available in the following file:

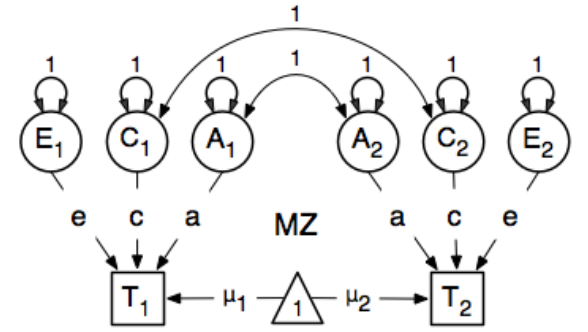
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/UnivariateTwinAnalysis_PathRaw.R

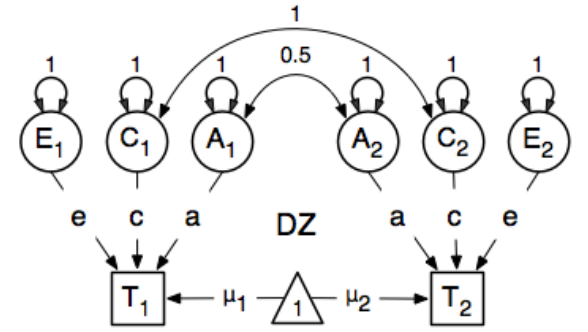
A parallel version of this example, using matrix specification of models rather than paths, can be found here:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/UnivariateTwinAnalysis_MatrixRaw.R

2.5.1 ACE Model: a Twin Analysis

A twin analysis is a typical example of multiple groups, in this case MZ twins and DZ twins, with different expectations for the covariance structure (and possibly means). We illustrate the model here with the corresponding two path diagrams:





Data

Let us assume you have collected data on a large sample of twin pairs for your phenotype of interest. For illustration purposes, we use Australian data on body mass index (BMI) which are saved in a text file *twinData*, which comes with the OpenMx package. We use R to read the data into a data.frame and define the objects `selVars` for the variables selected for analysis, and `aceVars` for the latent variables to simplify the OpenMx code. We then create two subsets of the data for MZ females (mzData) and DZ females (dzData) respectively with the code below, and generate some descriptive statistics, namely means and covariances.

```
# Load Data
data(twinData)

# Select Variables for Analysis
selVars   <- c('bmi1','bmi2')
aceVars   <- c("A1","C1","E1","A2","C2","E2")

# Select Data for Analysis
mzData    <- subset(twinData, zyg==1, selVars)
```

```
dzData    <- subset(twinData, zyg==3, selVars)

# Generate Descriptive Statistics
colMeans(mzData,na.rm=TRUE)
colMeans(dzData,na.rm=TRUE)
cov(mzData,use="complete")
cov(dzData,use="complete")
```

Model Specification

There are different ways to draw a path diagram of the ACE model. The most commonly used approach is with the three latent variables in circles at the top, separately for twin 1 and twin 2 respectively called **A1**, **C1**, **E1** and **A2**, **C2**, **E2**. The latent variables are connected to the observed variables in boxes at the bottom, representing the measures for twin 1 and twin 2: **T1** and **T2**, by single-headed arrows from the latent to the manifest variables. Path coefficients **a**, **c** and **e** are estimated but constrained to be the same for twin 1 and twin 2, as well as for MZ and DZ twins. As MZ twins share all their genotypes, the double-headed path connecting **A1** and **A2** is fixed to one in the MZ diagram. DZ twins share on average half their genes, and as a result the corresponding path is fixed to 0.5 in the DZ diagram. Environmental factors that are shared between twins are assumed to increase similarity between twins to the same extent in MZ and DZ twins (equal environments assumption), thus the double-headed path connecting **C1** and **C2** is fixed to one in both diagrams above. The unique environmental factors are by definition uncorrelated between twins.

Let's go through the paths specification step by step. First, we start with the `require(OpenMx)` statement. We include the full code here. As MZ and DZ have to be evaluated together, the models for each will be arguments of a bigger model. Given the diagrams for the MZ and the DZ group look rather similar, we start by specifying all the common elements which will be stored in a list *paths* and then shared with the two submodels for each of the twin types, defined in separate `mxModel` commands. The latter two `MxModel` objects (*modelMZ* and *modelDZ*) are arguments of the overall model, and will be saved together in the R object *modelACE* and thus be run together.

```
require(OpenMx)

# Path objects for Multiple Groups
manifestVars=selVars
latentVars=aceVars
# variances of latent variables
latVariances <- mxPath( from=aceVars, arrows=2,
                        free=FALSE, values=1 )
# means of latent variables
latMeans     <- mxPath( from="one", to=aceVars, arrows=1,
                        free=FALSE, values=0 )
# means of observed variables
obsMeans     <- mxPath( from="one", to=selVars, arrows=1,
                        free=TRUE, values=20, labels="mean" )
# path coefficients for twin 1
pathAceT1    <- mxPath( from=c("A1","C1","E1"), to="bmi1", arrows=1,
                        free=TRUE, values=.5,  label=c("a","c","e") )
# path coefficients for twin 2
pathAceT2    <- mxPath( from=c("A2","C2","E2"), to="bmi2", arrows=1,
                        free=TRUE, values=.5,  label=c("a","c","e") )
# covariance between C1 & C2
covC1C2      <- mxPath( from="C1", to="C2", arrows=2,
                        free=FALSE, values=1 )
# covariance between A1 & A2 in MZ twins
covA1A2_MZ   <- mxPath( from="A1", to="A2", arrows=2,
                        free=FALSE, values=1 )
# covariance between A1 & A2 in DZ twins
covA1A2_DZ   <- mxPath( from="A1", to="A2", arrows=2,
```

```
                        free=FALSE, values=.5 )

# Data objects for Multiple Groups
dataMZ        <- mxData( observed=mzData, type="raw" )
dataDZ        <- mxData( observed=dzData, type="raw" )

# Combine Groups
paths         <- list( latVariances, latMeans, obsMeans,
                       pathAceT1, pathAceT2, covC1C2 )
modelMZ       <- mxModel(model="MZ", type="RAM", manifestVars=selVars,
                         latentVars=aceVars, paths, covA1A2_MZ, dataMZ )
modelDZ       <- mxModel(model="DZ", type="RAM", manifestVars=selVars,
                         latentVars=aceVars, paths, covA1A2_DZ, dataDZ )
minus2ll      <- mxAlgebra( expression=MZ.fitfunction + DZ.fitfunction,
                            name="minus2loglikelihood" )
obj           <- mxFitFunctionAlgebra( "minus2loglikelihood" )
modelACE      <- mxModel(model="ACE", modelMZ, modelDZ, minus2ll, obj )

# Run Model
fitACE        <- mxRun(modelACE)
sumACE        <- summary(fitACE)
```

Now we will discuss the script line by line. For further details on RAM, see [RAM1990]. Each line can be pasted into R, and then evaluated together once the whole model is specified. Models specifying paths are translated into 'RAM' specifications for optimization, indicated by using the `type="RAM"` within the `mxModel` statements. We start the path diagram specification by providing the names for the manifest variables in `manifestVars` and the latent variables in `latentVars`. We use here the `selVars` and `aceVars` objects that we created previously when preparing the data.

..[RAM1990] McArdle, J.J. & Boker, S.M. (1990). RAMpath: Path diagram software. Denver: Data Transforms Inc.

```
manifestVars=selVars
latentVars=aceVars
```

We start by specifying paths for the variances and means of the latent variables. These include double-headed arrows from each latent variable back to itself, fixed at one.

```
# variances of latent variables
latVariances <- mxPath( from=aceVars, arrows=2,
                        free=FALSE, values=1 )
```

and single-headed arrows from the triangle (with a fixed value of one) to each of the latent variables, fixed at zero.

```
# means of latent variables
latMeans      <- mxPath( from="one", to=aceVars, arrows=1,
                        free=FALSE, values=0 )
```

Next we specify paths for the means of the observed variables using single-headed arrows from `one` to each of the manifest variables. These are set to be free and given a start value of 20. As we use the same label ("mean") for the two means, they are constrained to be equal. Remember that R 'recycles'.

```
# means of observed variables
obsMeans      <- mxPath( from="one", to=selVars, arrows=1,
                        free=TRUE, values=20, labels="mean" )
```

The main paths of interest are those from each of the latent variables to the respective observed variable. These are also estimated (thus all are set free), get a start value of 0.5 and appropriate labels. We chose the start value of .5 by

dividing the observed variance, here about .7-.8 in three for the three sources of variance, and then taking the square root as we're estimating the path coefficients, but these are squared to obtain their contribution to the variance.

```
# path coefficients for twin 1
pathAceT1     <- mxPath( from=c("A1","C1","E1"), to="bmi1", arrows=1,
                         free=TRUE, values=.5,  label=c("a","c","e") )
# path coefficients for twin 2
pathAceT2     <- mxPath( from=c("A2","C2","E2"), to="bmi2", arrows=1,
                         free=TRUE, values=.5,  label=c("a","c","e") )
```

As the common environmental factors are by definition the same for both twins, we fix the correlation between **C1** and **C2** to one.

```
# covariance between C1 & C2
covC1C2       <- mxPath( from="C1", to="C2", arrows=2,
                         free=FALSE, values=1 )
```

Next we create the paths that are specific to the MZ group or the DZ group and are later included into the respective models, `modelMZ` and `modelDZ`, which are combined in *modelACE*. In the MZ model we add the path for the correlation between **A1** and **A2** which is fixed to one. In the DZ model the correlation between **A1** and **A2** is fixed to 0.5 instead.

```
# covariance between A1 & A2 in MZ's
covA1A2_MZ    <- mxPath( from="A1", to="A2", arrows=2,
                         free=FALSE, values=1 )
# covariance between A1 & A2 in DZ's
covA1A2_DZ    <- mxPath( from="A1", to="A2", arrows=2,
                         free=FALSE, values=.5 )
```

That concludes the specification of the paths from which the models will be generated for MZ and DZ twins separately. Next we move to the `mxData` commands that call up the data.frame with the MZ raw data, *mzData*, and the DZ raw data, *dzData*, respectively, with the type specified explicitly as `raw`. These are stored in two MxData objects.

```
dataMZ        <- mxData( observed=mzData, type="raw" )
dataDZ        <- mxData( observed=dzData, type="raw" )
```

As we indicated earlier, we're collecting all the mxPaths objects that are in common between the two models in a list called *paths*, which will then be included in the respective models that we'll build next with the `mxModel` statements. First we give the model a name, "MZ", to refer back to it later when we need to add the fit functions. Next we tell OpenMx that we're specifying a path model by using the RAM `type`, which requires us to include both the `manifestVars` and the `latentVars` arguments. Then we include the list of paths generated before that are common between the two models, and the path that is specific to either the MZ or the DZ model. Last we add the data objects for the MZ and DZ group respectively.

```
# Combine Groups
paths         <- list( latVariances, latMeans, obsMeans,
                       pathAceT1, pathAceT2, covC1C2 )
modelMZ       <- mxModel(model="MZ", type="RAM", manifestVars=selVars,
                         latentVars=aceVars, paths, covA1A2_MZ, dataMZ )
modelDZ       <- mxModel(model="DZ", type="RAM", manifestVars=selVars,
                         latentVars=aceVars, paths, covA1A2_DZ, dataDZ )
```

Finally, both models need to be evaluated simultaneously. We generate the sum of the fit functions for the two groups, using `mxAlgebra`, and use the result (*minus2loglikelihood*) as argument of the `mxFitFunctionAlgebra` command. We specify a new `mxModel` - with a new name using the `model=""` notation, which has the *modelMZ* and *modelDZ* as its arguments. We also include the objects summing the likelihood and evaluating it.

```
minus2ll      <- mxAlgebra( expression=MZ.fitfunction + DZ.fitfunction,
                            name="minus2loglikelihood" )
```

```
obj          <- mxFitFunctionAlgebra( "minus2loglikelihood" )
modelACE     <- mxModel(model="ACE", modelMZ, modelDZ, minus2ll, obj )
```

Model Fitting

We need to invoke the `mxRun` command to start the model evaluation and optimization. Detailed output will be available in the resulting object, which can be obtained by a `print()` statement, or a more succinct output can be obtained with the `summary` function.

```
#Run ACE model
fitACE       <- mxRun(modelACE)
sumACE       <- summary(fitACE)
```

Often, however, one is interested in specific parts of the output. In the case of twin modeling, we typically will inspect the likelihood, the expected covariance matrices and mean vectors, the parameter estimates, and possibly some derived quantities, such as the standardized variance components, obtained by dividing each of the components by the total variance. Note in the code below that the `mxEval` command allows easy extraction of the values in the various matrices which form the first argument, with the model name as second argument. Once these values have been put in new objects, we can use any regular R expression to derive further quantities or organize them in a convenient format for including in tables. Note that helper functions could easily (and will likely) be written for standard models to produce 'standard' output.

```
# Generate & Print Output
# additive genetic variance, a^2
A  <- mxEval(a*a, fitACE)
# shared environmental variance, c^2
C  <- mxEval(c*c, fitACE)
# unique environmental variance, e^2
E  <- mxEval(e*e, fitACE)
# total variance
V  <- (A+C+E)
# standardized A
a2 <- A/V
# standardized C
c2 <- C/V
# standardized E
e2 <- E/V
# table of estimates
estACE <- rbind(cbind(A,C,E),cbind(a2,c2,e2))
# likelihood of ACE model
LL_ACE <- mxEval(fitfunction, fitACE)
```

2.5.2 Alternative Models: an AE Model

To evaluate the significance of each of the model parameters, nested submodels are fit in which the parameters of interest are fixed to zero. If the likelihood ratio test between the two models (one including the parameter and the other not) is significant, the parameter that is dropped from the model significantly contributes to the variance of the phenotype in question. Here we show how we can fit the AE model as a submodel with a change in the two `mxPath` commands. We re-specify the path from **C1** to **bmi1** to be fixed to zero, and do the same for the path from **C2** to **bmi2**. We need to rebuild both *modelMZ* and *modelDZ*, so that they are now built with the changed paths, as well as the overall model which we now call *modelAE*. We can run this model in the same way as before, by combining the fit functions of the two groups and generate similar summaries of the results.

```
#Run AE model
# path coefficients for twin 1
pathAceT1    <- mxPath( from=c("A1","C1","E1"), to="bmi1", arrows=1,
                        free=c(T,F,T), values=c(.6,0,.6),  label=c("a","c","e") )
# path coefficients for twin 2
pathAceT2    <- mxPath( from=c("A2","C2","E2"), to="bmi2", arrows=1,
                        free=c(T,F,T), values=c(.6,0,.6),  label=c("a","c","e") )

# Combine Groups
paths        <- list( latVariances, latMeans, obsMeans,
                        pathAceT1, pathAceT2, covC1C2 )
modelMZ      <- mxModel(model="MZ", type="RAM", manifestVars=selVars,
                        latentVars=aceVars, paths, covA1A2_MZ, dataMZ )
modelDZ      <- mxModel(model="DZ", type="RAM", manifestVars=selVars,
                        latentVars=aceVars, paths, covA1A2_DZ, dataDZ )
modelAE      <- mxModel(model="AE", modelMZ, modelDZ, minus2ll, obj )

# Run Model
fitAE        <- mxRun(modelAE)
sumAE        <- summary(fitAE)

# Generate & Print Output
A  <- mxEval(a*a, fitAE)
C  <- mxEval(c*c, fitAE)
E  <- mxEval(e*e, fitAE)
V  <- (A+C+E)
a2 <- A/V
c2 <- C/V
e2 <- E/V
estAE <- rbind(cbind(A, C, E),cbind(a2, c2, e2))
LL_AE <- mxEval(fitfunction, fitAE)
LRT_ACE_AE <- LL_AE - LL_ACE
estACE
estAE
LRT_ACE_AE
```

We use a likelihood ratio test (or take the difference between -2 times the log-likelihoods of the two models, for the difference in degrees of freedom) to determine the best fitting model. In this example, the Chi-square likelihood ratio test is 0 for 1 degree of freedom, indicating the the *c* parameter does not contribute to the variance at all. This can also be seen in the 0 estimates for the *c* parameter in the ACE model and identical parameters for *a* and *e* in the ACE and AE models.

While the approach outlined above works just fine, the same can be accomplished with the `omxSetParameters` helper function, that allows the user to specify a parameter label in a model whose attributes are to be changed, in this case by setting `free` to FALSE and `values` to 0. Prior to making this change, we copied the original model into a new model and gave it a new name, so that we have separate model objects for the two nested models that can then be compared with `mxCompare`.

```
modelAE    <- mxModel( modelACE, name="AE" )
modelAE    <- omxSetParameters( modelAE, labels="c", free=FALSE, values=0 )
fitAE      <- mxRun(modelAE)
sumAE      <- summary(fitAE)
mxCompare(fitACE, fitAE)
```

See *Genetic Epidemiology, Matrix Specification* for matrix specification of these models.

2.6 Definition Variables, Path Specification

This example will demonstrate the use of OpenMx definition variables with the analysis of a simple two group dataset. What are definition variables? Essentially, definition variables can be thought of as observed variables that are used to change the statistical model on an individual case basis. In essence, it is as though one or more variables in the raw data vectors are used to specify the statistical model for that individual. Many different types of statistical model can be specified in this fashion; some are readily specified in standard fashion, and some cannot. To illustrate, we implement a two-group model. The groups differ in their means but not in their variances and covariances. This situation could easily be modeled in a regular multiple group fashion - it is only implemented using definition variables to illustrate their use. The results are verified using summary statistics and an Mx 1.0 script for comparison is also available.

2.6.1 Mean Differences

The example shows the use of definition variables to test for mean differences. It is available in the following file:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/DefinitionMeans_PathRaw.R

A parallel version of this example, using matrix specification of models rather than paths, can be found here:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/DefinitionMeans_MatrixRaw.R

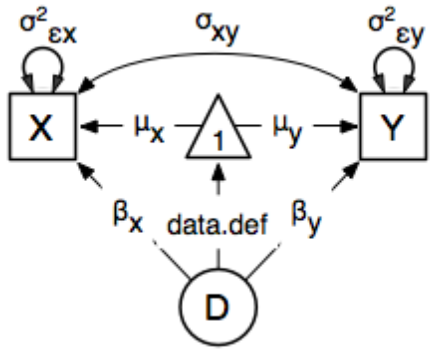
Statistical Model

Algebraically, we are going to fit the following model to the observed *x* and *y* variables:

$$x_i = \mu_x + \beta_x * def + \epsilon_{xi}$$
$$y_i = \mu_y + \beta_y * def + \epsilon_{yi}$$

where def is the definition variable. The residual sources of variance, ϵ_{xi} and ϵ_{yi} covary to the extent σ_{xy}. So, the task is to estimate: the two means μ_x and μ_y; the deviations from these means due to belonging to the group identified by having def set to 1 (as opposed to zero), β_x and β_y; and the parameters of the variance covariance matrix: cov(ϵ_x, ϵ_y).

Our task is to implement the model shown in the figure below:



Data Simulation

Our first step to running this model is to simulate the data to be analyzed. Each individual is measured on two observed variables, *x* and *y*, and a third variable *def* which denotes their group membership with a 1 or a 0. These values for group membership are not accidental, and must be adhered to in order to obtain readily interpretable results. Other values such as 1 and 2 would yield the same model fit, but would make the interpretation more difficult.

```
library(MASS)     # to get hold of mvrnorm function
set.seed(200)     # to make the simulation repeatable
N               <- 500     # sample size, per group
Sigma           <- matrix(c(1,.5,.5,1),2,2)
group1          <- mvrnorm(N, c(1,2), Sigma) # Use mvrnorm from MASS package
group2          <- mvrnorm(N, c(0,0), Sigma)
```

We make use of the superb R function `mvrnorm` in order to simulate N=500 records of data for each group. These observations correlate 0.5 and have a variance of 1, per the matrix *Sigma*. The means of *x* and *y* in group 1 are 1.0 and 2.0, respectively; those in group 2 are both zero. The output of the `mvrnorm` function calls are matrices with 500 rows and 2 columns, which are stored in group 1 and group 2. Now we create the definition variable

```
# Put the two groups together, create a definition variable,
# and make a list of which variables are to be analyzed (selVars)
xy              <- rbind(group1,group2)       # Bind groups together by rows
dimnames(xy)[2]<- list(c("x","y"))            # Add names
def             <- rep(c(1,0),each=N);        # Add def var [2n] for group status
selVars         <- c("x","y")                 # Make selection variables object
```

The objects *xy* and *def* might be combined in a data frame. However, in this case we won't bother to do it externally, and simply paste them together in the `mxData` function call.

Model Specification

The following code contains all of the components of our model. Before specifying a model, the OpenMx library must be loaded into R using either the `require()` or `library()` function. This code uses the `mxModel` function to create an MxModel object, which we'll then run. Note that all the objects required for estimation (data, matrices, an expectation function, and a fit function) are declared within the `mxModel` function. This type of code structure is recommended for OpenMx scripts generally.

```
dataRaw      <- mxData( observed=data.frame(xy,def), type="raw" )
# variances
variances    <- mxPath( from=c("x","y"), arrows=2,
                        free=TRUE, values=1, labels=c("Varx","Vary") )
# covariances
covariances  <- mxPath( from="x", to="y", arrows=2,
                        free=TRUE, values=.1, labels=c("Covxy") )
# means
means        <- mxPath( from="one", to=c("x","y"), arrows=1,
                        free=TRUE, values=1, labels=c("meanx","meany") )
# definition value
defValues    <- mxPath( from="one", to="DefDummy", arrows=1,
                        free=FALSE, values=1, labels="data.def" )
# beta weights
betaWeights  <- mxPath( from="DefDummy", to=c("x","y"), arrows=1,
                        free=TRUE, values=1, labels=c("beta_1","beta_2") )

defMeansModel <- mxModel("Definition Means Path Specification", type="RAM",
                         manifestVars=selVars, latentVars="DefDummy",
                         dataRaw, variances, covariances, means,
                         defValues, betaWeights)
```

The first argument in an `mxModel` function has a special function. If an object or variable containing an `MxModel` object is placed here, then `mxModel` adds to or removes pieces from that model. If a character string (as indicated by double quotes) is placed first, then that becomes the name of the model. Models may also be named by including a `name` argument. This model is named `"Definition Means Path Specification"`. The second argument of the `mxModel` function call declares that we are going to be using RAM specification of the model, using directional

and bidirectional path coefficients between the variables. Model specification is carried out using two lists of variables, `manifestVars` and `latentVars`.

```
manifestVars=selVars
latentVars="DefDummy"
```

Next, we declare where the data are, and their type, by creating an MxData object with the `mxData` function. This code first references the object where our data are, then uses the `type` argument to specify that this is raw data. Analyses using definition variables have to use raw data, so that the model can be specified on an individual data vector level.

```
dataRaw      <- mxData( observed=data.frame(xy,def), type="raw" )
```

Then `mxPath` functions are used to specify paths between the manifest and latent variables. In the present case, we need four mxPath commands to specify the model. The first is for the variances of the *x* and *y* variables, and the second specifies their covariance. The third specifies a path from the mean vector, always known by the special keyword `one`, to each of the observed variables, and to the single latent variable `DefDummy`. This last path is specified to contain the definition variable, by virtue of the `data.def` label. Definition variables are part of the data so the first part is always `data.`. The second part refers to the actual variable in the dataset whose values are modeled. Finally, two paths are specified from the `DefDummy` latent variable to the observed variables. These parameters estimate the deviation of the mean of those with a `data.def` value of 1 versus those with `data.def` values of zero.

```
# variances
variances    <- mxPath( from=c("x","y"), arrows=2,
                        free=TRUE, values=1, labels=c("Varx","Vary") )
# covariances
covariances  <- mxPath( from="x", to="y", arrows=2,
                        free=TRUE, values=.1, labels=c("Covxy") )
# means
means        <- mxPath( from="one", to=c("x","y"), arrows=1,
                        free=TRUE, values=1, labels=c("meanx","meany") )
# definition value
defValues    <- mxPath( from="one", to="DefDummy", arrows=1,
                        free=FALSE, values=1, labels="data.def" )
# beta weights
betaWeights  <- mxPath( from="DefDummy", to=c("x","y"), arrows=1,
                        free=TRUE, values=1, labels=c("beta_1","beta_2") )
```

We can then run the model and examine the output with a few simple commands.

Model Fitting

```
# Run the model
defMeansFit<-mxRun(defMeansModel)

defMeansFit$matrices
```

The R object *defmeansFit* contains matrices and algebras; here we are interested in the matrices, which can be seen with the `defmeansFit$matrices` entry. In path notation, the unidirectional, one-headed arrows appear in the matrix **A**, the two-headed arrows in **S**, and the mean vector single headed arrows in **M**.

```
# Compare OpenMx estimates to summary statistics from raw data,
# remembering to knock off 1 and 2 from group 1's data
# so as to estimate variance of combined sample without
# the mean difference contributing to the variance estimate.

# First compute some summary statistics from data
```

```
ObsCovs <- cov(rbind(group1 - rep(c(1,2), each=N), group2))
ObsMeansGroup1 <- c(mean(group1[,1]), mean(group1[,2]))
ObsMeansGroup2 <- c(mean(group2[,1]), mean(group2[,2]))

# Second extract parameter estimates and matrix algebra results from model
Sigma <- mxEval(S[1:2,1:2], defMeansFit)
Mu <- mxEval(M[1:2], defMeansFit)
beta <- mxEval(A[1:2,3], defMeansFit)

# Third, check to see if things are more or less equal
omxCheckCloseEnough(ObsCovs,Sigma,.01)
omxCheckCloseEnough(ObsMeansGroup1,as.vector(Mu+beta),.001)
omxCheckCloseEnough(ObsMeansGroup2,as.vector(Mu),.001)
```

These models may also be specified using matrices instead of paths. See *Definition Variables, Matrix Specification* for matrix specification of these models.

2.7 Ordinal and Joint Ordinal-Continuous Model Specification

This chapter deals with the specification of models that are either fit exclusively to ordinal variables or to a mix of ordinal and continuous variables. It extends the continuous data common factor model found in previous chapters to ordinal data.

The examples for this chapter can be found in the following file:

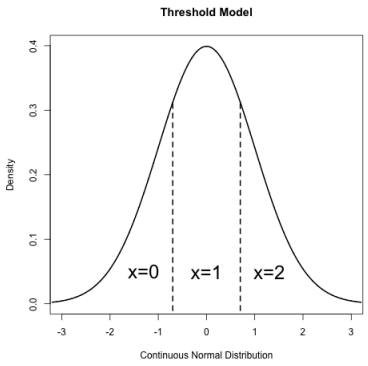
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/OneFactorOrdinal_PathRaw.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/OneFactorJoint_PathRaw.R

The continuous version of this model for raw data can be found the previous demos here:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/OneFactorModel_PathRaw.R

2.7.1 Ordinal Data Modeling

OpenMx models ordinal data under a threshold model. A continuous normal distribution is assumed to underly every ordinal variable. These latent continuous distributions are only observed as being above or below a threshold, where there is one fewer threshold than observed categories in the data. For example, consider a variable with three ordered categories indicated by the values zero, one and two. Under this approach, this variable is assumed to follow a normal distribution that is partitioned or cut by two thresholds: individuals with underlying scores below the first threshold have an observed value of zero, individuals with latent scores between the thresholds are observed with values of one, and individuals with underlying scores give observed values of two.



Each threshold may be freely estimated or assigned as a fixed parameter, depending on the desired model. In addition to the thresholds, ordinal variables still have a mean and variance that describes the parameters of the underlying continuous distribution. However, this underlying distribution must be scaled by fixing at least two parameters to identify the model. One method of identification fixes the mean and variance to specific values, most commonly to a standard normal distribution with a mean of zero and a variance of one. A variation on this method fixes the residual variance of the categorical variable to one, which is often easier to specify. Alternatively, categorical variables may be identified by fixing two thresholds to non-equivalent constant values. These methods will differ in the scale assigned to the ordinal variables (and thus, the scale of the parameters estimated from them), but all identify the same model and should provide equally valid results.

OpenMx allows for the inclusion of continuous and ordinal variables in the same model, as well as models with only continuous or only ordinal variables. Any number of continuous variables may be included in an OpenMx model; however, maximum likelihood estimation for ordinal data must be limited to twenty ordinal variables regardless of the number of continuous variables. Further technical details on ordinal and joint continuous-ordinal optimization are contained at the end of this chapter.

Data Specification

To use ordinal variables in OpenMx, users must identify ordinal variables by specifying those variables as ordered factors in the included data. Ordinal models can only be fit to raw data; if data is described as a covariance or other moment matrix, then the categorical nature of the data was already modeled to generate that moment matrix. Ordinal variables must be defined as specific columns in an R data frame.

Factors are a type of variable included in an R data frame. Unlike numeric or continuous variables, which must include only numeric and missing values, observed values for factors are treated as character strings. All factors contain a `levels` argument, which lists the possible values for a factor. Ordered factors contain information about the ordering of possible levels. Both R and OpenMx have tools for manipulating factors in data frames. The R functions `factor()` and `as.factor()` (and companions `ordered()` and `as.ordered()`) can be used to specify ordered factors. OpenMx includes a helper function `mxFactor()` which more directly prepares ordinal variables as ordered factors in preparation for inclusion in OpenMx models. The code below demonstrates the `mxFactor()` function, replacing the variable *z1* that was initially read as a continuous variable and treating it as an ordinal variable with two levels. This process is repeated for *z2* (two levels) and *z3* (three levels).

```
data(myFADataRaw)

oneFactorOrd <- myFADataRaw[,c("z1", "z2", "z3")]

oneFactorOrd$z1 <- mxFactor(oneFactorOrd$z1, levels=c(0, 1))
oneFactorOrd$z2 <- mxFactor(oneFactorOrd$z2, levels=c(0, 1))
oneFactorOrd$z3 <- mxFactor(oneFactorOrd$z3, levels=c(0, 1, 2))
```

Threshold Specification

Just as covariances and means are included in models using `mxPath` when `type='RAM'` is enabled, thresholds may be included in models using the `mxThreshold` function. This function creates a list of thresholds to be added to your model, just as `mxPath` creates a list of paths. As an example, the data prep example above includes two binary variables (*z1* and *z2*) and one variable with three categories (*z3*). This means that models fit to this data should contain thresholds for three variables (for *z1*, *z2* and *z3*). This can be done with three separate calls to the `mxThreshold` function, as shown here.

```
mxThreshold(vars="z1", nThresh=1, free=TRUE, values=-1)
mxThreshold(vars="z2", nThresh=1, free=TRUE, values=0)
mxThreshold(vars="z3", nThresh=2, free=TRUE, values=c(-.5, 1.2))
```

The `mxThreshold` function first requires a variable to assign thresholds to, as well as a number of thresholds. In the first use of `mxThreshold` above, those are specified using the `vars` and `nThresh` arguments. The remaining arguments match those used by `mxPath`: threshold parameters should be designated as `free`, be given starting `values`, and optionally given `labels` and boundaries (`lbound` and `ubound`).

In this example, variables 'z1' and 'z2' are binary, with a single freely estimated threshold for each variable with starting values of -1 and 0, respectively. The meaning of these thresholds will depend on the mean and variance of these variables; as we are freely estimating thresholds for binary variables, the mean and variances of these variables should be constrained to fixed values. The third function call represents variable 'z3', which contains two thresholds and thus three categories. These two thresholds are assigned free parameters with staring values of -0.5 and 1.2, and the mean and variance of this variable should also be constrained to fixed values for identification. For variables with multiple thresholds, starting values should be monotonically increasing in each column such that the first column represents the first threshold and lowest value and the last column represents the last threshold and highest value.

Alternatively, `mxThreshold` can be used to specify thresholds for multiple variables at once. In the code below, `mxThreshold` is used to specify thresholds for all variables simultaneously. First, the `vars` argument contains a vector of variable names for which thresholds should be specified. The `nThresh` argument then specifies how many thresholds should be assigned to each variable: 1 each for *z1* and *z2*, and two for *z3*. The `free` argument states that all specified thresholds are to be freely estimated (the one value is repeated for all four thresholds). Finally, starting values are given using the `values` argument: -1 for *z1*, 0 for *z2*, and -.5 and 1.2 for *z3*.

```
mxThreshold(vars=c("z1","z2","z3"), nThresh=c(1,1,2), free=TRUE, values=c(-1,0,-.5,1.2) )
```

There are a few common errors regarding the use of thresholds in OpenMx. First, threshold values within each variable must be strictly increasing, such that the value in any element of the threshold matrix must be greater than all values above it in that column. In the above example, the second threshold for *z3* is set at 1.2, above the value of -.5 for the first threshold. OpenMx will return an error when your thresholds are not strictly increasing. There are no restrictions on values across variables: the second threshold for *z3* could be below all thresholds for *z1* and *z2* provided it exceeded the value for the first *z3* threshold. Second, the variables in your model that are assigned thresholds must match ordinal factors in the data. Additionally, free parameters should only be included for thresholds present in your data: including a second freely estimated threshold for *z1* or *z2* in this example would not directly impede model estimation, but would remain at its starting value and count as a free parameter for the purposes of calculating fit statistics.

It is also important to remember that specifying thresholds is not sufficient to get an ordinal data model to run. In addition, the scale of each ordinal variable must be identified just like the scale of a latent variable. The most common method for this involves constraining a ordinal item's mean to zero and either its total or residual variance to a constant value (i.e., one). For variables with two or more thresholds, ordinal variables may also be identified by constraining two thresholds to fixed values. Models that don't identify the scale of their ordinal variables should not converge.

Thresholds may also be expressed in matrix form. This is described in more detail in the matrix version of this chapter.

Users of original or ''classic" Mx may recall specifying thresholds not in absolute terms, but as deviations. This method estimated the difference between each threshold for a variable and the previous one, which ensured that thresholds were in the correct order (i.e., that the second threshold for a variable was not lower than the first). Users should still employ this method using `mxAlgebra` for more complex models, as during optimization, thresholds may otherwise get out of proper order, causing optimization to stop.

Including Thresholds in Models

If you use `mxThreshold` to specify thresholds, there is nothing left to do prior to running your model. However, if you manually create a threshold matrix, you must also specify the name of this matrix in your expectation function. This is described in more detail in the matrix version of this chapter.

2.7.2 Common Factor Model

All of the raw data examples through the documentation may be converted to ordinal examples by the inclusion of ordinal data, the specification of a threshold matrix and inclusion of that threshold matrix in the objective function.

Ordinal Data

The following example is a version of the continuous data common factor model referenced at the beginning of this chapter. Aside from replacing the continuous variables `x1-x6` with the ordinal variables `z1-z3`, the code below simply incorporates the steps referenced above into the existing example. Data preparation occurs first, with the added `mxFactor` statements to identify ordinal variables and their ordered levels.

```
require(OpenMx)

data(myFADataRaw)
oneFactorOrd <- myFADataRaw[,c("z1","z2","z3")]

oneFactorOrd$z1 <- mxFactor(oneFactorOrd$z1, levels=c(0,1))
oneFactorOrd$z2 <- mxFactor(oneFactorOrd$z2, levels=c(0,1))
oneFactorOrd$z3 <- mxFactor(oneFactorOrd$z3, levels=c(0,1,2))
```

Model specification can be achieved by appending the above threshold matrix and objective function to either the path or matrix common factor examples. The path example below has been altered by changing the variable names from `x1-x6` to `z1-z3`, adding the threshold matrix and objective function, and identifying the ordinal variables by constraining their means to be zero and their residual variances to be one.

```
dataRaw      <- mxData( observed=oneFactorOrd, type="raw" )
# residual variances
resVars      <- mxPath( from=c("z1","z2","z3"), arrows=2,
                        free=FALSE, values=c(1,1,1), labels=c("e1","e2","e3") )
# latent variance
latVar       <- mxPath( from="F1", arrows=2,
                        free=TRUE, values=1, labels ="varF1" )
# factor loadings
facLoads     <- mxPath( from="F1", to=c("z1","z2","z3"), arrows=1,
                        free=c(FALSE,TRUE,TRUE), values=1, labels=c("l1","l2","l3") )
# means
means        <- mxPath( from="one", to=c("z1","z2","z3","F1"), arrows=1,
                        free=FALSE, values=0,
                        labels=c("meanz1","meanz2","meanz3","meanF") )
# thresholds
thresholds   <- mxThreshold( vars=c("z1","z2","z3"), nThresh=c(1,1,2),
                        free=TRUE, values=c(-1,0,-.5,1.2) )
oneFactorModel <- mxModel("Common Factor Model Path Specification", type="RAM",
                        manifestVars=c("z1","z2","z3"), latentVars="F1",
                        dataRaw, resVars, latVar, facLoads, means, thresholds)
```

This model may then be optimized using the `mxRun` command.

```
oneFactorResults <- mxRun(oneFactorModel)
```

Joint Ordinal-Continuous Data

Models with both continuous and ordinal variables may be specified just like any other ordinal data model. Threshold matrices in these models should contain columns only for the ordinal variables, and should contain column names to

designate which variables are to be treated as ordinal. In the example below, the one factor model above is estimated with three continuous variables (`x1-x3`) and three ordinal variables (`z1-z3`).

```
require(OpenMx)

oneFactorJoint <- myFADataRaw[,c("x1","x2","x3","z1","z2","z3")]

oneFactorJoint$z1 <- mxFactor(oneFactorOrd$z1, levels=c(0,1))
oneFactorJoint$z2 <- mxFactor(oneFactorOrd$z2, levels=c(0,1))
oneFactorJoint$z3 <- mxFactor(oneFactorOrd$z3, levels=c(0,1,2))

dataRaw      <- mxData( observed=oneFactorJoint, type="raw" )
# residual variances
resVars      <- mxPath( from=c("x1","x2","x3","z1","z2","z3"), arrows=2,
                        free=c(TRUE, TRUE, TRUE, FALSE, FALSE, FALSE),
                        values=1, labels=c("e1","e2","e3","e4","e5","e6") )
# latent variance
latVar       <- mxPath( from="F1", arrows=2,
                        free=FALSE, values=1, labels ="varF1" )
# factor loadings
facLoads     <- mxPath( from="F1", to=c("x1","x2","x3","z1","z2","z3"), arrows=1,
                        free=TRUE, values=1, labels=c("l1","l2","l3","l4","l5","l6") )
# means
means        <- mxPath( from="one", to=c("x1","x2","x3","z1","z2","z3","F1"), arrows=1,
                        free=c(TRUE,TRUE,TRUE,FALSE,FALSE,FALSE,FALSE), values=0,
                        labels=c("meanx1","meanx2","meanx3",
                                 "meanz1","meanz2","meanz3","meanF") )
# thresholds
thresholds   <- mxThreshold(vars=c("z1","z2","z3"), nThresh=c(1,1,2),
                        free=TRUE, values=c(-1,0,-.5,1.2) )
oneFactorJointModel <- mxModel("Common Factor Model Path Specification", type="RAM",
                        manifestVars=c("x1","x2","x3","z1","z2","z3"), latentVars="F1",
                        dataRaw, resVars, latVar, facLoads, means, thresholds)
```

This model may then be optimized using the `mxRun` command.

```
oneFactorJointResults <- mxRun(oneFactorJointModel)
```

2.7.3 Technical Details

Maximum likelihood estimation for ordinal variables is done by generating expected covariance and mean matrices for the latent continuous variables underlying the set of ordinal variables, then integrating the multivariate normal distribution defined by those covariances and means. The likelihood for each row of the data is defined as the multivariate integral of the expected distribution over the interval defined by the thresholds bordering that row's data. OpenMx uses Alan Genz's SADMVN routine for multivariate normal integration (see http://www.math.wsu.edu/faculty/genz/software/software.html for more information).

When continuous variables are present, OpenMx utilizes a block decomposition to separate the continuous and ordinal covariance matrices for FIML. The likelihood of the continuous variables is calculated normally. The effects of the point estimates of the continuous variables is projected out of the expected covariance matrix of the ordinal data. The likelihood of the ordinal data is defined as the multivariate integral over the distribution defined by the resulting ordinal covariance matrix.

2.8 Growth Mixture Modeling, Path Specification

This example will demonstrate how to specify a growth mixture model using path specification. Unlike other examples, this application will not be demonstrated with covariance data, as this model can only be fit to raw data. The script for this example can be found in the following file:

** http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/GrowthMixtureModel_PathRaw.R

A parallel example using matrix specification can be found here:

** http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/GrowthMixtureModel_MatrixRaw.R

The latent growth curve used in this example is the same one fit in the latent growth curve example. Path and matrix versions of that example for raw data can be found here:

** http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/LatentGrowthCurveModel_PathRaw.R

** http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/LatentGrowthCurveModel_MatrixRaw.R

2.8.1 Mixture Modeling

Mixture modeling is an approach where data are assumed to be governed by some type of mixture distribution. This includes a large class of models, including many varieties of mixture modeling, latent class analysis and related models with binary or categorical latent variables. This example will demonstrate a growth mixture model, where change over time is modeled with a linear growth curve and the distribution of latent intercepts and slopes is governed by a mixture of two distributions. The model can thus be described as a combination of two growth curves, weighted by a class proportion variable, as shown below.

$$x_{ij} = p_1(Intercept_{i1} + \lambda_1 Slope_{i1} + \epsilon) + p_2(Intercept_{i2} + \lambda_2 Slope_{i2} + \epsilon)$$

To scale the class proportion variable as a probability, it must be scaled such that it is strictly positive and the set of all class probabilities sum to a value of one.

$$\sum_{i=1}^{k} p_i = 1$$

Data

The data for this example can be found in the data object `myGrowthMixtureData`. These data contain five time ordered variables named `x1` through `x5`, just like the growth curve demo mentioned previously. It is important to note that raw data is required for mixture modeling, as moment matrices do not contain all of the information required to estimate the model.

```
data(myGrowthMixtureData)
names(myGrowthMixtureData)
```

Model Specification

Specifying a mixture model can be categorized into two general phases. The first phase of model specification pertains to creating the models for each class. The second phase specifies the way those classes are mixed. In OpenMx, this is done using a model tree. Each class is created as a separate `MxModel` object, and those class-specific models are all placed into a larger or parent model. The parent model contains the class proportion parameter(s) and the data.

Creating the class-specific models is done the same way as every other model. We'll begin by specifying the model for the first class using the `mxPath` function. The code below specifies a five-occasion linear growth curve, virtually

identical to the one in the linear growth curve example referenced above. The only changes made to this model are the names of the free parameters; the means, variances and covariance of the intercept and slope terms are now followed by the number 1 to distinguish them from free parameters in the other class.

The fit function for each of the class-specific models must return the likelihoods for each individual rather than the default log likelihood for the entire sample. OpenMx fit functions that handle raw data have the option to return a vector of likelihoods for each row rather than a single likelihood value for the dataset. This option can be accessed as an argument in the fit function `mxFitFunctionML`.

```
# residual variances
resVars      <- mxPath( from=c("x1","x2","x3","x4","x5"), arrows=2,
                        free=TRUE, values = c(1,1,1,1,1),
                        labels=c("residual","residual","residual","residual","residual") )
# latent variances and covariance
latVars      <- mxPath( from=c("intercept","slope"), arrows=2, connect="unique.pairs",
                        free=TRUE, values=c(1,.4,1), labels=c("vari1","cov1","vars1") )
# intercept loadings
intLoads     <- mxPath( from="intercept", to=c("x1","x2","x3","x4","x5"), arrows=1,
                        free=FALSE, values=c(1,1,1,1,1) )
# slope loadings
sloLoads     <- mxPath( from="slope", to=c("x1","x2","x3","x4","x5"), arrows=1,
                        free=FALSE, values=c(0,1,2,3,4) )
# manifest means
manMeans     <- mxPath( from="one", to=c("x1","x2", "x3", "x4","x5"), arrows=1,
                        free=FALSE, values=c(0,0,0,0,0) )
# latent means
latMeans     <- mxPath( from="one", to=c("intercept","slope"), arrows=1,
                        free=TRUE,  values=c(0,-1), labels=c("meani1","means1") )
# enable the likelihood vector
funML        <- mxFitFunctionML(vector=TRUE)
class1       <- mxModel("Class1", type="RAM",
                        manifestVars=c("x1","x2","x3","x4","x5"),
                        latentVars=c("intercept","slope"),
                        resVars, latVars, intLoads, sloLoads, manMeans, latMeans,
                        funML)
```

We could create the model for our second class by copy and pasting the code above, but that can yield needlessly long scripts. We can also use the `mxModel` function to edit an existing model object, allowing us to change only the parameters that vary across classes. The `mxModel` call below begins with an existing `MxModel` object (`class1`) rather than a model name. The subsequent `mxPath` functions add new paths to the model, replacing any existing paths that describe the same relationship. As we did not give the model a name at the beginning of the `mxModel` function, we must use the `name` argument to identify this model by name.

```
# latent variances and covariance
latVars2     <- mxPath( from=c("intercept","slope"), arrows=2, connect="unique.pairs",
                        free=TRUE, values=c(1,.5,1), labels=c("vari2","cov2","vars2") )
# latent means
latMeans2    <- mxPath( from="one", to=c("intercept", "slope"), arrows=1,
                        free=TRUE, values=c(5,1), labels=c("meani2","means2") )
class2       <- mxModel(class1, name="Class2", latVars2, latMeans2)
```

While the class-specific models can be specified using either path or matrix specification, the class proportion parameters must be specified using a matrix, though it can be specified a number of different ways. The challenge of specifying class probabilities lies in their inherent constraint: class probabilities must be non-negative and sum to unity. The code below demonstrates one method of specifying class proportion parameters and rescaling them as probabilities.

This method for specifying class probabilities consists of two parts. In the first part, the matrix in the object `classP` contains two elements representing the class proportions for each class. One class is designated as a reference class by

fixing their proportion at a value of one (class 2 below). All other classes are assigned free parameters in this matrix, and should be interpreted as proportion of sample in that class per person in the reference class. These parameters should have a lower bound at or near zero. Specifying class proportions rather than class probabilities avoids the degrees of freedom issue inherent to class probability parameters by only estimating k-1 parameters for k classes.

```
classP      <- mxMatrix( type="Full", nrow=2, ncol=1,
                         free=c(TRUE, FALSE), values=1, lbound=0.001,
                         labels = c("p1","p2"), name="Props" )
```

We still need probabilities, which require the second step shown below. Dividing the class proportion matrix above by its sum will rescale the proportions into probabilities. This is slightly more difficult that it appears at first, as the k x 1 matrix of class proportions and the scalar sum of that matrix aren't conformable to either matrix or element-wise operations. Instead, we can use a Kronecker product of the class proportion matrix and the inverse of the sum of that matrix. This operation is carried out by the `mxAlgebra` function placed in the object `classS` below.

```
classS      <- mxAlgebra( Props%x%(1/sum(Props)), name="classProbs" )
```

There are several alternatives to the two functions above that merit discussion. While the‘‘mxConstraint‘‘ function would appear at first to be a simpler way to specify the class probabilities, but using the `mxConstraint` function complicates this type of model estimation. When all k class probabilities are freely estimated then constrained, then the class probability parameters are collinear, creating a parameter covariance matrix that is not of full rank. This prevents OpenMx from calculating standard errors for any model parameters. Additionally, there are multiple ways to use algebras different than the one above to specify the class proportion and/or class probability parameters, each varying in complexity and utility. While specifying models with two classes can be done slightly more simply than presented here, the above method is equally appropriate for all numbers of classes.

Finally, we can specify the mixture model. We must first specify the model's -2 log likelihood function defined as:

$$-2LL = -2 * \sum_{i=1}^{n} \sum_{k=1}^{m} \log(p_k l_{ki})$$

This is specified using an `mxAlgebra` function, and used as the argument to the `mxFitFunctionAlgebra` function. Then the fit function, matrices and algebras used to define the mixture distribution, the models for the respective classes and the data are all placed in one final `mxModel` object, shown below.

```
algFit      <- mxAlgebra( -2*sum(log(classProbs[1,1]%x%Class1.fitfunction
                         + classProbs[2,1]%x%Class2.fitfunction)),
                         name="mixtureObj")
fit         <- mxFitFunctionAlgebra("mixtureObj")
dataRaw     <- mxData( observed=myGrowthMixtureData, type="raw" )

gmm         <- mxModel("Growth Mixture Model",
                         dataRaw, class1, class2, classP, classS, algFit, fit )

gmmFit      <- mxRun(gmm, suppressWarnings=TRUE)

summary(gmmFit)
```

Multiple Runs: Serial Method

The results of a mixture model can sometimes depend on starting values. It is a good idea to run a mixture model with a variety of starting values to make sure results you find are not the result of a local minimum in the likelihood space. This section will describe a serial (i.e., running one model at a time) method for randomly generating starting values and re-running a model, which is appropriate for a wide range of methods. The next section will cover parallel (multiple models simultaneously) estimation procedures. Both of these examples are available in the `GrowthMixtureModelRandomStarts` demo.

** http://openmx.psyc.virginia.edu/svn/trunk/models/nightly/GrowthMixtureModelRandomStarts.R

One way to access the starting values in a model is by using the `omxGetParameters` function. This function takes an existing model as an argument and returns the names and values of all free parameters. Using this function on our growth mixture model, which is stored in an objected called `gmm`, gives us back the starting values we specified above.

```
    omxGetParameters(gmm)
#      p1 residual    vari1     cov1    vars1   meani1   means1
#     1.0      1.0      1.0      0.4      1.0      0.0     -1.0
#   vari2     cov2    vars2   meani2   means2
#     1.0      0.5      1.0      5.0      1.0
```

A companion function to `omxGetParameters` is `omxSetParameters`, which can be used to alter one or more named parameters in a model. This function can be used to change the values, freedom and labels of any parameters in a model, returning an MxModel object with the specified changes. The code below shows how to change the residual variance starting value from 1.0 to 0.5. Note that the output of the `omxSetParameters` function is placed back into the object `gmm`.

```
gmm <- omxSetParameters(gmm, labels="residual", values=0.5)
```

The MxModel in the object `gmm` can now be run and the results compared with other sets of staring values. Starting values can also be sampled from distributions, allowing users to automate starting value generation, which is demonstrated below. The `omxGetParameters` function is used to find the names of the free parameters and define three matrices: a matrix `input` that holds the starting values for any run; a matrix `output` that holds the converged values of each parameter; and a matrix `fit` that contains the -2 log likelihoods and other relevant model fit statistics. Each of these matrices contains one row for every set of starting values. Starting values are randomly generated from a set of uniform distributions using the `runif` function, allowing the ranges inherent to each parameter to be enforced (i.e., variances are positive, etc). A `for` loop repeatedly runs the model with starting values from the `input` matrix and places the final estimates and fit statistics in the `output` and `fit` matrices, respectively.

```
# how many trials?
trials <- 20

# place all of the parameter names in a vector
parNames <- names(omxGetParameters(gmm))

# make a matrix to hold all of the
input <- matrix(NA, trials, length(parNames))
dimnames(input) <- list(c(1: trials), c(parNames))

output <- matrix(NA, trials, length(parNames))
dimnames(output) <- list(c(1: trials), c(parNames))

fit <- matrix(NA, trials, 5)
dimnames(fit) <- list(c(1:trials), c("Minus2LL","Status","Iterations","pclass1","time"))

# populate the class probabilities
input[,"p1"] <- runif(trials, 0.1, 0.9)
input[,"p1"] <- input[,"p1"]/(1-input[,"p1"])

# populate the variances
v <- c("vari1", "vars1", "vari2", "vars2", "residual")
input[,v] <- runif(trials*5, 0, 10)

# populate the means
m <- c("meani1", "means1", "meani2", "means2")
input[,m] <- runif(trials*4, -5, 5)

# populate the covariances
```

```
r <- runif(trials*2, -0.9, 0.9)
scale <- c( sqrt(input[,"vari1"]*input[,"vars1"]), sqrt(input[,"vari2"]*input[,"vars2"]))
input[,c("cov1", "cov2")] <- r * scale

for (i in 1: trials){
    temp1 <- omxSetParameters(gmm, labels=parNames, values=input[i,] )
    temp1 <- mxModel(model=temp1, name=paste("Starting Values Set", i))
    temp2 <- mxRun(temp1, unsafe=TRUE, suppressWarnings=TRUE, checkpoint=TRUE)

    output[i,] <- omxGetParameters(temp2)
    fit[i,] <- c(
        temp2$output$Minus2LogLikelihood,
        temp2$output$status[[1]],
        temp2$output$iterations,
        round(temp2$classProbs$result[1,1], 4),
        temp2$output$wallTime
        )
    }
```

Viewing the contents of the `fit` matrix shows the -2 log likelihoods for each of the runs, as well as the convergence status, number of iterations and class probabilities, shown below.

```
fit[,1:4]
#       Minus2LL Status Iterations   pclass1
#    1  8739.050      0         41 0.3991078
#    2  8739.050      0         40 0.6008913
#    3  8739.050      0         44 0.3991078
#    4  8739.050      1         31 0.3991079
#    5  8739.050      0         32 0.3991082
#    6  8739.050      1         34 0.3991089
#    7  8966.628      0         22 0.9990000
#    8  8966.628      0         24 0.9990000
#    9  8966.628      0         23 0.0010000
#    10 8966.628      1         36 0.0010000
#    11 8963.437      6         25 0.9990000
#    12 8966.628      0         28 0.9990000
#    13 8739.050      1         47 0.6008916
#    14 8739.050      1         36 0.3991082
#    15 8739.050      0         43 0.3991076
#    16 8739.050      0         46 0.6008948
#    17 8739.050      1         50 0.3991092
#    18 8945.756      6         50 0.9902127
#    19 8739.050      0         53 0.3991085
#    20 8966.628      0         23 0.9990000
```

There are several things to note about the above results. First, the minimum -2 log likelihood was reached in 12 of 20 sets of staring values, all with NPSOL statuses of either zero (seven times) or one (five times). Additionally, the class probabilities are equivalent within five digits of precision, keeping in mind that no the model as specified contains no restriction as to which class is labeled "class 1" (probability equals .3991) and "class 2" (probability equals .6009). The other eight sets of starting values showed higher -2 log likelihood values and class probabilities at the set upper or lower bounds, indicating a local minimum. We can also view this information using R's `table` function.

```
table(round(fit[,1], 3), fit[,2])

#              0 1 6
#     8739.05  7 5 0
#     8945.756 0 0 1
#     8963.437 0 0 1
```

```
#     8966.628 5 1 0
```

We should have a great deal of confidence that the solution with class probabilities of .399 and .601 is the correct one.

Multiple Runs: Parallel Method

OpenMx supports multicore processing through the `snowfall` library, which is described in the "Multicore Execution" section of the documentation and in the following demo:

** http://openmx.psyc.virginia.edu/svn/trunk/models/passing/BootstrapParallel.R

Using multiple processors can greatly improve processing time for model estimation when a model contains independent submodels. While the growth mixture model in this example does contain submodels (i.e., the class specific models), they are not independent, as they both depend on a set of shared parameters ("residual", "pclass1").

However, multicore estimation can be used instead of the `for` loop in the above section for testing alternative sets of starting values. Instead of changing the starting values in the `gmm` object repeatedly, multiple copies of the model contained in `gmm` must be placed into parent or container model. Either the above `for` loop or a set of "apply" statements can be used to generate the model.

The example below first initializes the `snowfall` library, which also loads the `snow` library. The `sfInit` function initializes parallel; you must supply the number of processors on your computer or grid for the analysis, then reload OpenMx as a snowfall library.

```
require(snowfall)
sfInit(parallel=TRUE, cpus=4)
sfLibrary(OpenMx)
```

From there, parallel optimization requires that a holder or top model (named "Top" in the object `topModel` below) contain a set of independent submodels. In our example, each independent submodel will consist of a copy of the above `gmm` model with a different set of starting values. Using the matrix of starting values from the serial example above (`input`), we can create a function called `makeModel` that can be used to create these submodels. While this function is entirely optional, it allows us to use the `lapply` function to create a list of submodels for optimization. Once those submodels are placed in the `submodels` slot of the object `topModel`, we can run this model just like any other. A second function, `fitStats`, can then be used to get the results from each submodel.

```
topModel    <- mxModel("Top")

makeModel   <- function(modelNumber){
    temp    <- mxModel(gmm, independent=TRUE, name=paste("Iteration", modelNumber, sep=""))
    temp    <- omxSetParameters(temp, labels=parNames, values=input[modelNumber,])
    return(temp)
}

mySubs      <- lapply(1:20, makeModel)
topModel    <- mxModel(topModel, mySubs)
results     <- mxRun(topModel)

fitStats    <- function(model){
    retval  <- c(
        model$output$Minus2LogLikelihood,
        model$output$status[[1]],
        model$output$iterations,
        round(model$classProbs$result[1,1], 4)
        )
    return(retval)
}
```

```
resultsFit  <- t(omxSapply(results$submodels, fitStats))
sfStop()
```

This parallel method saves computational time, but requires additional coding. For models as small as the one in this example (total processing time of approximately 2 seconds), the speed-up from using the parallel version is marginal (approximately 35-50 seconds for the serial method against 20-30 seconds for the parallel version). However, as models get more complex or require a greater number of random starts, the parallel method can provide substantial time savings. Regardless of method, re-running models with varying starting values is an essential part of running multivariate models.

CHAPTER

THREE

EXAMPLES, MATRIX SPECIFICATION

3.1 Regression, Matrix Specification

Our next example will show how regression can be carried out from structural modeling perspective. This example is in three parts; a simple regression, a multiple regression, and multivariate regression. There are two versions of each example are available; one where the data is supplied as a covariance matrix and vector of means, and one with raw data. These examples are available in the following files: Parallel versions of this example, using matrix specification of models rather than paths, can be found here:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/SimpleRegression_MatrixCov.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/SimpleRegression_MatrixRaw.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/MultipleRegression_MatrixCov.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/MultipleRegression_MatrixRaw.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/MultivariateRegression_MatrixCov.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/MultivariateRegression_MatrixRaw.R

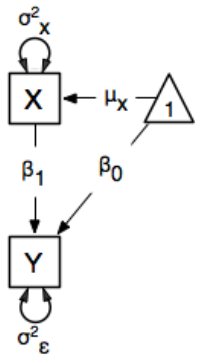
This example will focus on the RAM approach to building structural models. A parallel version of this example, using path-centric rather than matrix specification, is available here:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/SimpleRegression_PathCov.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/SimpleRegression_PathRaw.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/MultipleRegression_PathCov.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/MultipleRegression_PathRaw.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/MultivariateRegression_PathCov.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/MultivariateRegression_PathRaw.R

3.1.1 Simple Regression

We begin with a single dependent variable (y) and a single independent variable (x). The relationship between these variables takes the following form:

$$y = \beta_0 + \beta_1 * x + \epsilon$$



In this model, the mean of y is dependent on both regression coefficients (and by extension, the mean of x). The variance of y depends on both the residual variance and the product of the regression slope and the variance of x. This model contains five parameters from a structural modeling perspective β_0, β_1, σ^2_ϵ, and the mean and variance of x). We are modeling a covariance matrix with three degrees of freedom (two variances and one covariance) and a means vector with two degrees of freedom (two means). Because the model has as many parameters (5) as the data have degrees of freedom, this model is fully saturated.

Data

Our first step to running this model is to include the data to be analyzed. The data must first be placed in a variable or object. For raw data, this can be done with the `read.table` function. The data provided has a header row, indicating the names of the variables.

```
data(myRegDataRaw)
```

The names of the variables provided by the header row can be displayed with the names() function.

```
names(myRegDataRaw)
```

As you can see, our data has four variables in it. However, our model only contains two variables, x and y. To use only them, we will select only the variables we want and place them back into our data object. That can be done with the R code below.

```
SimpleDataRaw <- myRegDataRaw[,c("x","y")]
```

For covariance data, we do something very similar. We create an object to house our data. Instead of reading in raw data from an external file, we can include a covariance matrix. This requires the `matrix()` function, which needs to know what values are in the covariance matrix, how big it is, and what the row and column names are (in dimnames). As our model also references means, we will include a vector of means in a separate object. Data is selected in the same way as before.

```
myRegDataCov <- matrix(
    c(0.808,-0.110, 0.089, 0.361,
     -0.110, 1.116, 0.539, 0.289,
      0.089, 0.539, 0.933, 0.312,
      0.361, 0.289, 0.312, 0.836), nrow=4,
    dimnames=list( c("w","x","y","z"), c("w","x","y","z")) )

SimpleDataCov <- myRegDataCov[c("x","y"),c("x","y")]

myRegDataMeans <- c(2.582, 0.054, 2.574, 4.061)
names(myRegDataMeans) <- c("w","x","y","z")

SimpleDataMeans <- myRegDataMeans[c(2,3)]
```

Model Specification

The following code contains all of the components of our model. Before running a model, the OpenMx library must be loaded into R using either the `require()` or `library()` function. This code uses the `mxModel` function to create an `MxModel` object, which we will then run. Note the difference in capitalization for the first letter.

```
require(OpenMx)

dataRaw      <- mxData( observed=SimpleDataRaw, type="raw" )
matrA        <- mxMatrix( type="Full", nrow=2, ncol=2,
                          free=c(F,F,T,F), values=c(0,0,1,0),
                          labels=c(NA,NA,"beta1",NA), byrow=TRUE, name="A" )
matrS        <- mxMatrix( type="Symm", nrow=2, ncol=2,
                          free=c(T,F,F,T), values=c(1,0,0,1),
                          labels=c("varx",NA,NA,"residual"), byrow=TRUE, name="S" )
matrF        <- mxMatrix( type="Iden", nrow=2, ncol=2, name="F" )
matrM        <- mxMatrix( type="Full", nrow=1, ncol=2,
                          free=c(T,T), values=c(0,0),
                          labels=c("meanx","beta0"), name="M")
expRAM       <- mxExpectationRAM("A","S","F","M", dimnames=c("x","y"))
funML        <- mxFitFunctionML()

uniRegModel  <- mxModel("Simple Regression Matrix Specification",
                        dataRaw, matrA, matrS, matrF, matrM, expRAM, funML)
```

We are presenting the code here in the piecewise style and thus will create several of the pieces up front before putting them together in the `mxModel` statement. We will pre-specify the MxData object *dataRaw*, and the various MxMatrix objects to define the **A**, **S**, **F** and **M** matrices, as well as the expectation and fit function objects that link them together. These are then included as arguments of the MxModel object.

This `mxModel` function can be split into several parts. First, we give the model a title. The first argument in an `mxModel` function has a special function. If an object or variable containing an `MxModel` object is placed here, then `mxModel` adds to or removes pieces from that model. If a character string (as indicated by double quotes) is placed first, then that becomes the name of the model. Models may also be named by including a `name` argument. This model is named `Simple Regression Matrix Specification`.

The second component of our code creates an `MxData` object. The example above, reproduced here, first references the object where our data is, then uses the `type` argument to specify that this is raw data.

```
dataRaw      <- mxData( observed=SimpleDataRaw, type="raw" )
```

If we were to use a covariance matrix and vector of means as data, we would replace the existing `mxData` function with this one:

```
dataCov      <- mxData( observed=SimpleDataCov, type="cov", numObs=100,
                        means=SimpleDataMeans )
```

The next four functions specify the four matrices that make up the RAM specified model. Each of these matrices defines part of the relationship between the observed variables. These matrices are then combined by the expectation function, which follows the four `mxMatrix` functions, to define the expected covariances and means for the supplied data. In all of the included matrices, the order of variables matches those in the data. Therefore, the first row and column of all matrices corresponds to the *x* variable, while the second row and column of all matrices corresponds to the *y* variable.

The **A** matrix is created first. This matrix specifies all of the asymmetric paths or regressions among the variables. A free parameter in the **A** matrix defines a regression of the variable represented by that row on the variable represented by that column. For clarity, all matrices are specified with the `byrow` argument set to `TRUE`, which allows better correspondence between the matrices as displayed below and their position in `mxMatrix` objects. In the section of

code below, a free parameter is specified as the regression of *y* on *x*, with a starting value of 1, and a label of "beta1". This matrix is named "A".

```
# asymmetric paths
matrA        <- mxMatrix( type="Full", nrow=2, ncol=2,
                          free=c(F,F,T,F), values=c(0,0,1,0),
                          labels=c(NA,NA,"beta1",NA), byrow=TRUE, name="A" )
```

The second mxMatrix function specifies the **S** matrix. This matrix specifies all of the symmetric paths or covariances among the variables. By definition, this matrix is symmetric, but all elements are specified in the matrix below. It is also possible to just specify the unique elements, being the elements on the diagonal and below (or above). A free parameter in the **S** matrix represents a variance or covariance between the variables represented by the row and column that parameter is in. In the code below, two free parameters are specified. The free parameter in the first row and column of the **S** matrix is the variance of *x* (labeled "varx"), while the free parameter in the second row and column is the residual variance of *y* (labeled "residual"). This matrix is named "S".

```
# symmetric paths
matrS        <- mxMatrix( type="Symm", nrow=2, ncol=2,
                          free=c(T,F,F,T), values=c(1,0,0,1),
                          labels=c("varx",NA,NA,"residual"), byrow=TRUE, name="S" )
```

The third mxMatrix function specifies the **F** matrix. This matrix is used to filter latent variables out of the expected covariance of the manifest variables, or to reorder the manifest variables. When there are no latent variables in a model and the order of manifest variables is the same in the model as in the data, then this filter matrix is simply an identity matrix.

There are no free parameters in any **F** matrix.

```
# filter matrix
matrF        <- mxMatrix( type="Iden", nrow=2, ncol=2, name="F" )
```

The fourth and final mxMatrix function specifies the **M** matrix. This matrix is used to specify the means and intercepts of our model. Exogenous or independent variables receive means, while endogenous or dependent variables get intercepts, or means conditional on regression on other variables. This matrix contains only one row. This matrix consists of two free parameters; the mean of *x* (labeled "meanx") and the intercept of *y* (labeled "beta0"). This matrix gives starting values of 0 for both parameters, and is named "M".

```
# means
matrM        <- mxMatrix( type="Full", nrow=1, ncol=2,
                          free=c(T,T), values=c(0,0),
                          labels=c("meanx","beta0"), name="M")
```

The final parts of this model are the expectation and fit functions. These define how the specified matrices combine to create the expected covariance matrix and the expected means of the data, and the fit function to be minimized, respectively. In a RAM specified model, the expected covariance matrix is defined as:

$$ExpCovariance = F * (I - A)^{-1} * S * ((I - A)^{-1})' * F'$$

The expected means are defined as:

$$ExpMean = F * (I - A)^{-1} * M$$

The free parameters in the model can then be estimated using maximum likelihood for covariance and means data, and full information maximum likelihood for raw data. Although users may define their own expected covariance matrices using mxExpectationNormal and other functions in OpenMx, the mxExpectationRAM function computes the expected covariance and means matrices when the **A**, **S**, **F** and **M** matrices are specified. The **M** matrix is required both for raw data and for covariance or correlation data that includes a means vector. The mxExpectationRAM function takes four arguments, which are the names of the **A**, **S**, **F** and **M** matrices in your model. The mxFitFunctionML yields maximum likelihood estimates of structural equation models. It uses full information maximum likelihood when the data are raw.

```
expRAM       <- mxExpectationRAM("A","S","F","M", dimnames=c("x","y"))
funML        <- mxFitFunctionML()
```

The model now includes an observed covariance matrix (i.e., data), model matrices, an expectation function, and a fit function. So the model has all the required elements to define the expected covariance matrix and estimate parameters.

Model Fitting

We've created an `MxModel` object, and placed it into an object or variable named `uniRegModel`. We can run this model by using the `mxRun` function, which is placed in the object `uniRegFit` in the code below. We then view the output by referencing the `output` slot, as shown here.

```
uniRegFit <- mxRun(uniRegModel)
```

The `$output` slot contains a great deal of information, including parameter estimates and information about the matrix operations underlying our model. A more parsimonious report on the results of our model can be viewed using the `summary()` function, as shown here.

```
uniRegFit$output
summary(uniRegFit)
```

Alternative Specification

Rather than using the RAM approach the regression model with matrices can also be specified differently and more directly comparable to the regression equation. This approach uses a special kind of variable, called a definition variable, which will be explained in more detail in *Definition Variables, Matrix Specification*. Below is the complete code.

```
selVars <- c("y")

dataRaw     <- mxData( observed=SimpleDataRaw, type="raw" )
dataX       <- mxMatrix( type="Full", nrow=1, ncol=1,
                         free=FALSE, labels=c("data.x"), name="X" )
intercept   <- mxMatrix( type="Full", nrow=1, ncol=1,
                         free=T, values=0, labels="beta0", name="intercept" )
regCoef     <- mxMatrix( type="Full", nrow=1, ncol=1,
                         free=T, values=1, labels="beta1", name="regCoef" )
resVar      <- mxMatrix( type="Diag", nrow=1, ncol=1,
                         free=T, values=1, labels="residual", name="resVar" )
expMean     <- mxAlgebra( expression= intercept + regCoef %*% X, name="expMean" )
expCov      <- mxAlgebra( expression= resVar, name="expCov" )
exp         <- mxExpectationNormal( covariance="expCov", means="expMean",
                                    dimnames=selVars )
funML       <- mxFitFunctionML()

uniRegModel <- mxModel("Simple Regression Matrix Specification",
                       dataRaw, dataX, intercept, regCoef, resVar,
                       expMean, expCov, exp, funML )
```

Note the the `mxData` statement has not changed. The first key change is that we put the variable *x* in a matrix X by using a special type of label assignment in an `mxMatrix` statement. The matrix is a `Full` **1x1** fixed matrix. The label has two parts: the first part is called `data.` which indicates that the name used in the second part (`x`) is a variable found in the dataset referred to in the `mxData` statement. This variable can now be used as part of any algebra, and is no longer considered a dependent variable.

```
dataRaw     <- mxData( observed=SimpleDataRaw, type="raw" )
dataX       <- mxMatrix( type="Full", nrow=1, ncol=1,
                         free=FALSE, labels=c("data.x"), name="X" )
```

Next, we specify three matrices, one for the intercept, one for the regression coefficient, and one for the residual variance. In this example, the first two matrices are `Full` **1x1** matrices with a free element. We give them labels consistent with their names in a regression equation, namely `beta0` and `beta1`. The third matrix is a `Diag` **1x1** matrix with a free element for the residual variance, named `resVar`.

```
intercept   <- mxMatrix( type="Full", nrow=1, ncol=1,
                         free=T, values=0, labels="beta0", name="intercept" )
regCoef     <- mxMatrix( type="Full", nrow=1, ncol=1,
                         free=T, values=1, labels="beta1", name="regCoef" )
resVar      <- mxMatrix( type="Diag", nrow=1, ncol=1,
                         free=T, values=1, labels="residual", name="resVar" )
```

Now we can explicitly specify the formula for the expected means and covariances using `mxAlgebra` statement. Note that we here use the variable in the matrix **X** as part of the algebra. We regress y on x in the means model and simply have the residual variance in the covariance model.

```
expMean     <- mxAlgebra( expression= intercept + regCoef %*% X, name="expMean" )
expCov      <- mxAlgebra( expression= resVar, name="expCov" )
```

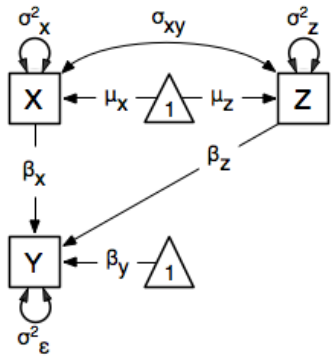
Finally, we call up the results of the algebras as the arguments for the expectation function. The dimnames map the data to the model. Note that `selVars` now includes only the y variable. The fit function declares that the model is fit using maximum likelihood. When combined with raw data this means full information maximum likelihood (FIML) is optimized.

```
exp         <- mxExpectationNormal( covariance="expCov", means="expMean",
                                    dimnames=selVars )
funML       <- mxFitFunctionML()
```

3.1.2 Multiple Regression

In the next part of this demonstration, we move to multiple regression. The regression equation for our model looks like this:

$$y = \beta_0 + \beta_x * x + \beta_z * z + \epsilon$$



Our dependent variable y is now predicted from two independent variables, x and z. Our model includes 3 regression parameters (β_0, β_x, β_z), a residual variance (σ_ϵ^2) and the observed means, variances and covariance of x and z, for a total of 9 parameters. Just as with our simple regression, this model is fully saturated.

We prepare our data the same way as before, selecting three variables instead of two.

```
MultipleDataRaw <- myRegDataRaw[,c("x","y","z")]

MultipleDataCov <- myRegDataCov[c("x","y","z"),c("x","y","z")]

MultipleDataMeans <- myRegDataMeans[c(2,3,4)]
```

Now, we can move on to our code. It is identical in structure to our simple regression code, containing the same **A**, **S**, **F** and **M** matrices. With the addition of a third variable, the **A**, **S** and **F** matrices become **3x3**, while the **M** matrix becomes a **1x3** matrix.

```
dataRaw     <- mxData( observed=MultipleDataRaw, type="raw" )
matrA       <- mxMatrix( type="Full", nrow=3, ncol=3,
                         free=c(F,F,F,  T,F,T,  F,F,F),
                         values=c(0,0,0,  1,0,1,  0,0,0),
                         labels=c(NA,NA,NA, "betax",NA,"betaz", NA,NA,NA),
                         byrow=TRUE, name="A" )
matrS       <- mxMatrix( type="Symm", nrow=3, ncol=3,
                         free=c(T,F,T,  F,T,F,  T,F,T),
                         values=c(1,0,.5,  0,1,0,  .5,0,1),
                         labels=c("varx",NA,"covxz", NA,"residual",NA, "covxz",NA,"varz"),
                         byrow=TRUE, name="S" )
matrF       <- mxMatrix( type="Iden", nrow=3, ncol=3, name="F" )
matrM       <- mxMatrix( type="Full", nrow=1, ncol=3,
                         free=c(T,T,T), values=c(0,0,0),
                         labels=c("meanx","beta0","meanz"), name="M" )
exp         <- mxExpectationRAM("A","S","F","M", dimnames=c("x","y","z") )
funML       <- mxFitFunctionML()

multiRegModel <- mxModel("Multiple Regression Matrix Specification",
                         dataRaw, matrA, matrS, matrF, matrM, exp, funML)
```

The `mxData` function now takes a different data object (`MultipleDataRaw` replaces `SingleDataRaw`, adding an additional variable), but is otherwise unchanged. The `mxExpectationRAM` and `mxFitFunctionML` do not change. The only differences between this model and the simple regression script can be found in the **A**, **S**, **F** and **M** matrices, which have expanded to accommodate a second independent variable.

The **A** matrix now contains two free parameters, representing the regressions of the dependent variable *y* on both *x* and *z*. As regressions appear on the row of the dependent variable and the column of the independent variable, these two parameters are both on the second (*y*) row of the **A** matrix.

```
# asymmetric paths
matrA       <- mxMatrix( type="Full", nrow=3, ncol=3,
                         free=c(F,F,F,  T,F,T,  F,F,F),
                         values=c(0,0,0,  1,0,1,  0,0,0),
                         labels=c(NA,NA,NA, "betax",NA,"betaz", NA,NA,NA),
                         byrow=TRUE, name="A" )
```

We've made a similar changes in the other matrices. The **S** matrix includes not only a variance term for the *z* variable, but also a covariance between the two independent variables. The **F** matrix still does not contain free parameters, but has expanded in size. The **M** matrix includes an additional free parameter for the mean of *z*.

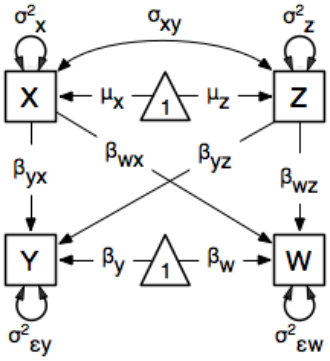
The model is run and output is viewed just as before, using the `mxRun` function, `$output` and the `summary()` function to run, view and summarize the completed model.

3.1.3 Multivariate Regression

The structural modeling approach allows for the inclusion of not only multiple independent variables (i.e., multiple regression), but multiple dependent variables as well (i.e., multivariate regression). Versions of multivariate regression

are sometimes fit under the heading of path analysis. This model will extend the simple and multiple regression frameworks we've discussed above, adding a second dependent variable *w*.

$$y = \beta_y + \beta_{yx} * x + \beta_{yz} * z + \epsilon_y$$
$$w = \beta_w + \beta_{wx} * x + \beta_{wz} * z + \epsilon_w$$



We now have twice as many regression parameters, a second residual variance, and the same means, variances and covariances of our independent variables. As with all of our other examples, this is a fully saturated model.

Data import for this analysis will actually be slightly simpler than before. The data we imported for the previous examples contains only the four variables we need for this model. We can use `myRegDataRaw`, `myRegDataCov`, and `myRegDataMeans` in our models.

```
    data(myRegDataRaw)

myRegDataCov <- matrix(
    c(0.808,-0.110, 0.089, 0.361,
     -0.110, 1.116, 0.539, 0.289,
      0.089, 0.539, 0.933, 0.312,
      0.361, 0.289, 0.312, 0.836), nrow=4,
    dimnames=list( c("w","x","y","z"), c("w","x","y","z")) )

myRegDataMeans <- c(2.582, 0.054, 2.574, 4.061)
```

Our code should look very similar to our previous two models. The `mxData` function will reference the data referenced above, while the `mxExpectationRAM` again refers to the **A**, **S**, **F** and **M** matrices. Just as with the multiple regression example, the **A**, **S** and **F** expand to order **4x4**, and the **M** matrix now contains one row and four columns.

```
dataRaw     <- mxData( observed=myRegDataRaw, type="raw" )
matrA       <- mxMatrix( type="Full", nrow=4, ncol=4,
                         free=c(F,T,F,T,  F,F,F,F,  F,T,F,T,  F,F,F,F),
                         values=c(0,1,0,1,  0,0,0,0,  0,1,0,1,  0,0,0,0),
                         labels=c(NA,"betawx",NA,"betawz",
                                  NA, NA,     NA, NA,
                                  NA,"betayx",NA,"betayz",
                                  NA, NA,     NA, NA),
                         byrow=TRUE, name="A" )
matrS       <- mxMatrix( type="Symm", nrow=4, ncol=4,
                         free=c(T,F,F,F,  F,T,F,T,  F,F,T,F,  F,T,F,T),
                         values=c(1, 0,0, 0,  0, 1,0,.5,  0, 0,1, 0,  0,.5,0, 1),
                         labels=c("residualw", NA,      NA,          NA,
                                  NA,         "varx",   NA,         "covxz",
                                  NA,          NA,     "residualy", NA,
                                  NA,         "covxz",  NA,         "varz"),
                         byrow=TRUE, name="S" )
```

```
matrF       <- mxMatrix( type="Iden", nrow=4, ncol=4, name="F" )
matrM       <- mxMatrix( type="Full", nrow=1, ncol=4,
                         free=c(T,T,T,T), values=c(0,0,0,0),
                         labels=c("betaw","meanx","betay","meanz"), name="M" )
exp         <- mxExpectationRAM("A","S","F","M", dimnames=c("w","x","y","z") )
funML       <- mxFitFunctionML()

multivariateRegModel <- mxModel("Multiple Regression Matrix Specification",
                         dataRaw, matrA, matrS, matrF, matrM, exp, funML)
```

The only additional components to our `mxMatrix` functions are the inclusion of the *w* variable, which becomes the first row and column of all matrices. The model is run and output is viewed just as before, using the `mxRun` function, `$output` and the `summary()` function to run, view and summarize the completed model.

These models may also be specified using paths instead of matrices. See *Regression, Path Specification* for path specification of these models.

3.2 Factor Analysis, Matrix Specification

This example will demonstrate latent variable modeling via the common factor model using RAM matrices for model specification. We'll walk through two applications of this approach: one with a single latent variable, and one with two latent variables. As with previous examples, these two applications are split into four files, with each application represented separately with raw and covariance data. These examples can be found in the following files:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/OneFactorModel_MatrixCov.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/OneFactorModel_MatrixRaw.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/TwoFactorModel_MatrixCov.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/TwoFactorModel_MatrixRaw.R

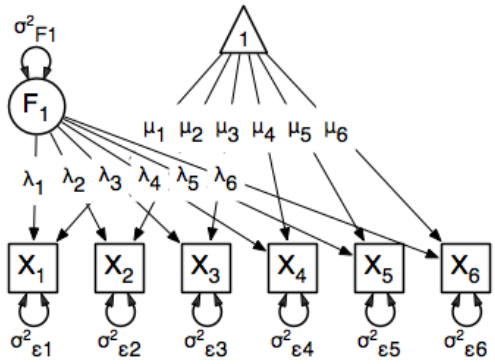
Parallel versions of this example, using path-centric specification of models rather than matrices, can be found here:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/OneFactorModel_PathCov.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/OneFactorModel_PathRaw.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/TwoFactorModel_PathCov.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/TwoFactorModel_PathRaw.R

3.2.1 Common Factor Model

The common factor model is a method for modeling the relationships between observed variables believed to measure or indicate the same latent variable. While there are a number of exploratory approaches to extracting latent factor(s), this example uses structural modeling to fit confirmatory factor models. The model for any person and path diagram of the common factor model for a set of variables x_1 - x_6 are given below.

$$x_{ij} = \mu_j + \lambda_j * \eta_i + \epsilon_{ij}$$



While 19 parameters are displayed in the equation and path diagram above (six manifest variances, six manifest means, six factor loadings and one factor variance), we must constrain either the factor variance or one factor loading to a constant to identify the model and scale the latent variable. As such, this model contains 18 parameters. Unlike the manifest variable examples we've run up until now, this model is not fully saturated. The means and covariance matrix for six observed variables contain 27 degrees of freedom, and thus our model contains 9 degrees of freedom.

Data

Our first step to running this model is to include the data to be analyzed. The data for this example contain nine variables. We'll select the six we want for this model using the selection operators used in previous examples. Both raw and covariance data are included below, but only one is required for any model.

```
data(myFADataRaw)
names(myFADataRaw)

oneFactorRaw <- myFADataRaw[,c("x1", "x2", "x3", "x4", "x5", "x6")]

myFADataCov <- matrix(
    c(0.997, 0.642, 0.611, 0.672, 0.637, 0.677, 0.342, 0.299, 0.337,
      0.642, 1.025, 0.608, 0.668, 0.643, 0.676, 0.273, 0.282, 0.287,
      0.611, 0.608, 0.984, 0.633, 0.657, 0.626, 0.286, 0.287, 0.264,
      0.672, 0.668, 0.633, 1.003, 0.676, 0.665, 0.330, 0.290, 0.274,
      0.637, 0.643, 0.657, 0.676, 1.028, 0.654, 0.328, 0.317, 0.331,
      0.677, 0.676, 0.626, 0.665, 0.654, 1.020, 0.323, 0.341, 0.349,
      0.342, 0.273, 0.286, 0.330, 0.328, 0.323, 0.993, 0.472, 0.467,
      0.299, 0.282, 0.287, 0.290, 0.317, 0.341, 0.472, 0.978, 0.507,
      0.337, 0.287, 0.264, 0.274, 0.331, 0.349, 0.467, 0.507, 1.059), nrow=9,
dimnames=list( c("x1","x2","x3","x4","x5","x6","y1","y2","y3"),
               c("x1","x2","x3","x4","x5","x6","y1","y2","y3")) )

oneFactorCov <- myFADataCov[c("x1","x2","x3","x4","x5","x6"),
                            c("x1","x2","x3","x4","x5","x6")]

myFADataMeans <- c(2.988, 3.011, 2.986, 3.053, 3.016, 3.010, 2.955, 2.956, 2.967)
names(myFADataMeans) <- c("x1","x2","x3","x4","x5","x6")

oneFactorMeans <- myFADataMeans[1:6]
```

Model Specification

The following code contains all of the components of our model. Before running a model, the OpenMx library must be loaded into R using either the `require()` or `library()` function. All objects required for estimation (data,

matrices, an expectation function, and a fit function) are specified. This code uses the `mxModel` function to create an `MxModel` object, which we will then run.

```
manifestVars <- c("x1","x2","x3","x4","x5","x6")
latentVars <- "F1"

dataRaw      <- mxData( observed=myFADataRaw, type="raw" )
matrA        <- mxMatrix( type="Full", nrow=7, ncol=7,
                          free=  c(F,F,F,F,F,F,F,
                                   F,F,F,F,F,F,T,
                                   F,F,F,F,F,F,T,
                                   F,F,F,F,F,F,T,
                                   F,F,F,F,F,F,T,
                                   F,F,F,F,F,F,T,
                                   F,F,F,F,F,F,F),
                          values=c(0,0,0,0,0,0,1,
                                   0,0,0,0,0,0,1,
                                   0,0,0,0,0,0,1,
                                   0,0,0,0,0,0,1,
                                   0,0,0,0,0,0,1,
                                   0,0,0,0,0,0,1,
                                   0,0,0,0,0,0,0),
                          labels=c(NA,NA,NA,NA,NA,NA,"l1",
                                   NA,NA,NA,NA,NA,NA,"l2",
                                   NA,NA,NA,NA,NA,NA,"l3",
                                   NA,NA,NA,NA,NA,NA,"l4",
                                   NA,NA,NA,NA,NA,NA,"l5",
                                   NA,NA,NA,NA,NA,NA,"l6",
                                   NA,NA,NA,NA,NA,NA,NA),
                          byrow=TRUE, name="A" )
matrS        <- mxMatrix( type="Symm", nrow=7, ncol=7,
                          free=  c(T,F,F,F,F,F,F,
                                   F,T,F,F,F,F,F,
                                   F,F,T,F,F,F,F,
                                   F,F,F,T,F,F,F,
                                   F,F,F,F,T,F,F,
                                   F,F,F,F,F,T,F,
                                   F,F,F,F,F,F,T),
                          values=c(1,0,0,0,0,0,0,
                                   0,1,0,0,0,0,0,
                                   0,0,1,0,0,0,0,
                                   0,0,0,1,0,0,0,
                                   0,0,0,0,1,0,0,
                                   0,0,0,0,0,1,0,
                                   0,0,0,0,0,0,1),
                          labels=c("e1",NA,  NA,  NA,  NA,  NA,  NA,
                                   NA, "e2", NA,  NA,  NA,  NA,  NA,
                                   NA,  NA, "e3", NA,  NA,  NA,  NA,
                                   NA,  NA,  NA, "e4", NA,  NA,  NA,
                                   NA,  NA,  NA,  NA, "e5", NA,  NA,
                                   NA,  NA,  NA,  NA,  NA, "e6", NA,
                                   NA,  NA,  NA,  NA,  NA,  NA, "varF1"),
                          byrow=TRUE, name="S" )
matrF        <- mxMatrix( type="Full", nrow=6, ncol=7,
                          free=FALSE,
                          values=c(1,0,0,0,0,0,0,
                                   0,1,0,0,0,0,0,
                                   0,0,1,0,0,0,0,
                                   0,0,0,1,0,0,0,
```

```
                                    0,0,0,0,1,0,0,
                                    0,0,0,0,0,1,0),
                          byrow=TRUE, name="F" )
matrM       <- mxMatrix( type="Full", nrow=1, ncol=7,
                          free=c(T,T,T,T,T,T,F),
                          values=c(1,1,1,1,1,1,0),
                          labels=c("meanx1","meanx2","meanx3",
                                   "meanx4","meanx5","meanx6",NA),
                          name="M" )
exp         <- mxExpectationRAM("A","S","F","M",
                                dimnames=c(manifestVars, latentVars))
funML       <- mxFitFunctionML()

oneFactorModel <- mxModel("Common Factor Model Matrix Specification",
                          dataRaw, matrA, matrS, matrF, matrM, exp, funML)
```

This `mxModel` function can be split into several parts. First, we give the model a name. The first argument in an `mxModel` function has a special function. If an object or variable containing an `MxModel` object is placed here, then `mxModel` adds to or removes pieces from that model. If a character string (as indicated by double quotes) is placed first, then that becomes the name of the model. Models may also be named by including a `name` argument. This model is named `"Common Factor Model Matrix Specification"`.

The second component of our code creates an `MxData` object. The example above, reproduced here, first references the object where our data is, then uses the `type` argument to specify that this is raw data.

```
dataRaw     <- mxData( observed=myFADataRaw, type="raw" )
```

If we were to use a covariance matrix and vector of means as data, we would replace the existing `mxData` function with this one:

```
dataCov     <- mxData( observed=oneFactorCov, type="cov", numObs=500,
                       means=oneFactorMeans )
```

Model specification is carried out using `mxMatrix` functions to create matrices for a RAM specified model. The **A** matrix specifies all of the asymmetric paths or regressions in our model. In the common factor model, these parameters are the factor loadings. This matrix is square, and contains as many rows and columns as variables in the model (manifest and latent, typically in that order). Regressions are specified in the **A** matrix by placing a free parameter in the row of the dependent variable and the column of independent variable.

The common factor model requires that one parameter (typically either a factor loading or factor variance) be constrained to a constant value. In our model, we will constrain the first factor loading to a value of 1, and let all other loadings be freely estimated. All factor loadings have a starting value of one and labels of `"l1"` - `"l6"`.

```
# asymmetric paths
matrA       <- mxMatrix( type="Full", nrow=7, ncol=7,
                          free=  c(F,F,F,F,F,F,F,
                                   F,F,F,F,F,F,T,
                                   F,F,F,F,F,F,T,
                                   F,F,F,F,F,F,T,
                                   F,F,F,F,F,F,T,
                                   F,F,F,F,F,F,T,
                                   F,F,F,F,F,F,F),
                          values=c(0,0,0,0,0,0,1,
                                   0,0,0,0,0,0,1,
                                   0,0,0,0,0,0,1,
                                   0,0,0,0,0,0,1,
                                   0,0,0,0,0,0,1,
                                   0,0,0,0,0,0,1,
                                   0,0,0,0,0,0,0),
```

```
                        labels=c(NA,NA,NA,NA,NA,NA,"l1",
                                 NA,NA,NA,NA,NA,NA,"l2",
                                 NA,NA,NA,NA,NA,NA,"l3",
                                 NA,NA,NA,NA,NA,NA,"l4",
                                 NA,NA,NA,NA,NA,NA,"l5",
                                 NA,NA,NA,NA,NA,NA,"l6",
                                 NA,NA,NA,NA,NA,NA,NA),
                        byrow=TRUE, name="A" )
```

The second matrix in a RAM model is the **S** matrix, which specifies the symmetric or covariance paths in our model. This matrix is symmetric and square, and contains as many rows and columns as variables in the model (manifest and latent, typically in that order). The symmetric paths in our model consist of six residual variances and one factor variance. All of these variances are given starting values of one and labels `"e1"` - `"e6"` and `"varF1"`.

```
# symmetric paths
matrS        <- mxMatrix( type="Symm", nrow=7, ncol=7,
                        free=  c(T,F,F,F,F,F,F,
                                 F,T,F,F,F,F,F,
                                 F,F,T,F,F,F,F,
                                 F,F,F,T,F,F,F,
                                 F,F,F,F,T,F,F,
                                 F,F,F,F,F,T,F,
                                 F,F,F,F,F,F,T),
                        values=c(1,0,0,0,0,0,0,
                                 0,1,0,0,0,0,0,
                                 0,0,1,0,0,0,0,
                                 0,0,0,1,0,0,0,
                                 0,0,0,0,1,0,0,
                                 0,0,0,0,0,1,0,
                                 0,0,0,0,0,0,1),
                        labels=c("e1",NA,  NA,  NA,  NA,  NA,  NA,
                                 NA, "e2", NA,  NA,  NA,  NA,  NA,
                                 NA,  NA, "e3", NA,  NA,  NA,  NA,
                                 NA,  NA,  NA, "e4", NA,  NA,  NA,
                                 NA,  NA,  NA,  NA, "e5", NA,  NA,
                                 NA,  NA,  NA,  NA,  NA, "e6", NA,
                                 NA,  NA,  NA,  NA,  NA,  NA, "varF1"),
                        byrow=TRUE, name="S" )
```

The third matrix in our RAM model is the **F** or filter matrix. Our data contains six observed variables, but the **A** and **S** matrices contain seven rows and columns. For our model to define the covariances present in our data, we must have some way of projecting the relationships defined in the **A** and **S** matrices onto our data. The **F** matrix filters the latent variables out of the expected covariance matrix, and can also be used to reorder variables.

The **F** matrix will always contain the same number of rows as manifest variables and columns as total (manifest and latent) variables. If the manifest variables in the **A** and **S** matrices precede the latent variables and are in the same order as the data, then the **F** matrix will be the horizontal adhesion of an identity matrix and a zero matrix. This matrix contains no free parameters, and is made with the `mxMatrix` function below.

```
# filter matrix
matrF        <- mxMatrix( type="Full", nrow=6, ncol=7,
                        free=FALSE,
                        values=c(1,0,0,0,0,0,0,
                                 0,1,0,0,0,0,0,
                                 0,0,1,0,0,0,0,
                                 0,0,0,1,0,0,0,
                                 0,0,0,0,1,0,0,
                                 0,0,0,0,0,1,0),
```

```
                        byrow=TRUE, name="F" )
```

The last matrix of our model is the **M** matrix, which defines the means and intercepts for our model. This matrix describes all of the regressions on the constant in a path model, or the means conditional on the means of exogenous variables. This matrix contains a single row, and one column for every manifest and latent variable in the model. In our model, the latent variable has a constrained mean of zero, while the manifest variables have freely estimated means, labeled "meanx1" through "meanx6".

```
# means
matrM        <- mxMatrix( type="Full", nrow=1, ncol=7,
                        free=c(T,T,T,T,T,T,F),
                        values=c(1,1,1,1,1,1,0),
                        labels=c("meanx1","meanx2","meanx3",
                                 "meanx4","meanx5","meanx6",NA),
                        name="M" )
```

The final parts of this model are the expectation function and the fit function. The expectation defines how the specified matrices combine to create the expected covariance matrix of the data. The fit defines how the expectation is compared to the data to create a single scalar number that is minimized. In a RAM specified model, the expected covariance matrix is defined as:

$$ExpCovariance = F * (I - A)^{-1} * S * ((I - A)^{-1})' * F'$$

The expected means are defined as:

$$ExpMean = F * (I - A)^{-1} * M$$

The free parameters in the model can then be estimated using maximum likelihood for covariance and means data, and full information maximum likelihood for raw data. Although users may define their own expected covariance matrices using `mxExpectationNormal` and other functions in OpenMx, the `mxExpectationRAM` function computes the expected covariance and means matrices when the **A**, **S**, **F** and **M** matrices are specified. The **M** matrix is required both for raw data and for covariance or correlation data that includes a means vector. The `mxExpectationRAM` function takes four arguments, which are the names of the **A**, **S**, **F** and **M** matrices in your model. The `mxFitFunctionML` yields maximum likelihood estimates of structural equation models. It uses full information maximum likelihood when the data are raw.

```
exp          <- mxExpectationRAM("A","S","F","M",
                                 dimnames=c(manifestVars, latentVars))
funML        <- mxFitFunctionML()
```

The model now includes an observed covariance matrix (i.e., data), model matrices, an expectation function, and a fit function. So the model has all the required elements to define the expected covariance matrix and estimate parameters.

The model can now be run using the `mxRun` function, and the output of the model can be accessed from the `$output` slot of the resulting model. A summary of the output can be reached using `summary()`.

```
oneFactorFit <- mxRun(oneFactorModel)

oneFactorFit$output
summary(oneFactorFit)
```

Rather than specifying the model using RAM notation, we can also write the model explicitly with self-declared matrices, matching the formula for the expected mean and covariance structure of the one factor model:

$$mu_x = varMeans + (facLoadings * facMeans)'$$
$$sigma_x = facLoadings * facVariances * facLoadings' + resVariances$$

We start with displaying the complete script. Note that we have used the succinct form of coding and that the `mxData` command did not change.

```
dataRaw      <- mxData( observed=myFADataRaw, type="raw" )
facLoads     <- mxMatrix( type="Full", nrow=6, ncol=1, values=1, free=c(F,T,T,T,T,T),
                          labels=c("l1","l2","l3","l4","l5","l6"), name="facLoadings" )
facVars      <- mxMatrix( type="Symm", nrow=1, ncol=1, values=1, free=T,
                          labels="varF1", name="facVariances" )
resVars      <- mxMatrix( type="Diag", nrow=6, ncol=6, free=T, values=1,
                          labels=c("e1","e2","e3","e4","e5","e6"), name="resVariances" )
varMeans     <- mxMatrix( type="Full", nrow=1, ncol=6, values=1, free=T,
                          labels=c("meanx1","meanx2","meanx3","meanx4","meanx5","meanx6"),
                          name="varMeans" )
facMeans     <- mxMatrix( type="Full", nrow=1, ncol=1, values=0, free=F, name="facMeans" )
expCov       <- mxAlgebra( expression= facLoadings %&% facVariances + resVariances,
                           name="expCov" )
expMean      <- mxAlgebra( expression= varMeans + t(facLoadings %*% facMeans),
                           name="expMean" )
exp          <- mxExpectationNormal( covariance="expCov", means="expMean",
                                     dimnames=manifestVars)
funML        <- mxFitFunctionML()

oneFactorModel <- mxModel("Common Factor Model Matrix Specification",
                          dataRaw, facLoads, facVars, resVars, varMeans, facMeans,
                          expCov, expMean, exp, funML)

oneFactorFit<-mxRun(oneFactorModel)
```

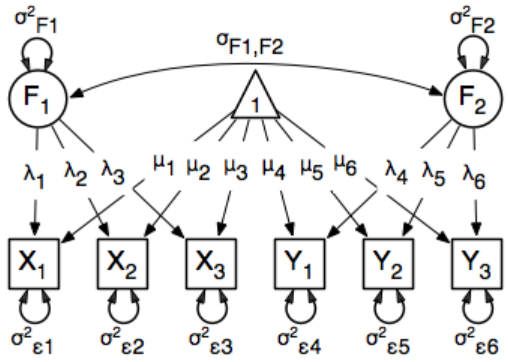
The first `mxMatrix` statement declares a `Full` **6x1** matrix of factor loadings to be estimated, called "facLoadings". We fix the first factor loading to 1 for identification. Even though we specify just one start value of 1 which is recycled for each of the elements in the matrix, it becomes the fixed value for the first factor loading and the start value for the other factor loadings. The second `mxMatrix` is a `symmetric` **1x1** which estimates the variance of the factor, named "facVariances". The third `mxMatrix` is a `Diag` **6x6** matrix for the residual variances, named "resVariances". The fourth `mxMatrix` is a `Full` **1x6** matrix of free elements for the means of the observed variables, called "varMeans". The fifth `mxMatrix` is a `Full` **1x1** matrix with a fixed value of zero for the factor mean, named "facMeans".

We then use two algebra statements to work out the expected means and covariance matrices. Note that the formula's for the expression of the expected covariance and the expected mean vector map directly on to the mathematical equations. The arguments for the `mxExpectationNormal` function now refer to these algebras for the expected covariance and expected means. The `dimnames` are used to map them onto the observed variables. The fit function compares the expectation and the observation (i.e. data) to optimize free parameters.

3.2.2 Two Factor Model

The common factor model can be extended to include multiple latent variables. The model for any person and path diagram of the common factor model for a set of variables x_1 - x_3 and y_1 - y_3 are given below.

$$x_{ij} = \mu_j + \lambda_j * \eta_{1i} + \epsilon_{ij}$$
$$y_{ij} = \mu_j + \lambda_j * \eta_{2i} + \epsilon_{ij}$$



Our model contains 21 parameters (six manifest variances, six manifest means, six factor loadings, two factor variances and one factor covariance), but each factor requires one identification constraint. Like in the common factor model above, we will constrain one factor loading for each factor to a value of one. As such, this model contains 19 parameters. The means and covariance matrix for six observed variables contain 27 degrees of freedom, and thus our model contains 8 degrees of freedom.

The data for the two factor model can be found in the `myFAData` files introduced in the common factor model. For this model, we will select three *x* variables (`x1-x3`) and three *y* variables (`y1-y3`).

```
twoFactorRaw <- myFADataRaw[,c("x1", "x2", "x3", "y1", "y2", "y3")]

twoFactorCov <- myFADataCov[c("x1","x2","x3","y1","y2","y3"),
                            c("x1","x2","x3","y1","y2","y3")]

twoFactorMeans <- myFADataMeans[c(1:3,7:9)]
```

Specifying the two factor model is virtually identical to the single factor case. The `mxData` function has been changed to reference the appropriate data, but is identical in usage. We've added a second latent variable, so the **A** and **S** matrices are now of order **8x8**. Similarly, the **F** matrix is now of order **6x8** and the **M** matrix of order **1x8**. The `mxExpectationRAM` has not changed. The code for our two factor model looks like this:

```
dataRaw      <- mxData( observed=myFADataRaw, type="raw" )
matrA        <- mxMatrix( type="Full", nrow=8, ncol=8,
                          free=  c(F,F,F,F,F,F,F,F,
                                   F,F,F,F,F,F,T,F,
                                   F,F,F,F,F,F,T,F,
                                   F,F,F,F,F,F,F,F,
                                   F,F,F,F,F,F,F,T,
                                   F,F,F,F,F,F,F,T,
                                   F,F,F,F,F,F,F,F,
                                   F,F,F,F,F,F,F,F),
                          values=c(0,0,0,0,0,0,1,0,
                                   0,0,0,0,0,0,1,0,
                                   0,0,0,0,0,0,1,0,
                                   0,0,0,0,0,0,0,1,
                                   0,0,0,0,0,0,0,1,
                                   0,0,0,0,0,0,0,1,
                                   0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0),
                          labels=c(NA,NA,NA,NA,NA,NA,"l1",NA,
                                   NA,NA,NA,NA,NA,NA,"l2",NA,
                                   NA,NA,NA,NA,NA,NA,"l3",NA,
                                   NA,NA,NA,NA,NA,NA,NA,"l4",
                                   NA,NA,NA,NA,NA,NA,NA,"l5",
                                   NA,NA,NA,NA,NA,NA,NA,"l6",
```

```
                                    NA,NA,NA,NA,NA,NA,NA,NA,
                                    NA,NA,NA,NA,NA,NA,NA,NA),
                          byrow=TRUE, name="A" )
matrS         <- mxMatrix( type="Symm", nrow=8, ncol=8,
                          free=  c(T,F,F,F,F,F,F,F,
                                   F,T,F,F,F,F,F,F,
                                   F,F,T,F,F,F,F,F,
                                   F,F,F,T,F,F,F,F,
                                   F,F,F,F,T,F,F,F,
                                   F,F,F,F,F,T,F,F,
                                   F,F,F,F,F,F,T,T,
                                   F,F,F,F,F,F,T,T),
                          values=c(1,0,0,0,0,0,0,0,
                                   0,1,0,0,0,0,0,0,
                                   0,0,1,0,0,0,0,0,
                                   0,0,0,1,0,0,0,0,
                                   0,0,0,0,1,0,0,0,
                                   0,0,0,0,0,1,0,0,
                                   0,0,0,0,0,0,1,.5,
                                   0,0,0,0,0,0,.5,1),
                          labels=c("e1",NA,  NA,  NA,  NA,  NA,  NA,  NA,
                                   NA, "e2", NA,  NA,  NA,  NA,  NA,  NA,
                                   NA,  NA, "e3", NA,  NA,  NA,  NA,  NA,
                                   NA,  NA,  NA, "e4", NA,  NA,  NA,  NA,
                                   NA,  NA,  NA,  NA, "e5", NA,  NA,  NA,
                                   NA,  NA,  NA,  NA,  NA, "e6", NA,  NA,
                                   NA,  NA,  NA,  NA,  NA,  NA,"varF1","cov",
                                   NA,  NA,  NA,  NA,  NA,  NA,"cov","varF2"),
                          byrow=TRUE, name="S" )
matrF         <- mxMatrix( type="Full", nrow=6, ncol=8,
                          free=FALSE,
                          values=c(1,0,0,0,0,0,0,0,
                                   0,1,0,0,0,0,0,0,
                                   0,0,1,0,0,0,0,0,
                                   0,0,0,1,0,0,0,0,
                                   0,0,0,0,1,0,0,0,
                                   0,0,0,0,0,1,0,0),
                          byrow=TRUE, name="F" )
matrM         <- mxMatrix( type="Full", nrow=1, ncol=8,
                          free=c(T,T,T,T,T,T,F,F),
                          values=c(1,1,1,1,1,1,0,0),
                          labels=c("meanx1","meanx2","meanx3",
                                   "meanx4","meanx5","meanx6",NA,NA),
                          name="M" )
exp           <- mxExpectationRAM("A","S","F","M",
                                  dimnames=c(manifestVars, latentVars))
funML         <- mxFitFunctionML()

twoFactorModel <- mxModel("Two Factor Model Matrix Specification",
                          dataRaw, matrA, matrS, matrF, matrM, exp, funML)
```

The four `mxMatrix` functions have changed slightly to accomodate the changes in the model. The **A** matrix, shown below, is used to specify the regressions of the manifest variables on the factors. The first three manifest variables (`"x1"-"x3"`) are regressed on `"F1"`, and the second three manifest variables (`"y1"-"y3"`) are regressed on `"F2"`. We must again constrain the model to identify and scale the latent variables, which we do by constraining the first loading for each latent variable to a value of one.

```
# asymmetric paths
matrA        <- mxMatrix( type="Full", nrow=8, ncol=8,
                          free=  c(F,F,F,F,F,F,F,F,
                                   F,F,F,F,F,F,T,F,
                                   F,F,F,F,F,F,T,F,
                                   F,F,F,F,F,F,F,F,
                                   F,F,F,F,F,F,F,T,
                                   F,F,F,F,F,F,F,T,
                                   F,F,F,F,F,F,F,F,
                                   F,F,F,F,F,F,F,F),
                          values=c(0,0,0,0,0,0,1,0,
                                   0,0,0,0,0,0,1,0,
                                   0,0,0,0,0,0,1,0,
                                   0,0,0,0,0,0,0,1,
                                   0,0,0,0,0,0,0,1,
                                   0,0,0,0,0,0,0,1,
                                   0,0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0,0),
                          labels=c(NA,NA,NA,NA,NA,NA,"l1",NA,
                                   NA,NA,NA,NA,NA,NA,"l2",NA,
                                   NA,NA,NA,NA,NA,NA,"l3",NA,
                                   NA,NA,NA,NA,NA,NA,NA,"l4",
                                   NA,NA,NA,NA,NA,NA,NA,"l5",
                                   NA,NA,NA,NA,NA,NA,NA,"l6",
                                   NA,NA,NA,NA,NA,NA,NA,NA,
                                   NA,NA,NA,NA,NA,NA,NA,NA),
                          byrow=TRUE, name="A" )
```

The **S** matrix has an additional row and column, and two additional parameters. For the two factor model, we must add a variance term for the second latent variable and a covariance between the two latent variables.

```
# symmetric paths
matrS        <- mxMatrix( type="Symm", nrow=8, ncol=8,
                          free=  c(T,F,F,F,F,F,F,F,
                                   F,T,F,F,F,F,F,F,
                                   F,F,T,F,F,F,F,F,
                                   F,F,F,T,F,F,F,F,
                                   F,F,F,F,T,F,F,F,
                                   F,F,F,F,F,T,F,F,
                                   F,F,F,F,F,F,T,T,
                                   F,F,F,F,F,F,T,T),
                          values=c(1,0,0,0,0,0,0,0,
                                   0,1,0,0,0,0,0,0,
                                   0,0,1,0,0,0,0,0,
                                   0,0,0,1,0,0,0,0,
                                   0,0,0,0,1,0,0,0,
                                   0,0,0,0,0,1,0,0,
                                   0,0,0,0,0,0,1,.5,
                                   0,0,0,0,0,0,.5,1),
                          labels=c("e1",NA,  NA,  NA,  NA,  NA,  NA,  NA,
                                   NA, "e2", NA,  NA,  NA,  NA,  NA,  NA,
                                   NA,  NA, "e3", NA,  NA,  NA,  NA,  NA,
                                   NA,  NA,  NA, "e4", NA,  NA,  NA,  NA,
                                   NA,  NA,  NA,  NA, "e5", NA,  NA,  NA,
                                   NA,  NA,  NA,  NA,  NA, "e6", NA,  NA,
                                   NA,  NA,  NA,  NA,  NA,  NA,"varF1","cov",
                                   NA,  NA,  NA,  NA,  NA,  NA,"cov","varF2"),
                          byrow=TRUE, name="S" )
```

The **F** and **M** matrices contain only minor changes. The **F** matrix is now of order 6x8, but the additional column is simply a column of zeros. The **M** matrix contains an additional column (with only a single row), which contains the mean of the second latent variable. As this model does not contain a parameter for that latent variable, this mean is constrained to zero.

The model is now ready to run using the `mxRun` function, and the output of the model can be accessed from the `$output` slot of the resulting model. A summary of the output can be reached using `summary()`.

These models may also be specified using paths instead of matrices. See *Factor Analysis, Path Specification* for path specification of these models.

3.3 Time Series, Matrix Specification

This example will demonstrate a growth curve model using RAM specified matrices. As with previous examples, this application is split into two files, one each raw and covariance data. These examples can be found in the following files:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/LatentGrowthCurveModel_MatrixRaw.R

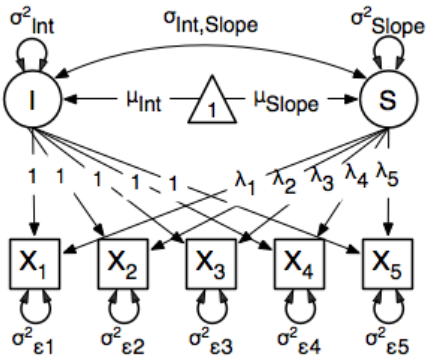
Parallel versions of this example, using path-centric specification of models rather than matrices, can be found here:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/LatentGrowthCurveModel_PathRaw.R

3.3.1 Latent Growth Curve Model

The latent growth curve model is a variation of the factor model for repeated measurements. For a set of manifest variables x_{i1} - x_{i5} measured at five discrete times for people indexed by the letter *i*, the growth curve model can be expressed both algebraically and via a path diagram as shown here:

$$x_{ij} = Intercept_i + \lambda_j * Slope_i + \epsilon_i$$



The values and specification of the λ parameters allow for alterations to the growth curve model. This example will utilize a linear growth curve model, so we will specify λ to increase linearly with time. If the observations occur at regular intervals in time, then λ can be specified with any values increasing at a constant rate. For this example, we will use [0, 1, 2, 3, 4] so that the intercept represents scores at the first measurement occasion, and the slope represents the rate of change per measurement occasion. Any linear transformation of these values can be used for linear growth curve models.

Our model for any number of variables contains 6 free parameters; two factor means, two factor variances, a factor covariance and a (constant) residual variance for the manifest variables. Our data contains five manifest variables, and so the covariance matrix and means vector contain 20 degrees of freedom. Thus, the linear growth curve model fit to these data has 14 degrees of freedom.

Data

The first step to running our model is to import data. The code below is used to import both raw data and a covariance matrix and means vector, either of which can be used for our growth curve model. This data contains five variables, which are repeated measurements of the same variable. As growth curve models make specific hypotheses about the variances of the manifest variables, correlation matrices generally aren't used as data for this model.

```
data(myLongitudinalData)

myLongitudinalDataCov<-matrix(
    c(6.362, 4.344, 4.915,  5.045,  5.966,
      4.344, 7.241, 5.825,  6.181,  7.252,
      4.915, 5.825, 9.348,  7.727,  8.968,
      5.045, 6.181, 7.727, 10.821, 10.135,
      5.966, 7.252, 8.968, 10.135, 14.220), nrow=5,
dimnames=list( c("x1","x2","x3","x4","x5"), c("x1","x2","x3","x4","x5")) )

myLongitudinalDataMeans <- c(9.864, 11.812, 13.612, 15.317, 17.178)
names(myLongitudinalDataMeans) <- c("x1","x2","x3","x4","x5")
```

Model Specification

The following code contains all of the components of our model. Before running a model, the OpenMx library must be loaded into R using either the `require()` or `library()` function. All objects required for estimation (data, matrices, an expectation function, and a fit function) are specified. This code uses the `mxModel` function to create an `MxModel` object, which we will then run.

```
require(OpenMx)

dataRaw      <- mxData( observed=myLongitudinalData, type="raw" )
matrA        <- mxMatrix( type="Full", nrow=7, ncol=7,
                          free=F,
                          values=c(0,0,0,0,0,1,0,
                                   0,0,0,0,0,1,1,
                                   0,0,0,0,0,1,2,
                                   0,0,0,0,0,1,3,
                                   0,0,0,0,0,1,4,
                                   0,0,0,0,0,0,0,
                                   0,0,0,0,0,0,0),
                          byrow=TRUE, name="A" )
matrS        <- mxMatrix( type="Symm", nrow=7, ncol=7,
                          free=  c(T,F,F,F,F,F,F,
                                   F,T,F,F,F,F,F,
                                   F,F,T,F,F,F,F,
                                   F,F,F,T,F,F,F,
                                   F,F,F,F,T,F,F,
                                   F,F,F,F,F,T,T,
                                   F,F,F,F,F,T,T),
                          values=c(0,0,0,0,0, 0, 0,
                                   0,0,0,0,0, 0, 0,
                                   0,0,0,0,0, 0, 0,
                                   0,0,0,0,0, 0, 0,
                                   0,0,0,0,0, 0, 0,
                                   0,0,0,0,0, 1,.5,
                                   0,0,0,0,0,.5, 1),
                          labels=c("residual", NA, NA, NA, NA, NA, NA,
                                   NA, "residual", NA, NA, NA, NA, NA,
```

```
                                    NA, NA, "residual", NA, NA, NA, NA,
                                    NA, NA, NA, "residual", NA, NA, NA,
                                    NA, NA, NA, NA, "residual", NA, NA,
                                    NA, NA, NA, NA, NA, "vari", "cov",
                                    NA, NA, NA, NA, NA, "cov", "vars"),
                           byrow= TRUE, name="S" )
matrF          <- mxMatrix( type="Full", nrow=5, ncol=7,
                           free=F,
                           values=c(1,0,0,0,0,0,0,
                                    0,1,0,0,0,0,0,
                                    0,0,1,0,0,0,0,
                                    0,0,0,1,0,0,0,
                                    0,0,0,0,1,0,0),
                           byrow=T, name="F" )
matrM          <- mxMatrix( type="Full", nrow=1, ncol=7,
                           free=c(F,F,F,F,F,T,T), values=c(0,0,0,0,0,1,1),
                           labels=c(NA,NA,NA,NA,NA,"meani","means"), name="M" )
exp            <- mxExpectationRAM("A","S","F","M",
                                   dimnames=c(names(myLongitudinalData),"intercept","slope"))
funML          <- mxFitFunctionML()

growthCurveModel <- mxModel("Linear Growth Curve Model Matrix Specification",
                            dataRaw, matrA, matrS, matrF, matrM, exp, funML)
```

The model begins with a name, in this case "Linear Growth Curve Model Matrix Specification". If the first argument is an object containing an `MxModel` object, then the model created by the `mxModel` function will contain all of the named entites in the referenced model object.

Data is supplied with the `mxData` function. This example uses raw data, but the `mxData` function in the code above could be replaced with the function below to include covariance data.

```
dataCov        <- mxData( myLongitudinalDataCov, type="cov", numObs=500,
                          means=myLongitudinalDataMeans )
```

The four `mxMatrix` functions define the **A**, **S**, **F** and **M** matrices used in RAM specification of models. In all four matrices, the first five rows or columns of any matrix represent the five manifest variables, the sixth the latent intercept variable, and the seventh the slope. The **A** and **S** matrices are of order **7x7**, the **F** matrix of order **5x7**, and the **M** matrix **1x7**.

The **A** matrix specifies all of the assymetric paths or regressions among variables. The only assymmetric paths in our model regress the manifest variables on the latent intercept and slope with fixed values. The regressions of the manifest variables on the intercept are in the first five rows and sixth column of the **A** matrix, all of which have a fixed value of one. The regressions of the manifest variables on the slope are in the first five rows and seventh column of the **A** matrix with fixed values in this series: [0, 1, 2, 3, 4].

```
# asymmetric paths
matrA          <- mxMatrix( type="Full", nrow=7, ncol=7,
                           free=F,
                           values=c(0,0,0,0,0,1,0,
                                    0,0,0,0,0,1,1,
                                    0,0,0,0,0,1,2,
                                    0,0,0,0,0,1,3,
                                    0,0,0,0,0,1,4,
                                    0,0,0,0,0,0,0,
                                    0,0,0,0,0,0,0),
                           byrow=TRUE, name="A" )
```

The **S** matrix specifies all of the symmetric paths among our variables, representing the variances and covariances in our model. The five manifest variables do not have any covariance parameters with any other variables, and all

are restricted to have the same residual variance. This variance term is constrained to equality by specifying five free parameters and giving all five parameters the same label `residual`. The variances and covariance of the latent variables are included as free parameters in the sixth and seventhns rows and columns of this matrix as well.

```
# symmetric paths
matrS        <- mxMatrix( type="Symm", nrow=7, ncol=7,
                          free=  c(T,F,F,F,F,F,F,
                                   F,T,F,F,F,F,F,
                                   F,F,T,F,F,F,F,
                                   F,F,F,T,F,F,F,
                                   F,F,F,F,T,F,F,
                                   F,F,F,F,F,T,T,
                                   F,F,F,F,F,T,T),
                          values=c(0,0,0,0,0, 0, 0,
                                   0,0,0,0,0, 0, 0,
                                   0,0,0,0,0, 0, 0,
                                   0,0,0,0,0, 0, 0,
                                   0,0,0,0,0, 0, 0,
                                   0,0,0,0,0, 1,.5,
                                   0,0,0,0,0,.5, 1),
                          labels=c("residual", NA, NA, NA, NA, NA, NA,
                                   NA, "residual", NA, NA, NA, NA, NA,
                                   NA, NA, "residual", NA, NA, NA, NA,
                                   NA, NA, NA, "residual", NA, NA, NA,
                                   NA, NA, NA, NA, "residual", NA, NA,
                                   NA, NA, NA, NA, NA, "vari", "cov",
                                   NA, NA, NA, NA, NA, "cov", "vars"),
                          byrow= TRUE, name="S" )
```

The third matrix in our RAM model is the **F** or filter matrix. This is used to "filter" the latent variables from the expected covariance of the observed data. The **F** matrix will always contain the same number of rows as manifest variables and columns as total (manifest and latent) variables. If the manifest variables in the **A** and **S** matrices precede the latent variables are in the same order as the data, then the **F** matrix will be the horizontal adhesion of an identity matrix and a zero matrix. This matrix contains no free parameters, and is made with the `mxMatrix` function below.

```
# filter matrix
matrF        <- mxMatrix( type="Full", nrow=5, ncol=7,
                          free=F,
                          values=c(1,0,0,0,0,0,0,
                                   0,1,0,0,0,0,0,
                                   0,0,1,0,0,0,0,
                                   0,0,0,1,0,0,0,
                                   0,0,0,0,1,0,0),
                          byrow=T, name="F" )
```

The final matrix in our RAM model is the **M** or means matrix, which specifies the means and intercepts of the variables in the model. While the manifest variables have expected means in our model, these expected means are entirely dependent on the means of the intercept and slope factors. In the **M** matrix below, the manifest variables are given fixed intercepts of zero while the latent variables are each given freely estimated means with starting values of 1 and labels of `"meani"` and `"means"`

```
# means
matrM        <- mxMatrix( type="Full", nrow=1, ncol=7,
                          free=c(F,F,F,F,F,T,T), values=c(0,0,0,0,0,1,1),
                          labels=c(NA,NA,NA,NA,NA,"meani","means"), name="M" )
```

The last pieces of our model are the `mxExpectaionRAM` and `mxFitFunctionML` functions, which define both how the specified matrices combine to create the expected covariance matrix of the data, and the fit function to be

minimized, respectively. As covered in earlier examples, the expected covariance matrix for a RAM model is defined as:

$$ExpCovariance = F * (I - A)^{-1} * S * ((I - A)^{-1})' * F'$$

The expected means are defined as:

$$ExpMean = F * (I - A)^{-1} * M$$

The free parameters in the model can then be estimated using maximum likelihood for covariance and means data, and full information maximum likelihood for raw data. The **M** matrix is required both for raw data and for covariance or correlation data that includes a means vector. The `mxExpectationRAM` function takes four arguments, which are the names of the **A**, **S**, **F** and **M** matrices in your model. The `mxFitFunctionML` function often takes no arguments.

The model is now ready to run using the `mxRun` function, and the output of the model can be accessed from the `$output` slot of the resulting model. A summary of the output can be reached using `summary()`.

```
growthCurveFit <- mxRun(growthCurveModel)

growthCurveFit$output
summary(growthCurveFit)
```

These models may also be specified using paths instead of matrices. See *Time Series, Path Specification* for path specification of these models.

3.4 Multiple Groups, Matrix Specification

An important aspect of structural equation modeling is the use of multiple groups to compare means and covariances structures between any two (or more) data groups, for example males and females, different ethnic groups, ages etc. Other examples include groups which have different expected covariances matrices as a function of parameters in the model, and need to be evaluated together for the parameters to be identified.

The example includes the heterogeneity model as well as its submodel, the homogeneity model and is available in the following file:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/BivariateHeterogeneity_MatrixRaw.R

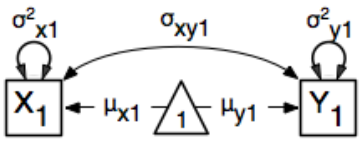
A parallel version of this example, using paths specification of models rather than matrices, can be found here:

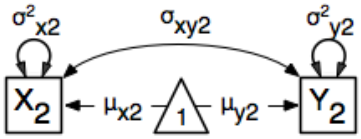
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/BivariateHeterogeneity_PathRaw.R

3.4.1 Heterogeneity Model

We will start with a basic example here, building on modeling means and variances in a saturated model. Assume we have two groups and we want to test whether they have the same mean and covariance structure.

The path diagram of the heterogeneity model for a set of variables x and y are given below.





Data

For this example we simulated two datasets (`xy1` and `xy2`) each with zero means and unit variances, one with a correlation of 0.5, and the other with a correlation of 0.4 with 1000 subjects each. We use the `mvrnorm` function in the `MASS` package, which takes three arguments: `Sample Size`, `Means`, `Covariance Matrix`). We check the means and covariance matrix in R and provide `dimnames` for the dataframe. See attached R code for simulation and data summary.

```
#Simulate Data
require(MASS)
#group 1
set.seed(200)
xy1 <- mvrnorm (1000, c(0,0), matrix(c(1,.5,.5,1),2,2))
#group 2
set.seed(200)
xy2 <- mvrnorm (1000, c(0,0), matrix(c(1,.4,.4,1),2,2))

#Print Descriptive Statistics
selVars <- c('X','Y')
summary(xy1)
cov(xy1)
dimnames(xy1) <- list(NULL, selVars)
summary(xy2)
cov(xy2)
dimnames(xy2) <- list(NULL, selVars)
```

Model Specification

As before, we include the OpenMx package using a `require` statement. We first fit a heterogeneity model, allowing differences in both the mean and covariance structure of the two groups. As we are interested whether the two structures can be equated, we have to specify the models for the two groups, named `group1` and `group2` within another model. The structure of the job thus look as follows, with two `mxModel` commands as arguments of another `mxModel` command. Note that `mxModel` commands are unlimited in the number of arguments.

For each of the groups, we fit a saturated model, using a Cholesky decomposition to generate the expected covariance matrix and a row vector for the expected means. Note that we have specified different labels for all the free elements, in the two `mxModel` statements. For more details, see example 1.

```
require(OpenMx)
```

```
#Fit Heterogeneity Model

chol1       <- mxMatrix( type="Lower", nrow=2,  ncol=2,
                         free=T, values=.5, labels=c("Ch11","Ch21","Ch31"), name="chol1" )
expCov1     <- mxAlgebra( expression=chol1 %*% t(chol1), name="expCov1" )
expMean1    <- mxMatrix( type="Full", nrow=1, ncol=2,
                         free=T, values=c(0,0), labels=c("mX1","mY1"), name="expMean1" )
dataRaw1    <- mxData( xy1, type="raw" )
exp1        <- mxExpectationNormal( covariance="expCov1", means="expMean1", selVars)
funML       <- mxFitFunctionML()
model1      <- mxModel("group1",
                       dataRaw1, chol1, expCov1, expMean1, exp1, funML)

chol2       <- mxMatrix( type="Lower", nrow=2,  ncol=2,
                         free=T, values=.5, labels=c("Ch12","Ch22","Ch32"), name="chol2" )
expCov2     <- mxAlgebra( expression=chol2 %*% t(chol2), name="expCov2" )
expMean2    <- mxMatrix( type="Full", nrow=1, ncol=2,
                         free=T, values=c(0,0), labels=c("mX2","mY2"), name="expMean2" )
dataRaw2    <- mxData( xy2, type="raw" )
exp2        <- mxExpectationNormal( covariance="expCov2", means="expMean2", selVars)
funML       <- mxFitFunctionML()
model2      <- mxModel("group2",
                       dataRaw2, chol2, expCov2, expMean2, exp2, funML)

fun          <- mxFitFunctionMultigroup(c("group1.fitfunction", "group2.fitfunction"))

bivHetModel  <- mxModel("bivariate Heterogeneity Matrix Specification",
                         model1, model2, fun )
```

We estimate five parameters (two means, two variances, one covariance) per group for a total of 10 free parameters. We cut the `Labels matrix:` parts from the output generated with `bivHetModel$group1$matrices` and `bivHetModel$group2$matrices`.

```
in group1                          in group2
    $S                                 $S
            X      Y                          X      Y
    X  "Ch11"     NA                   X  "Ch12"     NA
    Y  "Ch21"  "Ch22"                  Y  "Ch22" "Ch32"

    $M                                 $M
            X      Y                          X      Y
    [1,] "mX1" "mY1"                  [1,] "mX2" "mY2"
```

To evaluate both models together, we use an `mxFitFunctionMultigroup` command that adds up the values of the fit functions of the two groups.

```
fun          <- mxFitFunctionMultigroup(c("group1.fitfunction", "group2.fitfunction"))
```

Model Fitting

The `mxRun` command is required to actually evaluate the model. Note that we have adopted the following notation of the objects. The result of the `mxModel` command ends in "Model"; the result of the `mxRun` command ends in "Fit". Of course, these are just suggested naming conventions.

```
bivHetFit <- mxRun(bivHetModel)
```

A variety of output can be printed. We chose here to print the expected means and covariance matrices for the two groups and the likelihood of data given the model. The `mxEval` command takes any R expression, followed by the

fitted model name. Given that the model `bivHetFit` included two models (group1 and group2), we need to use the two level names, i.e. `group1.EM1` to refer to the objects in the correct model.

```
expMean1Het <- mxEval(group1.expMean1, bivHetFit)
expMean2Het <- mxEval(group2.expMean2, bivHetFit)
expCov1Het  <- mxEval(group1.expCov1, bivHetFit)
expCov2Het  <- mxEval(group2.expCov2, bivHetFit)
LLHet       <- bivHetFit$output$fit
```

3.4.2 Homogeneity Model: a Submodel

Next, we fit a model in which the mean and covariance structure of the two groups are equated to one another, to test whether there are significant differences between the groups. Rather than having to specify the entire model again, we copy the previous model `bivHetModel` into a new model `bivHomModel` to represent homogeneous structures.

```
#Fit Homogeneity Model
bivHomModel <- bivHetModel
```

As elements in matrices can be equated by assigning the same label, we now have to equate the labels of the free parameters in group 1 to the labels of the corresponding elements in group 2. This can be done by referring to the relevant matrices using the `ModelName$MatrixName` syntax, followed by `$labels`. Note that in the same way, one can refer to other arguments of the objects in the model. Here we assign the labels from group 1 to the labels of group 2, separately for the Cholesky matrices used for the expected covariance matrices and for the expected means vectors.

```
bivHomModel[['group2.chol2']]$labels <- bivHomModel[['group1.chol1']]$labels
bivHomModel[['group2.expMean2']]$labels <- bivHomModel[['group1.expMean1']]$labels
```

The specification for the submodel is reflected in the names of the labels which are now equal for the corresponding elements of the mean and covariance matrices, as below.

```
in group1
    $S
             X       Y
    X  "Ch11"     NA
    Y  "Ch21" "CH31"

    $M
             X       Y
    [1,] "mX1" "mY1"
```

```
in group2
    $S
             X       Y
    X  "Ch11"     NA
    Y  "Ch21" "Ch31"

    $M
             X       Y
    [1,] "mX1" "mY1"
```

We can produce similar output for the submodel, i.e. expected means and covariances and likelihood, the only difference in the code being the model name. Note that as a result of equating the labels, the expected means and covariances of the two groups should be the same.

```
bivHomFit <- mxRun(bivHomModel)
expMean1Hom <- mxEval(group1.expMean1, bivHomFit)
expMean2Hom <- mxEval(group2.expMean2, bivHomFit)
expCov1Hom  <- mxEval(group1.expCov1, bivHomFit)
expCov2Hom  <- mxEval(group2.expCov2, bivHomFit)
LLHom       <- bivHomFit$output$fit
```

Finally, to evaluate which model fits the data best, we generate a likelihood ratio test as the difference between -2 times the log-likelihood of the homogeneity model and -2 times the log-likelihood of the heterogeneity model. This statistic is asymptotically distributed as a Chi-square, which can be interpreted with the difference in degrees of freedom of the two models.

```
Chi <- LLHom-LLHet
LRT <- rbind(LLHet,LLHom,Chi)
LRT
```

These models may also be specified using paths instead of matrices. See *Multiple Groups, Path Specification* for path specification of these models.

3.5 Genetic Epidemiology, Matrix Specification

Mx is probably most popular in the behavior genetics field, as it was conceived with genetic models in mind, which rely heavily on multiple groups. We introduce here an OpenMx script for the basic genetic model in genetic epidemiologic research, the ACE model. This model assumes that the variability in a phenotype, or observed variable, of interest can be explained by differences in genetic and environmental factors, with A representing additive genetic factors, C shared/common environmental factors and E unique/specific environmental factors (see Neale & Cardon 1992, for a detailed treatment). To estimate these three sources of variance, data have to be collected on relatives with different levels of genetic and environmental similarity to provide sufficient information to identify the parameters. One such design is the classical twin study, which compares the similarity of identical (monozygotic, MZ) and fraternal (dizygotic, DZ) twins to infer the role of **A**, **C** and **E**.

The example starts with the ACE model and includes one submodel, the AE model. It is available in the following file:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/UnivariateTwinAnalysis_MatrixRaw.R

A parallel version of this example, using path specification of models rather than matrices, can be found here:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/UnivariateTwinAnalysis_PathRaw.R

3.5.1 ACE Model: a Twin Analysis

Data

Let us assume you have collected data on a large sample of twin pairs for your phenotype of interest. For illustration purposes, we use Australian data on body mass index (BMI) which are saved in a text file 'myTwinData.txt'. We use R to read the data into a data.frame and to create two subsets of the data for MZ females (*mzData*) and DZ females (*dzData*) respectively with the code below.

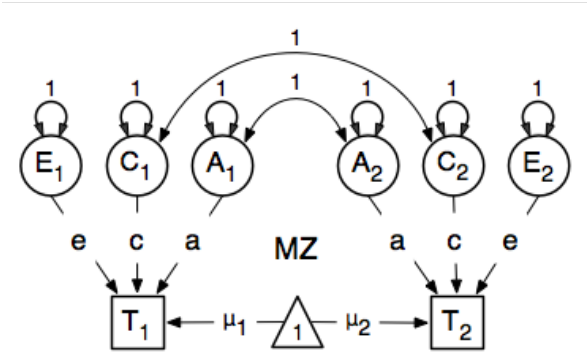
```
require(OpenMx)
require(psych)

# Load Data
data(twinData)
describe(twinData, skew=F)

# Select Variables for Analysis
Vars      <- 'bmi'
nv        <- 1       # number of variables
ntv       <- nv*2    # number of total variables
selVars   <- paste(Vars,c(rep(1,nv),rep(2,nv)),sep="")    #c('bmi1','bmi2')

# Select Data for Analysis
mzData    <- subset(twinData, zyg==1, selVars)
dzData    <- subset(twinData, zyg==3, selVars)
```

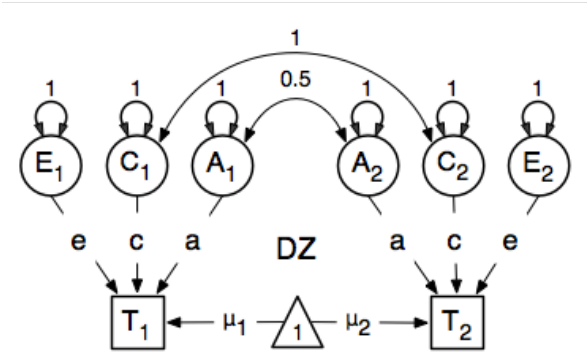
```
# Generate Descriptive Statistics
colMeans(mzData,na.rm=TRUE)
colMeans(dzData,na.rm=TRUE)
cov(mzData,use="complete")
cov(dzData,use="complete")
```

Model Specification

There are a variety of ways to set up the ACE model. The most commonly used approach in Mx is to specify three matrices for each of the three sources of variance. The matrix **a** represents the additive genetic path a, the **c** matrix is used for the shared environmental path c and the matrix **e** for the unique environmental path e. The expected variances and covariances between member of twin pairs are typically expressed in variance components (or the square of the path coefficients, i.e. a^2, c^2 and e^2). These quantities can be calculated using matrix algebra, by multiplying the **a** matrix by its transpose **t(a)**, and are called **A**, **C** and **E** respectively. Note that the transpose is not strictly needed in the univariate case, but will allow easier transition to the multivariate case.

We then use matrix algebra again to add the relevant matrices corresponding to the expectations for each of the statistics of the expected covariance matrix. The R functions `cbind` and `rbind` are used to concatenate the resulting matrices in the appropriate way. The expectations can be derived from the path diagrams for MZ and DZ twins. The expectation for the variance of either twin 1 or twin 2 to be included on the diagonal elements is the sum of the variance components (**A** + **C** + **E**). The predicted covariance between MZ twins is a function of their shared genes and environments (**A** + **C**). DZ twins on the other hand share only half their genes on average, but the shared environment is by definition also shared completely (.5* **A** + **C**).





Note that in R, lower and upper case names are distinguishable so we are using lower case letters for the matrices representing path coefficients **a**, **c** and **e**, rather than **X**, **Y** and **Z** that classic Mx users are familiar with. We continue to use the same upper case letters for matrices representing variance components **A**, **C** and **E**, corresponding to additive genetic (co)variance, shared environmental (co)variance and unique environmental (co)variance respectively,

calculated as the square of the path coefficients.

Let's go through each of the matrices step by step. First, we start with the `require(OpenMx)` statement. We include the full code here. As MZ and DZ have to be evaluated together, the models for each will be arguments of a bigger model. Given the models for the MZ and the DZ group look rather similar, we start by specifying all the common elements and the model-specific elements which will then be included in the two models (*modelMZ* and *modelDZ*) for each of the twin types, defined in separate `mxModel` commands. The combined model (*AceModel*) will then include the individual R objects, the MZ and DZ models with their respective R objects as well as the data and a fit function to combine them.

```
require(OpenMx)

# Set Starting Values
svMe      <- 20      # start value for means
svPa      <- .6      # start value for path coefficients (sqrt(variance/#ofpaths))

# ACE Model
# Matrices declared to store a, d, and e Path Coefficients
pathA     <- mxMatrix( type="Full", nrow=nv, ncol=nv,
                       free=TRUE, values=svPa, label="a11", name="a" )
pathC     <- mxMatrix( type="Full", nrow=nv, ncol=nv,
                       free=TRUE, values=svPa, label="c11", name="c" )
pathE     <- mxMatrix( type="Full", nrow=nv, ncol=nv,
                       free=TRUE, values=svPa, label="e11", name="e" )

# Matrices generated to hold A, C, and E computed Variance Components
covA      <- mxAlgebra( expression=a %*% t(a), name="A" )
covC      <- mxAlgebra( expression=c %*% t(c), name="C" )
covE      <- mxAlgebra( expression=e %*% t(e), name="E" )

# Algebra to compute total variances
covP      <- mxAlgebra( expression=A+C+E, name="V" )

# Algebra for expected Mean and Variance/Covariance Matrices in MZ & DZ twins
meanG     <- mxMatrix( type="Full", nrow=1, ncol=ntv,
                       free=TRUE, values=svMe, label="mean", name="expMean" )
covMZ     <- mxAlgebra( expression=rbind( cbind(V, A+C),
                                          cbind(A+C, V)), name="expCovMZ" )
covDZ     <- mxAlgebra( expression=rbind( cbind(V, 0.5%x%A+ C),
                                          cbind(0.5%x%A+C , V)), name="expCovDZ" )

# Data objects for Multiple Groups
dataMZ    <- mxData( observed=mzData, type="raw" )
dataDZ    <- mxData( observed=dzData, type="raw" )

# Objective objects for Multiple Groups
expMZ     <- mxExpectationNormal( covariance="expCovMZ", means="expMean",
                                  dimnames=selVars )
expDZ     <- mxExpectationNormal( covariance="expCovDZ", means="expMean",
                                  dimnames=selVars )
funML     <- mxFitFunctionML()

# Combine Groups
pars      <- list( pathA, pathC, pathE, covA, covC, covE, covP )
modelMZ   <- mxModel( pars, meanG, covMZ, dataMZ, expMZ, funML, name="MZ" )
modelDZ   <- mxModel( pars, meanG, covDZ, dataDZ, expDZ, funML, name="DZ" )
fitML     <- mxFitFunctionMultigroup(c("MZ.fitfunction","DZ.fitfunction") )
AceModel  <- mxModel( "ACE", pars, modelMZ, modelDZ, fitML )
```

```
# Run ADE model
AceFit    <- mxRun(AceModel, intervals=T)
AceSumm   <- summary(AceFit)
AceSumm
```

Each line can be pasted into R, and then evaluated together once the whole model is specified. First, we create R objects to hold start values for the means (*svME*) and the path coefficients (*svPA*) of the model. For the latter, we use the value of the variance divided by the number of variance components (paths) and take the square root.

```
# Set Starting Values
svMe      <- 20      # start value for means
svPa      <- .6      # start value for path coefficients (sqrt(variance/#ofpaths))
```

Given the current example is univariate (in the sense that we analyze one variable, even though we have measured it in two members of twin pairs), the matrices for the paths *a*, *c* and *e* are all `Full` **nv x nv** matrices, with *nv* defined as 1 above, assigned the `free` status `TRUE` and given a `0.6` starting value.

```
# ACE Model
# Matrices declared to store a, d, and e Path Coefficients
pathA     <- mxMatrix( type="Full", nrow=nv, ncol=nv,
                       free=TRUE, values=svPa, label="a11", name="a" )
pathC     <- mxMatrix( type="Full", nrow=nv, ncol=nv,
                       free=TRUE, values=svPa, label="c11", name="c" )
pathE     <- mxMatrix( type="Full", nrow=nv, ncol=nv,
                       free=TRUE, values=svPa, label="e11", name="e" )
```

While the names of these path coefficient matrices are given lower case names, similar to the convention that paths have lower case names, the names for the variance component matrices, obtained from multiplying matrices with their transpose have upper case letters "A", "C" and "E" which are distinct (as R is case-sensitive). Note that the label in the matrices above is distinct from the matrix names with `11` referring to the first row and column of the matrix. We also use an `mxAlgebra` to generate the predicted variance as the sum of the variance components.

```
# Matrices generated to hold A, C, and E computed Variance Components
covA      <- mxAlgebra( expression=a %*% t(a), name="A" )
covC      <- mxAlgebra( expression=c %*% t(c), name="C" )
covE      <- mxAlgebra( expression=e %*% t(e), name="E" )

# Algebra to compute total variances
covP      <- mxAlgebra( expression=A+C+E, name="V" )
```

As the focus is on individual differences, the model for the means is typically simple. We can estimate each of the means, in each of the two groups (MZ & DZ) as free parameters. Alternatively, we can establish whether the means can be equated across order and zygosity by fitting submodels to the saturated model. In this case, we opted to use one 'grand' mean, obtained by assigning the same label to the elements of the matrix `expMean` which is a `Full` **1 x ntv** matrix, where *ntv* is the number of total variables, with free element, labeled `mean` and given a start value of `20`. Note that the R object is called *meanG*, which becomes an argument of the two respective models. The `expMean` matrix name defined in the model is then used in both the MZ and DZ model expectations so that all four elements representing means are equated.

```
# Algebra for expected Mean
meanG     <- mxMatrix( type="Full", nrow=1, ncol=ntv,
                       free=TRUE, values=svMe, label="mean", name="expMean" )
```

Previous Mx users will likely be familiar with the look of the expected covariance matrices for MZ and DZ twin pairs. These **2x2** matrices are built by horizontal and vertical concatenation of the appropriate matrix expressions for the variance, the MZ or the DZ covariance. In R, concatenation of matrices is accomplished with the `rbind` and `cbind`

functions. Thus to represent the matrices in expression below in R, we use the following code.

$$covMZ = \begin{bmatrix} a^2 + c^2 + e^2 & a^2 + c^2 \\ a^2 + c^2 & a^2 + c^2 + e^2 \end{bmatrix}$$

$$covDZ = \begin{bmatrix} a^2 + c^2 + e^2 & .5a^2 + c^2 \\ .5a^2 + c^2 & a^2 + c^2 + e^2 \end{bmatrix}$$

```
# Algebra for expected and Variance/Covariance Matrices in MZ & DZ twins
covMZ      <- mxAlgebra( expression=rbind( cbind(V, A+C),
                                           cbind(A+C, V)), name="expCovMZ" )
covDZ      <- mxAlgebra( expression=rbind( cbind(V, 0.5%x%A+ C),
                                           cbind(0.5%x%A+ C, V)), name="expCovDZ" )
```

Next, the observed data are put in a `mxData` object which also includes a `type` argument, such that OpenMx can apply the appropriate fit function. The actual model expectations are combined in the `mxExpectationNormal` statements which reference the respective predicted covariance matrix, predicted means and list of selected variables to map them onto the data. The maximum likelihood fit function `mxFitFunction()` is used to obtain ML estimates of the parameters of the model.

```
# Data objects for Multiple Groups
dataMZ     <- mxData( observed=mzData, type="raw" )
dataDZ     <- mxData( observed=dzData, type="raw" )

# Objective objects for Multiple Groups
expMZ      <- mxExpectationNormal( covariance="expCovMZ", means="expMean",
                                   dimnames=selVars )
expDZ      <- mxExpectationNormal( covariance="expCovDZ", means="expMean",
                                   dimnames=selVars )
funML      <- mxFitFunctionML()
```

As the expected covariance matrices are different for the two groups of twins, we specify two `mxModel` commands which are given a distinct name and arguments for the predicted means and covariances, the data and the objective function to be used to optimize the model. The objects that are common to both models are combined in a list *pars* which is then included in both the MZ and DZ models and the overall model, which contains the two other models as arguments, as well as the `mxFitFunctionMultigroup` to evaluate both models simultaneously. We refer to the correct fit function by adding the name of the model to the two-level argument, i.e. `MZ.fitfunction`.

```
# Combine Groups
pars       <- list( pathA, pathC, pathE, covA, covC, covE, covP )
modelMZ    <- mxModel( pars, meanG, covMZ, dataMZ, expMZ, funML, name="MZ" )
modelDZ    <- mxModel( pars, meanG, covDZ, dataDZ, expDZ, funML, name="DZ" )
fitML      <- mxFitFunctionMultigroup(c("MZ.fitfunction","DZ.fitfunction") )
AceModel   <- mxModel( "ACE", pars, modelMZ, modelDZ, fitML )
```

Model Fitting

We need to invoke the `mxRun` command to start the model evaluation and optimization. Detailed output will be available in the resulting object, which can be obtained by a `summary` statement.

```
# Run ADE model
AceFit     <- mxRun(AceModel, intervals=T)
AceSumm    <- summary(AceFit)
AceSumm
```

Often, however, one is interested in specific parts of the output. In the case of twin modeling, we typically will inspect the expected covariance matrices and mean vectors, the parameter estimates, and possibly some derived quantities, such as the standardized variance components, obtained by dividing each of the components by the total variance. Note in the code below that the `mxEval` command allows easy extraction of the values in the various matrices/algebras which form the first argument, with the model name as second argument. Once these values have been put in new objects, we can use and regular R expression to derive further quantities or organize them in a convenient format for including in tables. Note that helper functions could (and will likely) easily be written for standard models to produce 'standard' output.

```
# Generate ACE Model Output
estMean   <- mxEval(expMean, AceFit$MZ)        # expected mean
estCovMZ  <- mxEval(expCovMZ, AceFit$MZ)       # expected covariance matrix for MZ's
estCovDZ  <- mxEval(expCovDZ, AceFit$DZ)       # expected covariance matrix for DZ's
estVA     <- mxEval(a*a, AceFit)               # additive genetic variance, a^2
estVC     <- mxEval(c*c, AceFit)               # dominance variance, d^2
estVE     <- mxEval(e*e, AceFit)               # unique environmental variance, e^2
estVP     <- (estVA+estVC+estVE)               # total variance
estPropVA <- estVA/estVP                       # standardized additive genetic variance
estPropVC <- estVC/estVP                       # standardized dominance variance
estPropVE <- estVE/estVP                       # standardized unique environmental variance
estACE    <- rbind(cbind(estVA,estVC,estVE),   # table of estimates
                   cbind(estPropVA,estPropVC,estPropVE))
LL_ACE    <- mxEval(objective, AceFit)         # likelihood of ADE model
```

3.5.2 Alternative Models: an AE Model

To evaluate the significance of each of the model parameters, nested submodels are fit in which these parameters are fixed to zero. If the likelihood ratio test between the two models is significant, the parameter that is dropped from the model significantly contributes to the phenotype in question. Here we show how we can fit the AE model as a submodel with a change in one `mxMatrix` command. First, we call up the previous 'full' model as the first argument of a new model *AeModel* and give it a new name `AE`. Next we re-specify the matrix **c** to be fixed to zero by changing the attributes associated with the specific parameter `c11` to fixed at zero using a `omxSetParameters` command. We can run this model in the same way as before and generate similar summaries of the results.

```
# Run AE model
AeModel   <- mxModel( AceFit, name="AE" )
AeModel   <- omxSetParameters( AeModel, labels="c11", free=FALSE, values=0 )
AeFit     <- mxRun(AeModel)

# Generate AE Model Output
estVA     <- mxEval(a*a, AeFit)                # additive genetic variance, a^2
estVE     <- mxEval(e*e, AeFit)                # unique environmental variance, e^2
estVP     <- (estVA+estVE)                     # total variance
estPropVA <- estVA/estVP                       # standardized additive genetic variance
estPropVE <- estVE/estVP                       # standardized unique environmental variance
estAE     <- rbind(cbind(estVA,estVE),         # table of estimates
                   cbind(estPropVA,estPropVE))
LL_AE     <- mxEval(objective, AeFit)          # likelihood of AE model
```

We use a likelihood ratio test (or take the difference between -2 times the log-likelihoods of the two models) to determine the best fitting model, and print relevant output.

```
LRT_ACE_AE <- LL_AE - LL_ACE

#Print relevant output
estACE
```

```
estAE
LRT_ACE_AE
```

These models may also be specified using paths instead of matrices, which allow for easier submodel specification. See *Genetic Epidemiology, Path Specification* for path specification of these models.

3.6 Definition Variables, Matrix Specification

This example will demonstrate the use of OpenMx definition variables with the implementation of a simple two group dataset. What are definition variables? Essentially, definition variables can be thought of as observed variables which are used to change the statistical model on an individual case basis. In essence, it is as though one or more variables in the raw data vectors are used to specify the statistical model for that individual. Many different types of statistical models can be specified in this fashion; some can be readily specified in standard fashion, and some that cannot. To illustrate, we implement a two-group model. The groups differ in their means but not in their variances and covariances. This situation could easily be modeled in a regular multiple group fashion - it is only implemented using definition variables to illustrate their use. The results are verified using summary statistics and an Mx 1.0 script for comparison is also available.

3.6.1 Mean Differences

The example shows the use of definition variables to test for mean differences. It is available in the following file:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/DefinitionMeans_MatrixRaw.R

A parallel version of this example, using path specification of models rather than matrices, can be found here:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/DefinitionMeans_PathRaw.R

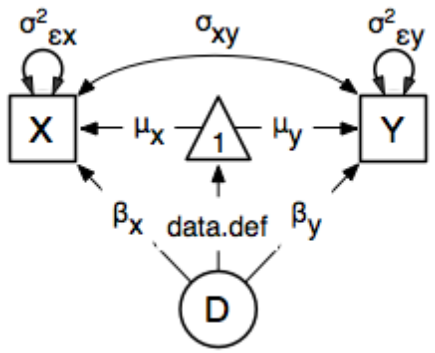
Statistical Model

Algebraically, we are going to fit the following model to the observed x and y variables:

$$x_i = \mu_x + \beta_x * def + \epsilon_{xi}$$
$$y_i = \mu_y + \beta_y * def + \epsilon_{yi}$$

where def is the definition variable and the residual sources of variance, ϵ_{xi} and ϵ_{yi} covary to the extent σ_{xy}. So, the task is to estimate: the two means μ_x and μ_y; the deviations from these means due to belonging to the group identified by having def set to 1 (as opposed to zero), β_x and β_y; and the parameters of the variance covariance matrix: cov(ϵ_x, ϵ_y) = ρ.

Our task is to implement the model shown in the figure below:



Data Simulation

Our first step to running this model is to simulate the data to be analyzed. Each individual is measured on two observed variables, *x* and *y*, and a third variable *def* which denotes their group membership with a 1 or a 0. These values for group membership are not accidental, and must be adhered to in order to obtain readily interpretable results. Other values such as 1 and 2 would yield the same model fit, but would make the interpretation more difficult.

```
library(MASS)   # to get hold of mvrnorm function
set.seed(200)
N               <- 500
Sigma           <- matrix(c(1,.5,.5,1),2,2)
group1          <- mvrnorm(N, c(1,2), Sigma) # Use mvrnorm from MASS package
group2          <- mvrnorm(N, c(0,0), Sigma)
```

We make use of the superb R function `mvrnorm` in order to simulate N=500 records of data for each group. These observations correlate .5 and have a variance of 1, per the matrix *Sigma*. The means of *x* and *y* in group 1 are 1.0 and 2.0, respectively; those in group 2 are both zero. The output of the `mvrnorm` function calls are matrices with 500 rows and 3 columns, which are stored in group 1 and group 2. Now we create the definition variable

```
# Put the two groups together, create a definition variable,
# and make a list of which variables are to be analyzed (selVars)
xy              <- rbind(group1,group2)       # Bind groups together by rows
dimnames(xy)[2]<- list(c("x","y"))            # Add names
def             <- rep(c(1,0),each=N);        # Add def var [2n] for group status
selVars         <- c("x","y")                 # Make selection variables object
```

The objects *xy* and *def* might be combined in a data frame. However, in this case we won't bother to do it externally, and simply paste them together in the `mxData` function call.

Model Specification

The following code contains all of the components of our model. Before running a model, the OpenMx library must be loaded into R using either the `require()` or `library()` function. This code uses the `mxModel` function to create an `mxModel` object, which we'll then run. Note that all the objects required for estimation (data, matrices, and an objective function) are declared within the `mxModel` function. This type of code structure is recommended for OpenMx scripts generally.

```
dataRaw      <- mxData( observed=data.frame(xy,def), type="raw" )
# covariance matrix
Sigma        <- mxMatrix( type="Symm", nrow=2, ncol=2,
                          free=TRUE, values=c(1, 0, 1), name="Sigma" )
# means
Mean         <- mxMatrix( type="Full", nrow=1, ncol=2,
                          free=TRUE, name="Mean" )
# regression coefficient
beta         <- mxMatrix( type="Full", nrow=1, ncol=2,
                          free=TRUE, values=c(0,0), name="beta" )
# definition variable
dataDef      <- mxMatrix( type="Full", nrow=1, ncol=2,
                          free=FALSE, labels=c("data.def"), name="def" )
Mu           <- mxAlgebra( expression=Mean + beta*def, name="Mu" )
exp          <- mxExpectationNormal( covariance="Sigma", means="Mu", dimnames=selVars )
funML        <- mxFitFunctionML()

defMeansModel <- mxModel("Definition  Means Matrix Specification",
                         dataRaw, Sigma, Mean, beta, dataDef, Mu, exp, funML)
```

The first argument in an mxModel function has a special purpose. If an object or variable containing an MxModel object is placed here, then mxModel adds to or removes pieces from that model. If a character string (as indicated by double quotes) is placed first, then that becomes the name of the model. Models may also be named by including a name argument. This model is named "Definition Means Matrix Specification".

Next, we declare where the data are, and their type, by creating an MxData object with the mxData function. This piece of code creates an MxData object. It first references the object where our data are, then uses the type argument to specify that this is raw data. Because the data are raw and the fit function is mxFitFunctionML, full information maximum likelihood is used in this mxModel. Analyses using definition variables have to use raw data, so that the model can be specified on an individual data vector level.

```
dataRaw      <- mxData( observed=data.frame(xy,def), type="raw" )
```

Model specification is carried out using mxMatrix functions to create matrices for the model. In the present case, we need four matrices. First is the predicted covariance matrix, Sigma. Next, we use three matrices to specify the model for the means. First is Mean which corresponds to estimates of the means for individuals with definition variables with values of zero. Individuals with definition variable values of 1 will have the value in Mean plus the value in the matrix beta. So both matrices are of size **1x2** and both contain two free parameters. There is a separate deviation for each of the variables, which will be estimated in the elements 1,1 and 1,2 of the beta matrix. Last, but by no means least, is the matrix def which contains the definition variable. The variable *def* in the mxData data frame is referred to in the matrix label as data.def. In the present case, the definition variable contains a 1 for group 1, and a zero otherwise.

The trick - commonly used in regression models - is to multiply the beta matrix by the def matrix. This multiplication is effected using an mxAlgebra function call:

```
beta         <- mxMatrix( type="Full", nrow=1, ncol=2,
                          free=TRUE, values=c(0,0), name="beta" )
dataDef      <- mxMatrix( type="Full", nrow=1, ncol=2,
                          free=FALSE, labels=c("data.def"), name="def" )
Mu           <- mxAlgebra( expression=Mean + beta*def, name="Mu" )
```

The result of this algebra is named Mu, and this handle is referred to in the mxExpectationNormal function call.

The last argument in this mxModel call is itself a function. It declares that the fit function to be optimized is maximum likelihood (ML), which is tagged mxFitFunctionML. Full information maximum likelihood (FIML) is used whenever the data allow, and does not need to be requested specifically. The third argument in this mxModel is another function. It declares the expectation function to be a normal distribution, mxExpectationNormal. This means the model is of a normal distribution with a particular mean and covariance. Hence, there are in turn two arguments to this function: the covariance matrix Sigma and the mean vector Mu. These matrices will be defined later in the mxModel function call.

```
mxFitFunctionML()
mxExpectationNormal( covariance="Sigma", means="Mu", dimnames=selVars )
```

We can then run the model and examine the output with a few simple commands.

Model Fitting

```
# Run the model
defMeansFit <- mxRun(defMeansModel)
defMeansFit$matrices
defMeansFit$algebras
```

It is possible to compare the estimates from this model to some summary statistics computed from the data:

```
# Compare OpenMx estimates to summary statistics computed from raw data.
# Note that to calculate the common variance,
# group 1 has 1 and 2 subtracted from every Xi and Yi in the sample data,
# so as to estimate variance of combined sample without the mean correction.

# First compute some summary statistics from data
ObsCovs        <- cov(rbind(group1 - rep(c(1,2),each=N), group2))
ObsMeansGroup1 <- c(mean(group1[,1]), mean(group1[,2]))
ObsMeansGroup2 <- c(mean(group2[,1]), mean(group2[,2]))

# Second extract parameter estimates and matrix algebra results from model
Sigma          <- mxEval(Sigma, defMeansFit)
Mu             <- mxEval(Mu, defMeansFit)
Mean           <- mxEval(Mean, defMeansFit)
beta           <- mxEval(beta, defMeansFit)

# Third, check to see if things are more or less equal
omxCheckCloseEnough(ObsCovs,Sigma,.01)
omxCheckCloseEnough(ObsMeansGroup1,as.vector(Mean+beta),.001)
omxCheckCloseEnough(ObsMeansGroup2,as.vector(Mean),.001)
```

These models may also be specified using paths instead of matrices. See *Definition Variables, Path Specification* for path specification of these models.

3.7 Ordinal and Joint Ordinal-Continuous Model Specification

This chapter deals with the specification of models that are either fit exclusively to ordinal variables or to a mix of ordinal and continuous variables. It extends the continuous data common factor model found in previous chapters to ordinal data.

The examples for this chapter can be found in the following files:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/OneFactorOrdinal_MatrixRawRAM.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/OneFactorJoint_MatrixRawRAM.R

The continuous version of this model for raw data can be found the previous demos here:

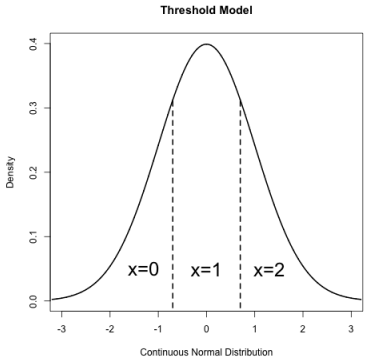
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/OneFactorModel_MatrixRaw.R

We will also discuss an example with simulated ordinal data using a regular matrix specification and an alternative one. We will discuss a common factor model with several indicators. This example can be found in the following files:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/OneFactorOrdinal_MatrixRaw.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/OneFactorOrdinal01_MatrixRaw.R

3.7.1 Ordinal Data Modeling

OpenMx models ordinal data under a threshold model. A continuous normal distribution is assumed to underly every ordinal variable. These latent continuous distributions are only observed as being above or below a threshold, where there is one fewer threshold than observed categories in the data. For example, consider a variable with three ordered categories indicated by the values zero, one and two. Under this approach, this variable is assumed to follow a normal distribution that is partitioned or cut by two thresholds: individuals with underlying scores below the first threshold have an observed value of zero, individuals with latent scores between the thresholds are observed with values of one, and individuals with underlying scores give observed values of two.



Each threshold may be freely estimated or assigned as a fixed parameter, depending on the desired model. In addition to the thresholds, ordinal variables still have a mean and variance that describes the parameters of the underlying continuous distribution. However, this underlying distribution must be scaled by fixing at least two parameters to identify the model. One method of identification fixes the mean and variance to specific values, most commonly to a standard normal distribution with a mean of zero and a variance of one. A variation on this method fixes the residual variance of the categorical variable to one, which is often easier to specify. Alternatively, categorical variables may be identified by fixing two thresholds to non-equivalent constant values. These methods will differ in the scale assigned to the ordinal variables (and thus, the scale of the parameters estimated from them), but all identify the same model and should provide equally valid results.

OpenMx allows for the inclusion of continuous and ordinal variables in the same model, as well as models with only continuous or only ordinal variables. Any number of continuous variables may be included in an OpenMx model; however, maximum likelihood estimation for ordinal data must be limited to twenty ordinal variables regardless of the number of continuous variables. Further technical details on ordinal and joint continuous-ordinal optimization are contained at the end of this chapter.

Data Specification

To use ordinal variables in OpenMx, users must identify ordinal variables by specifying those variables as ordered factors in the included data. Ordinal models can only be fit to raw data; if data is described as a covariance or other moment matrix, then the categorical nature of the data was already modeled to generate that moment matrix. Ordinal variables must be defined as specific columns in an R data frame.

Factors are a type of variable included in an R data frame. Unlike numeric or continuous variables, which must include only numeric and missing values, observed values for factors are treated as character strings. All factors contain a `levels` argument, which lists the possible values for a factor. Ordered factors contain information about the ordering of possible levels. Both R and OpenMx have tools for manipulating factors in data frames. The R functions `factor()` and `as.factor()` (and companions `ordered()` and `as.ordered()`) can be used to specify ordered factors. OpenMx includes a helper function `mxFactor()` which more directly prepares ordinal variables as ordered factors in preparation for inclusion in OpenMx models. The code below demonstrates the `mxFactor()` function, replacing the variable *z1* that was initially read as a continuous variable and treating it as an ordinal variable with two levels. This process is repeated for *z2* (two levels) and *z3* (three levels).

```
data(myFADataRaw)

oneFactorOrd <- myFADataRaw[,c("z1", "z2", "z3")]

oneFactorOrd$z1 <- mxFactor(oneFactorOrd$z1, levels=c(0, 1))
oneFactorOrd$z2 <- mxFactor(oneFactorOrd$z2, levels=c(0, 1))
oneFactorOrd$z3 <- mxFactor(oneFactorOrd$z3, levels=c(0, 1, 2))
```

Threshold Specification

Just as covariances and means are included in models by specifying matrices and algebras, thresholds may be included in models as threshold matrices. These matrices can be of user-specified type, though most will be of type `Full`. The columns of this matrix should correspond to the ordinal variables in your dataset, with the column names of this matrix corresponding to variables in your data. This assignment can be done either with the `dimnames` argument to `mxMatrix`, or by using the `threshnames` argument in your expectation function the same way `dimnames` arguments are used. The rows of your threshold matrix should correspond to the ordered thresholds for each variable, such that the first row is the lowest threshold for each variable, the second row is the next threshold (provided one or more of your variables have two thresholds), and so on for the maximum number of thresholds you have in your data. Rows of the threshold matrix beyond the number of thresholds in a particular variable should be fixed parameters with starting values of `NA`.

As an example, the data prep example above includes two binary variables (*z1* and *z2*) and one variable with three categories (*z3*). This means that the threshold matrix for models fit to this data should contain three columns (for *z1*, *z2* and *z3*) and two rows, as the variable *z3* requires two thresholds. The code below specifies a 2 x 3 `Full` matrix with free parameters for one threshold for *z1*, one threshold for *z2* and two thresholds for *z3*.

```
thresh       <- mxMatrix( type="Full", nrow=2, ncol=3,
                          free=c(TRUE,TRUE,TRUE,FALSE,FALSE,TRUE),
                          values=c(-1,0,-.5,NA,NA,1.2), byrow=TRUE, name="thresh" )
```

There are a few common errors regarding the use of thresholds in OpenMx. First, threshold values within each row must be strictly increasing, such that the value in any element of the threshold matrix must be greater than all values above it in that column. In the above example, the second threshold for *z3* is set at 1.2, above the value of -.5 for the first threshold. OpenMx will return an error when your thresholds are not strictly increasing. There are no restrictions on values across columns or variables: the second threshold for *z3* could be below all thresholds for *z1* and *z2* provided it exceeded the value for the first *z3* threshold. Second, the dimnames of the threshold matrix must match ordinal factors in the data. Additionally, free parameters should only be included for thresholds present in your data: including a second freely estimated threshold for *z1* or *z2* in this example would not directly impede model estimation, but would remain at its starting value and count as a free parameter for the purposes of calculating fit statistics.

It is also important to remember that specifying a threshold matrix is not sufficient to get an ordinal data model to run. In addition, the scale of each ordinal variable must be identified just like the scale of a latent variable. The most common method for this involves constraining a ordinal item's mean to zero and either its total or residual variance to a constant value (i.e., one). For variables with two or more thresholds, ordinal variables may also be identified by constraining two thresholds to fixed values. Models that don't identify the scale of their ordinal variables should not converge.

While thresholds can't be expressed as paths between variables like other parts of the model, OpenMx supports a path-like interface called `mxThreshold` as of version 2.0. This function is described in more detail in the ordinal data version of this chapter and the `mxThreshold` help file.

Users of original or ''classic" Mx may recall specifying thresholds not in absolute terms, but as deviations. This method estimated the difference between each threshold for a variable and the previous one, which ensured that thresholds were in the correct order (i.e., that the second threshold for a variable was not lower than the first). Users should still employ this method using `mxAlgebra` for more complex models, as during optimization, thresholds may otherwise get out of proper order, causing optimization to stop.

Including Thresholds in Models

Finally, the threshold matrix must be identified as such in the expectation function in the same way that other matrices are identified as means or covariance matrices. Both the `mxExpectationNormal` and `mxExpectationRAM` contain a `thresholds` argument, which takes the name of the matrix or algebra to be used as the threshold matrix for a given analysis. Although specifying `type='RAM'` generates a RAM expectation function, this expectation function must be replaced by one with a specified thresholds matrix.

You must specify `dimnames` (dimension names) for your thresholds matrix that correspond to the ordered factors in the data you wish to analyze. This may be done in either of two ways, both of which correspond to specifying dimnames for other OpenMx matrices. One method is to use the `threshnames` argument in the `mxExpectationNormal` or `mxExpectationRAM` functions, which specifies which variables are in a threshold matrix in the same way the `dimnames` argument specifies which variables are in the rest of the model. Another method is to specify dimnames for each matrix using the `dimnames` argument in the `mxMatrix` function. Either method may be used, but it is important to use the same method for all matrices in a given model (either using expectation function arguments `dimnames` and `threshnames` or supplying `dimnames` for all `mxMatrix` objects manually). Expectation function arguments `dimnames` and `threshnames` supersede the matrix `dimname` arguments, and `threshnames` will take the value of the `dimnames` if both `dimnames` and `thresholds` are specified but `threshnames` is omitted.

The code below specifies an `mxExpectationRAM` to include a thresholds matrix named `"thresh"`. When models are built using `type='RAM'`, the `dimnames` argument may be omitted, as the requisite dimnames for the `A`, `S`, `F` and `M` matrices are generated from the `manifestVars` and `latentVars` lists. However, the dimnames for the threshold matrix should be included using the `dimnames` argument in `mxMatrix`.

```
mxExpectationRAM(A="A", S="S", F="F", M="M", thresholds="thresh")
```

3.7.2 Common Factor Model

All of the raw data examples through the documentation may be converted to ordinal examples by the inclusion of ordinal data, the specification of a threshold matrix and inclusion of that threshold matrix in the objective function.

Ordinal Data

The following example is a version of the continuous data common factor model referenced at the beginning of this chapter. Aside from replacing the continuous variables `x1-x6` with the ordinal variables `z1-z3`, the code below simply incorporates the steps referenced above into the existing example. Data preparation occurs first, with the added `mxFactor` statements to identify ordinal variables and their ordered levels.

```
require(OpenMx)

data(myFADataRaw)

oneFactorOrd <- myFADataRaw[,c("z1", "z2", "z3")]

oneFactorOrd$z1 <- mxFactor(oneFactorOrd$z1, levels=c(0, 1))
oneFactorOrd$z2 <- mxFactor(oneFactorOrd$z2, levels=c(0, 1))
oneFactorOrd$z3 <- mxFactor(oneFactorOrd$z3, levels=c(0, 1, 2))
```

Model specification can be achieved by appending the above threshold matrix and expectation function to either the path or matrix common factor examples. The path example below has been altered by changing the variable names from `x1-x6` to `z1-z3`, adding the threshold matrix and expectation function, and identifying the ordinal variables by constraining their means to be zero and their residual variances to be one.

```
dataRaw      <- mxData(oneFactorOrd, type="raw")
# asymmetric paths
matrA        <- mxMatrix( type="Full", nrow=4, ncol=4,
                          free=c(F,F,F,T,
                                 F,F,F,T,
                                 F,F,F,T,
                                 F,F,F,F),
                          values=c(0,0,0,1,
                                   0,0,0,1,
```

```
                                0,0,0,1,
                                0,0,0,0),
                          labels=c(NA,NA,NA,"l1",
                                   NA,NA,NA,"l2",
                                   NA,NA,NA,"l3",
                                   NA,NA,NA,NA),
                          byrow=TRUE, name="A" )
# symmetric paths
matrS        <- mxMatrix( type="Symm", nrow=4, ncol=4,
                          free=FALSE,
                          values=diag(4),
                          labels=c("e1", NA, NA,  NA,
                                    NA,"e2", NA,  NA,
                                    NA,  NA,"e3", NA,
                                    NA,  NA, NA, "varF1"),
                          byrow=TRUE, name="S" )
# filter matrix
matrF        <- mxMatrix( type="Full", nrow=3, ncol=4,
                          free=FALSE, values=c(1,0,0,0,  0,1,0,0,  0,0,1,0),
                          byrow=TRUE, name="F" )
# means
matrM        <- mxMatrix( type="Full", nrow=1, ncol=4,
                          free=FALSE, values=0,
                          labels=c("meanz1","meanz2","meanz3",NA), name="M" )
thresh       <- mxMatrix( type="Full", nrow=2, ncol=3,
                          free=c(TRUE,TRUE,TRUE,FALSE,FALSE,TRUE),
                          values=c(-1,0,-.5,NA,NA,1.2), byrow=TRUE, name="thresh" )
exp          <- mxExpectationRAM("A","S","F","M", dimnames=c("z1","z2","z3","F1"),
                          thresholds="thresh", threshnames=c("z1","z2","z3"))
funML        <- mxFitFunctionML()

oneFactorOrdinalModel <- mxModel("Common Factor Model Matrix Specification",
                                 dataRaw, matrA, matrS, matrF, matrM, thresh, exp, funML)
```

This model may then be optimized using the `mxRun` command.

```
oneFactorOrdinalFit <- mxRun(oneFactorOrdinalModel)
```

Joint Ordinal-Continuous Data

Models with both continuous and ordinal variables may be specified just like any other ordinal data model. Threshold matrices in these models should contain columns only for the ordinal variables, and should contain column names to designate which variables are to be treated as ordinal. In the example below, the one factor model above is estimated with three continuous variables (`x1-x3`) and three ordinal variables (`z1-z3`).

```
require(OpenMx)

oneFactorJoint <- myFADataRaw[,c("x1", "x2", "x3", "z1", "z2", "z3")]

oneFactorJoint$z1 <- mxFactor(oneFactorOrd$z1, levels=c(0, 1))
oneFactorJoint$z2 <- mxFactor(oneFactorOrd$z2, levels=c(0, 1))
oneFactorJoint$z3 <- mxFactor(oneFactorOrd$z3, levels=c(0, 1, 2))

dataRaw      <- mxData(observed=oneFactorJoint, type="raw")
# asymmetric paths
matrA        <- mxMatrix( type="Full", nrow=7, ncol=7,
                          free=c(rep(c(F,F,F,F,F,F,T),6),rep(F,7)),
```

```
                          values=c(rep(c(0,0,0,0,0,0,1),6),rep(F,7)),
                          labels=rbind(cbind(matrix(NA,6,6),matrix(paste("l",1:6,sep=""),6,1)),
                           matrix(NA,1,7)),
                          byrow=TRUE, name="A" )
# symmetric paths
labelsS       <- matrix(NA,7,7); diag(labelsS) <- c(paste("e",1:6,sep=""),"varF1")
matrS         <- mxMatrix( type="Symm", nrow=7, ncol=7,
                          free= rbind(cbind(matrix(as.logical(diag(3)),3,3),matrix(F,3,4)),
                           matrix(F,4,7)),
                          values=diag(7), labels=labelsS, byrow=TRUE, name="S" )
# filter matrix
matrF         <- mxMatrix( type="Full", nrow=6, ncol=7,
                          free=FALSE, values=cbind(diag(6),matrix(0,6,1)),
                          byrow=TRUE, name="F" )
# means
matrM         <- mxMatrix( type="Full", nrow=1, ncol=7,
                          free=c(T,T,T,F,F,F,F), values=c(1,1,1,0,0,0,0),
                          labels=c("meanx1","meanx2","meanx3","meanz1","meanz2","meanz3",NA),
                          name="M" )
thresh        <- mxMatrix( type="Full", nrow=2, ncol=3,
                          free=c(TRUE,TRUE,TRUE,FALSE,FALSE,TRUE),
                          values=c(-1,0,-.5,NA,NA,1.2), byrow=TRUE, name="thresh" )
exp           <- mxExpectationRAM("A","S","F","M",
                                  dimnames=c("x1","x2","x3","z1","z2","z3","F1"),
                                  thresholds="thresh", threshnames=c("z1","z2","z3"))
funML         <- mxFitFunctionML()

oneFactorJointModel <- mxModel("Common Factor Model Matrix Specification",
                               dataRaw, matrA, matrS, matrF, matrM, thresh, exp, funML)
```

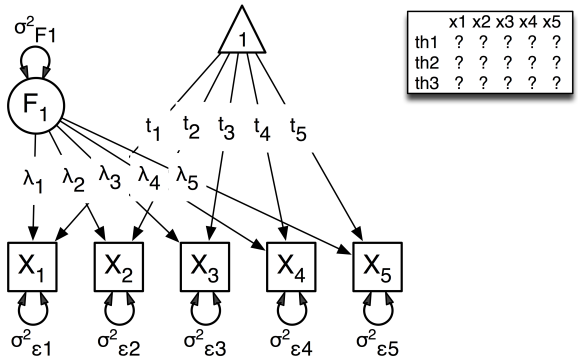
This model may then be optimized using the `mxRun` command.

```
oneFactorJointFit <- mxRun(oneFactorJointModel)
```

Simulated Ordinal Data

The common factor model is a method for modeling the relationships between observed variables believed to measure or indicate the same latent variable. While there are a number of exploratory approaches to extracting latent factor(s), this example uses structural modeling to fit a confirmatory factor model. The model for any person and path diagram of the common factor model for a set of variables x_1 - x_5 are given below.

$$x_{ij} = \mu_j + \lambda_j * \eta_i + \epsilon_{ij}$$



The path diagram above displays 16 parameters (represented in the arrows: 5 manifest variances, five manifest means, five factor loadings and one factor variance). However, given we are dealing with ordinal data in this example, we are estimating thresholds rather than means, with nThresholds being one less the number of categories in the variables, here 3. Furthermore, we must constrain either the factor variance or one factor loading to a constant to identify the model and scale the latent variable. In this instance, we chose to constrain the variance of the factor. We also need to constrain the total variances of the manifest variables, as ordinal variables do not have a scale of measurement. As such, this model contains 20 free parameters and is not fully saturated.

Data

Our first step to running this model is to include the data to be analyzed. The data for this example were simulated in R. Given the focus of this documentation is on OpenMx, we will not discuss the details of the simulation here, but we do provide the code so that the user can simulate data in a similar way.

```
# Step 1: set up simulation parameters
# Note: nVariables>=5, nThresholds>=1, nSubjects>=nVariables x nThresholds
# (maybe more) and model should be identified
nVariables   <-5
nFactors     <-1
nThresholds  <-3
nSubjects    <-500
isIdentified <- function(nVariables,nFactors)
    as.logical(1+sign((nVariables*(nVariables-1)/2)
    - nVariables*nFactors + nFactors*(nFactors-1)/2))
# if this function returns FALSE then model is not identified, otherwise it is.
isIdentified(nVariables,nFactors)

loadings     <- matrix(.7,nrow=nVariables,ncol=nFactors)
residuals    <- 1-(loadings * loadings)
sigma        <- loadings %*% t(loadings) + vec2diag(residuals)
mu           <- matrix(0,nrow=nVariables,ncol=1)

# Step 2: simulate multivariate normal data
set.seed(1234)
continuousData <- mvrnorm(n=nSubjects,mu,sigma)

# Step 3: chop continuous variables into ordinal data
# with nThresholds+1 approximately equal categories, based on 1st variable
quants       <- quantile(continuousData[,1],  probs = c((1:nThresholds)/(nThresholds+1)))
ordinalData  <- matrix(0,nrow=nSubjects,ncol=nVariables)
for(i in 1:nVariables)
{ ordinalData[,i] <- cut(as.vector(continuousData[,i]),c(-Inf,quants,Inf)) }

# Step 4: make the ordinal variables into R factors
ordinalData  <- mxFactor(as.data.frame(ordinalData),levels=c(1:(nThresholds+1)))

# Step 5: name the variables
bananaNames  <- paste("banana",1:nVariables,sep="")
names(ordinalData) <- bananaNames
```

Model Specification

The following code contains all of the components of our model. Before running a model, the OpenMx library must be loaded into R using either the `require()` or `library()` function. All objects required for estimation (data, matrices, an expectation function, and a fit function) are included in their functions. This code uses the `mxModel` function to create an `MxModel` object, which we will then run. We pre-specify a number of 'variables', namely

the number of variables analyzed `nVariables`, in this case 5, the number of factors `nFactors`, here one, and the number of thresholds `nThresholds`, here 3 or one less than the number of categories in the simulated ordinal variable.

```
facLoads      <- mxMatrix( type="Full", nrow=nVariables, ncol=nFactors,
                           free=TRUE, values=0.2, lbound=-.99, ubound=.99, name="facLoadings" )
vecOnes       <- mxMatrix( type="Unit", nrow=nVariables, ncol=1, name="vectorofOnes" )
resVars       <- mxAlgebra( expression=vectorofOnes -
                            (diag2vec(facLoadings %*% t(facLoadings))), name="resVariances" )
expCovs       <- mxAlgebra( expression=facLoadings %*% t(facLoadings)
                            + vec2diag(resVariances), name="expCovariances" )
expMeans      <- mxMatrix( type="Zero", nrow=1, ncol=nVariables, name="expMeans" )
threDevs      <- mxMatrix( type="Full", nrow=nThresholds, ncol=nVariables,
                           free=TRUE, values=.2,
                           lbound=rep( c(-Inf,rep(.01,(nThresholds-1))) , nVariables),
                           dimnames=list(c(), bananaNames), name="thresholdDeviations" )
unitLower     <- mxMatrix( type="Lower", nrow=nThresholds, ncol=nThresholds,
                           free=FALSE, values=1, name="unitLower" )
expThres      <- mxAlgebra( expression=unitLower %*% thresholdDeviations,
                            name="expThresholds" )
dataRaw       <- mxData( observed=ordinalData, type='raw' )
exp           <- mxExpectationNormal( covariance="expCovariances", means="expMeans",
                                      dimnames=bananaNames, thresholds="expThresholds" )
funML         <- mxFitFunctionML()

oneFactorThresholdModel <- mxModel("oneFactorThresholdModel", dataRaw,
                                   facLoads, vecOnes, resVars, expCovs, expMeans, threDevs,
                                   unitLower, expThres, dataRaw, exp, funML )
```

This `mxModel` function can be split into several parts. First, we give the model a name "Common Factor Threshold-Model Matrix Specification".

The second component of our code creates an `MxData` object. The example above, reproduced here, first references the object where our data is, then uses the `type` argument to specify that this is raw data.

```
dataRaw       <- mxData( observed=ordinalData, type='raw' )
```

The first `mxMatrix` statement declares a `Full` **nVariables x nFactors** matrix of factor loadings to be estimated, called "facLoadings", where the rows represent the dependent variables and the column(s) represent the independent variable(s). The common factor model requires that one parameter (typically either a factor loading or factor variance) be constrained to a constant value. In our model, we will constrain the factor variance to 1 for identification, and let all the factor loadings be freely estimated. Even though we specify just one start value of 0.2, it is recycled for each of the elements in the matrix. Given the factor variance is fixed to one, and the variances of the observed variables are fixed to one (see below), the factor loadings are standarized, and thus must lie between -.99 and .99 as indicated by the `lbound` and `ubound` values.

```
# factor loadings
facLoads      <- mxMatrix( type="Full", nrow=nVariables, ncol=nFactors,
                           free=TRUE, values=0.2, lbound=-.99, ubound=.99, name="facLoadings" )
```

Note that if `nFactors>1`, we could add a `standardized` `mxMatrix` to estimate the correlation between the factors. Such a matrix automatically has 1's on the diagonal, fixing the factor variances to one and thus allowing all the factor loadings to be estimated. In the current example, all the factor loadings are estimated which implies that the factor variance is fixed to 1. Alternatively, we could add a `symmetric` **1x1** matrix to estimates the variance of the factor, when one of the factor loadings is fixed.

As our data are ordinal, we further need to constrain the variances of the observed variables to unity. These variances are made up of the contributions of the latent common factor and the residual variances. The amount of variance explained by the common factor is obtained by squaring the factor loadings. We subtract the squared factor loadings

from 1 to get the amount explained by the residual variance, thereby implicitly fixing the variances of the observed variables to 1. To do this for all variables simultaneously, we use matrix algebra functions. We first specify a vector of One's by declaring a `Unit` **nVariables x 1** matrix called `vectorofOnes`. We need to subtract the squared factor loadings which are on the diagonal of the matrix multiplication of the factor loading matrix `facLoadings` and its transpose. To extract those into squared factor loadings into a vector, we use the `diag2vec` function. This new vector is subtracted from the `vectorofOnes` using an `mxAlgebra` statement to generate the residual variances, and named `resVariances`.

```
vecOnes      <- mxMatrix( type="Unit", nrow=nVariables, ncol=1, name="vectorofOnes" )
# residuals
resVars      <- mxAlgebra( expression=vectorofOnes -
                          (diag2vec(facLoadings %*% t(facLoadings))), name="resVariances" )
```

We then use the reverse function `vec2diag` to put the residual variances on the diagonal and add the contributions through the common factor from the matrix multipication of the factor loadings matrix and its transpose to obtain the formula for the expected covariances, aptly named `expCovariances`.

```
# expected covariances
expCovs      <- mxAlgebra( expression=facLoadings %*% t(facLoadings)
                           + vec2diag(resVariances), name="expCovariances" )
```

When fitting to ordinal rather than continuous data, we estimate thresholds rather than means. The matrix of thresholds is of size **nThresholds x nVariables** where `nThresholds` is one less than the number of categories for the ordinal variable(s). We still specify a matrix of means, however, it is fixed to zero. An alternative approach is to fix the first two thresholds (to zero and one, see below), which allows us to estimate means and variances in a similar way to fitting to continuous data. Let's first specify the model with zero means and free thresholds.

The means are specified as a `Zero` **1 x nVariables** matrix, called `expMeans`. A means matrix always contains a single row, and one column for every manifest variable in the model.

```
# expected means
expMeans     <- mxMatrix( type="Zero", nrow=1, ncol=nVariables, name="expMeans" )
```

The mean of the factor(s) is also fixed to 0, which is implied by not including a matrix for it. Alternatively, we could explicitly add a `Full` **1 x nFactors** matrix with a fixed value of zero for the factor mean(s), named "facMeans".

We estimate the `Full` **nThresholds x nVariables** matrix. To make sure that the thresholds systematically increase from the lowest to the highest, we estimate the first threshold and the increments compared to the previous threshold by constraining the increments to be positive. This is accomplished through some R algebra, concatenating *minus infinity* and (nThreshold-1) times .01 as the lower bound for the remaining estimates. This matrix of `thresholdDeviations` is then pre-multiplied by a `lower` triangular matrix of ones of size **nThresholds x nThresholds** to obtain the expected thresholds in increasing order in the `thresholdMatrix`.

```
threDevs     <- mxMatrix( type="Full", nrow=nThresholds, ncol=nVariables,
                          free=TRUE, values=.2,
                          lbound=rep( c(-Inf,rep(.01,(nThresholds-1))) , nVariables),
                          dimnames=list(c(), bananaNames), name="thresholdDeviations" )
unitLower    <- mxMatrix( type="Lower", nrow=nThresholds, ncol=nThresholds,
                          free=FALSE, values=1, name="unitLower" )
# expected thresholds
expThres     <- mxAlgebra( expression=unitLower %*% thresholdDeviations,
                          name="expThresholds" )
```

The final parts of this model are the expectation function and the fit function. The choice of expectation function determines the required arguments. Here we fit to raw ordinal data, thus we specify the matrices for the expected covariance matrix of the data, as well as the expected means and thresholds previously specified. We use `dimnames` to map the model for means, thresholds and covariances onto the observed variables.

```
exp          <- mxExpectationNormal( covariance="expCovariances", means="expMeans",
                                     dimnames=bananaNames, thresholds="expThresholds" )
funML        <- mxFitFunctionML()
```

The free parameters in the model can then be estimated using full information maximum likelihood (FIML) for covariances, means and thresholds. FIML is specified by using raw data with the `mxFitFunctionML`. To estimate free parameters, the model is run using the `mxRun` function, and the output of the model can be accessed from the `$output` slot of the resulting model. A summary of the output can be reached using `summary()`.

```
oneFactorThresholdFit <- mxRun(oneFactorThresholdModel)

oneFactorThresholdFit$output
summary(oneFactorThresholdFit)
```

Alternative Specification

As indicate above, the model can be re-parameterized such that means and variances of the observed variables are estimated similar to the continuous case, by fixing the first two thresholds. This basically rescales the parameters of the model. Below is the full script:

```
facLoads     <- mxMatrix( type="Full", nrow=nVariables, ncol=nFactors,
                          free=TRUE, values=0.2, lbound=-.99, ubound=2, name="facLoadings" )
resVars      <- mxMatrix( type="Diag", nrow=nVariables, ncol=nVariables,
                          free=TRUE, values=0.9, name="resVariances" )
expCovs      <- mxAlgebra( expression=facLoadings %*% t(facLoadings) + resVariances,
                           name="expCovariances" )
expMeans     <- mxMatrix( type="Full", nrow=1, ncol=nVariables, free=TRUE, name="expMeans" )
threDevs     <- mxMatrix( type="Full", nrow=nThresholds, ncol=nVariables,
                          free=rep( c(F,F,rep(T,(nThresholds-2))), nVariables),
                          values=rep( c(0,1,rep(.2,(nThresholds-2))), nVariables),
                          lbound=rep( c(-Inf,rep(.01,(nThresholds-1))), nVariables),
                          dimnames=list(c(), bananaNames), name="thresholdDeviations" )
unitLower    <- mxMatrix( type="Lower", nrow=nThresholds, ncol=nThresholds,
                          free=FALSE, values=1, name="unitLower" )
expThres     <- mxAlgebra( expression=unitLower %*% thresholdDeviations,
                           name="expThresholds" )

colOnes      <- mxMatrix( type="Unit", nrow=nThresholds, ncol=1, name="columnofOnes" )
matMeans     <- mxAlgebra( expression=expMeans %x% columnofOnes, name="meansMatrix" )
matVars      <- mxAlgebra( expression=sqrt(t(diag2vec(expCovariances))) %x% columnofOnes,
                           name="variancesMatrix" )
matThres     <- mxAlgebra( expression=(expThresholds - meansMatrix) / variancesMatrix,
                           name="thresholdMatrix" )
identity     <- mxMatrix( type="Iden", nrow=nVariables, ncol=nVariables, name="Identity" )
stFacLoads   <- mxAlgebra( expression=solve(sqrt(Identity * expCovariances)) %*% facLoadings,
                           name="standFacLoadings" )
dataRaw      <- mxData( observed=ordinalData, type='raw' )
exp          <- mxExpectationNormal( covariance="expCovariances", means="expMeans",
                                     dimnames=bananaNames, thresholds="expThresholds" )
funML        <- mxFitFunctionML()

oneFactorThreshold01Model <- mxModel("oneFactorThreshold01Model", dataRaw,
                                     facLoads, resVars, expCovs, expMeans, threDevs,
                                     unitLower, expThres,
                                     colOnes, matMeans, matVars, matThres, identity,
                                     stFacLoads, dataRaw, exp, funML )
```

We will only highlight the changes from the previous model specification. By fixing the first and second threshold to 0 and 1 respectively for each variable, we are now able to estimate a mean and a variance for each variable instead. If we are estimating the variances of the observed variables, the factor loadings are no longer standardized, thus we relax the upper boundary on the factor loading matrix `facLoadings` to be 2. The residual variances are now directly estimated as a `Diagonal` matrix of size **nVariables x nVariables**, and given a start value higher than that for the factor loadings. As the residual variances are already on the diagonal of the `resVariances` matrix, we no longer need to add the `vec2diag` function to obtain the `expCovariances` matrix.

```
facLoads     <- mxMatrix( type="Full", nrow=nVariables, ncol=nFactors,
                          free=TRUE, values=0.2, lbound=-.99, ubound=2, name="facLoadings" )
resVars      <- mxMatrix( type="Diag", nrow=nVariables, ncol=nVariables,
                          free=TRUE, values=0.9, name="resVariances" )
expCovs      <- mxAlgebra( expression=facLoadings %*% t(facLoadings) + resVariances,
                           name="expCovariances" )
```

Next, we now estimate the means for the observed variables and thus change the `expMeans` matrix to a `Full` matrix, and set it free. The most complicated change happens to the matrix of `thresholdDeviations`. Its type and dimensions stay the same. However, we now fix the first two thresholds, but allow the remainder of the thresholds (in this case, just one) to be estimated. We use the R `rep` function to make this happen. The `values` statement now has the fixed value of 0 for the first threshold, the fixed value of 1 for the second threshold, and the start value of .2 for the remaining threshold(s). Finally, no change is required for the `lbound` matrix, which is still necessary to keep the estimated increments (third threshold and possible more) positive.

```
expMeans     <- mxMatrix( type="Full", nrow=1, ncol=nVariables, free=TRUE, name="expMeans" )
threDevs     <- mxMatrix( type="Full", nrow=nThresholds, ncol=nVariables,
                          free=rep( c(F,F,rep(T,(nThresholds-2))), nVariables),
                          values=rep( c(0,1,rep(.2,(nThresholds-2))), nVariables),
                          lbound=rep( c(-Inf,rep(.01,(nThresholds-1))), nVariables),
                          dimnames=list(c(), bananaNames), name="thresholdDeviations" )
```

These are all the changes required to fit the alternative specification, which should give the same likelihood and goodness-of-fit statistics as the original one. We have added some matrices and algebra to calculate the 'standardized' thresholds and factor loadings which should be equal to those obtained with the original specification. To standardize the thresholds, the respective mean is subtracted from the thresholds, by expanding the means matrix to the same size as the threshold matrix. The result is divided by the corresponding standard deviation. To standardize the factor loadings, they are pre-multiplied by the inverse of the standard deviations.

```
colOnes      <- mxMatrix( type="Unit", nrow=nThresholds, ncol=1, name="columnofOnes" )
matMeans     <- mxAlgebra( expression=expMeans %x% columnofOnes, name="meansMatrix" )
matVars      <- mxAlgebra( expression=sqrt(t(diag2vec(expCovariances))) %x% columnofOnes,
                           name="variancesMatrix" )
matThres     <- mxAlgebra( expression=(expThresholds - meansMatrix) / variancesMatrix,
                           name="thresholdMatrix" )
identity     <- mxMatrix( type="Iden", nrow=nVariables, ncol=nVariables, name="Identity" )
stFacLoads   <- mxAlgebra( expression=solve(sqrt(Identity * expCovariances))
                           %*% facLoadings, name="standFacLoadings" )
```

3.7.3 Technical Details

Maximum likelihood estimation for ordinal variables by generating expected covariance and mean matrices for the latent continuous variables underlying the set of ordinal variables, then integrating the multivariate normal distribution defined by those covariances and means. The likelihood for each row of the data is defined as the multivariate integral of the expected distribution over the interval defined by the thresholds bordering that row's data. OpenMx uses Alan Genz's SADMVN routine for multivariate normal integration (see http://www.math.wsu.edu/faculty/genz/software/software.html for more information).

When continuous variables are present, OpenMx utilizes a block decomposition to separate the continuous and ordinal covariance matrices for FIML. The likelihood of the continuous variables is calculated normally. The effects of the point estimates of the continuous variables is projected out of the expected covariance matrix of the ordinal data. The likelihood of the ordinal data is defined as the multivariate integral over the distribution defined by the resulting ordinal covariance matrix.

3.8 Growth Mixture Modeling, Matrix Specification

This example will demonstrate how to specify a growth mixture model using matrix specification. This model can only be fit to raw data. The script for this example can be found in the following file:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/GrowthMixtureModel_MatrixRaw.R

A parallel example using path specification can be found here:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/GrowthMixtureModel_PathRaw.R

The latent growth curve used in this example is the same one fit in the latent growth curve example. Path and matrix versions of that example for raw data can be found here:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/LatentGrowthCurveModel_PathRaw.R
- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/LatentGrowthCurveModel_MatrixRaw.R

3.8.1 Mixture Modeling

Mixture modeling is an approach where data are assumed to be governed by some type of mixture distribution. This includes a large class of models, including many varieties of mixture modeling, latent class analysis and related models with binary or categorical latent variables. This example will demonstrate a growth mixture model, where change over time is modeled with a linear growth curve and the distribution of latent intercepts and slopes is governed by a mixture of two distributions. The model can thus be described as a combination of two growth curves, weighted by a class quantity variable, as shown below.

$$x_{ij} = p_1(Intercept_{i1} + \lambda_1 Slope_{i1} + \epsilon) + p_2(Intercept_{i2} + \lambda_2 Slope_{i2} + \epsilon)$$

To scale the class quantity variable as a probability, it must be scaled such that it is strictly positive and the set of all class probabilities sum to a value of one.

$$\sum_{i=1}^{k} p_i = 1$$

Data

The data for this example can be found in the data object *myGrowthMixtureData*. These data contain five time ordered variables named `x1` through `x5`, just like the growth curve demo mentioned previously. It is important to note that raw data is required for mixture modeling, as moment matrices do not contain all of the information required to estimate the model.

```
data(myGrowthMixtureData)
names(myGrowthMixtureData)
```

Model Specification

Specifying a mixture model can be categorized into two general phases. The first phase of model specification pertains to creating the models for each class. The second phase specifies the way those classes are mixed. In OpenMx, this is done using a model tree. Each class is created as a separate `MxModel` object, and those class-specific models are all placed into a larger or parent model. The parent model contains the class quantity parameter(s) and the data.

Creating the class-specific models is done the same way as every other model. We'll begin by specifying the model for the first class using RAM matrices. The code below specifies a five-occasion linear growth curve, virtually identical to the one in the linear growth curve example referenced above. The only real changes made to this model are the names of the free parameters; the means, variances and covariance of the intercept and slope terms are now followed by the number 1 to distinguish them from free parameters in the other class. However, we've made extensive use of R features to generate the required matrices, rather than specifying every element explicitly. This may appear less readable at first, but it is much easier to generalize to matrices of different size. Note that the `vector` argument in `mxFitFunctionML` has been set to "TRUE", which will be discussed in more detail shortly.

```
matrA        <- mxMatrix( type="Full", nrow=7, ncol=7,
                          free=F, values=rbind(cbind(matrix(0,5,5),
                          matrix(c(rep(1,5),0:4),5,2)),matrix(0,2,7)),
                          byrow=TRUE, name="A" )
labelsS      <- matrix(NA,5,5); diag(labelsS) <- "residual"
matrS        <- mxMatrix( type="Symm", nrow=7, ncol=7,
                          free=rbind(cbind(matrix(as.logical(diag(5)),5,5),
                          matrix(F,5,2)),cbind(matrix(F,2,5),matrix(T,2,2))),
                          values=rbind(cbind(matrix((diag(5)),5,5),
                          matrix(0,5,2)),cbind(matrix(0,2,5),matrix(c(1,.4,.4,1),2,2))),
                          labels=rbind(cbind(labelsS, matrix(NA,5,2)),cbind(matrix(NA,2,5),
                          matrix(c("vari1","cov1","cov1","vars1"),2,2))),
                          byrow= TRUE, name="S" )
matrF        <- mxMatrix( type="Full", nrow=5, ncol=7,
                          free=F, values=cbind(diag(5),matrix(0,5,2)),
                          byrow=T, name="F" )
matrM        <- mxMatrix( type="Full", nrow=1, ncol=7,
                          free=c(F,F,F,F,F,T,T),
                          values=c(0,0,0,0,0,0,-1),
                          labels=c(NA,NA,NA,NA,NA,"meani1","means1"), name="M" )
exp          <- mxExpectationRAM("A","S","F","M",
                          dimnames=c(names(myGrowthMixtureData),"intercept","slope"))
funML        <- mxFitFunctionML(vector=TRUE)
class1       <- mxModel("Class1", matrA, matrS, matrF, matrM, exp, funML)
```

We could create the model for our second class by copy and pasting the code above, but that can yield needlessly long scripts. We can also use the `mxModel` function to edit an existing model object, allowing us to change only the parameters that vary across classes. The `mxModel` call below begins with an existing `MxModel` object (`class1`) rather than a model name. The subsequent `mxMatrix` functions replace any existing matrices that have the same name. As we did not give the model a name at the beginning of the `mxModel` function, we must use the `name` argument to identify this model by name.

```
matrS2       <- mxMatrix( type="Symm", nrow=7, ncol=7,
                          free=rbind(cbind(matrix(as.logical(diag(5)),5,5),
                          matrix(F,5,2)),cbind(matrix(F,2,5),matrix(T,2,2))),
                          values=rbind(cbind(matrix((diag(5)),5,5),
                          matrix(0,5,2)),cbind(matrix(0,2,5),matrix(c(1,.5,.5,1),2,2))),
                          labels=rbind(cbind(labelsS, matrix(NA,5,2)),cbind(matrix(NA,2,5),
                          matrix(c("vari2","cov2","cov2","vars2"),2,2))),
                          byrow= TRUE, name="S2" )
matrM2       <- mxMatrix( type="Full", nrow=1, ncol=7,
                          free=c(F,F,F,F,F,T,T),
```

```
                        values=c(0,0,0,0,0,5,1),
                        labels=c(NA,NA,NA,NA,NA,"meani2","means2"), name="M2" )
exp         <- mxExpectationRAM("A","S2","F","M2",
                        dimnames=c(names(myGrowthMixtureData),"intercept","slope"))
class2      <- mxModel( class1, name="Class2", matrS2, matrM2, exp )
```

The `vector=TRUE` argument in the above code merits further discussion. The fit function for each of the class-specific models must return the likelihoods for each individual rather than the default log likelihood for the entire sample. OpenMx fit functions that handle raw data have the option to return a vector of likelihoods for each row rather than a single likelihood value for the dataset. This option can be accessed either as an argument in a function like `mxFitFunctionML`, as was done above, or with the syntax below.

```
class1@fitfunction@vector <- TRUE
class2@fitfunction@vector <- TRUE
```

While the class-specific models can be specified using either path or matrix specification, the class quantity parameters must be specified using a matrix, though it can be specified a number of different ways. The challenge of specifying class probabilities lies in their inherent constraint: class probabilities must be non-negative and sum to unity. The code below demonstrates one method of specifying class quantity parameters and rescaling them as probabilities.

This method for specifying class probabilities consists of two parts. In the first part, the matrix in the object *classQ* contains two elements representing the class quantities for each class. One class is designated as a reference class by fixing their quantity at a value of one (class 2 below). All other classes are assigned free parameters in this matrix, and should be interpreted as quantity of sample in that class per person in the reference class. These parameters should have a lower bound at or near zero. Specifying class quantities rather than class probabilities avoids the degrees of freedom issue inherent to class probability parameters by only estimating k-1 parameters for k classes.

```
classQ      <- mxMatrix( type="Full", nrow=2, ncol=1,
                        free=c(TRUE, FALSE), values=1, lbound=0.001,
                        labels=c("p1","p2"), name="classQuant" )
```

We still need probabilities, which require the second step shown below. Dividing the class quantity matrix above by its sum will rescale the quantities into probabilities. This is slightly more difficult than it appears at first, as the **k x 1** matrix of class quantities and the scalar sum of that matrix aren't conformable to either matrix or element-wise operations. Instead, we can use a Kronecker product of the class quantity matrix and the inverse of the sum of that matrix. This operation is carried out by the `mxAlgebra` function placed in the object *classP* below.

```
classP      <- mxAlgebra( classQuant %x% (1/sum(classQuant)), name="classProbs" )
```

There are several alternatives to the two functions above that merit discussion. While the `mxConstraint` function would appear at first to be a simpler way to specify the class probabilities, using the `mxConstraint` function complicates this type of model estimation. When all *k* class probabilities are freely estimated then constrained, then the class probability parameters are collinear, creating a parameter covariance matrix that is not of full rank. This prevents OpenMx from calculating standard errors for any model parameters. Additionally, there are multiple ways to use algebras different than the one above to specify the class quantity and/or class probability parameters, each varying in complexity and utility. While specifying models with two classes can be done slightly more simply than presented here, the above method is equally appropriate for all numbers of classes.

Finally, we can specify the mixture model. We must first specify the model's -2 log likelihood function defined as:

$$-2LL = -2 * \sum_{i=1}^{n} \sum_{k=1}^{m} \log(p_k l_{ki})$$

This is specified using an `mxAlgebra` function, and used as the argument to the `mxFitFunctionAlgebra` function. Then the fit function, matrices and algebras used to define the mixture distribution, the models for the respective classes and the data are all placed in one final `mxModel` object, shown below.

```
algFit       <- mxAlgebra( -2*sum(log(classProbs[1,1] %x% Class1.fitfunction
                                    + classProbs[2,1] %x% Class2.fitfunction)),
                           name="mixtureObj")
fit          <- mxFitFunctionAlgebra("mixtureObj")
dataRaw      <- mxData( observed=myGrowthMixtureData, type="raw" )

gmm          <- mxModel("Growth Mixture Model",
                        dataRaw, class1, class2, classP, classQ, algFit, fit )

gmmFit <- mxRun(gmm)

summary(gmmFit)
```

Multiple Runs: Serial Method

The results of a mixture model can sometimes depend on starting values. It is a good idea to run a mixture model with a variety of starting values to make sure results you find are not the result of a local minimum in the likelihood space. This section will describe a serial (i.e., running one model at a time) method for randomly generating starting values and re-running a model, which is appropriate for a wide range of methods. The next section will cover parallel (multiple models simultaneously) estimation procedures. Both of these examples are available in the `GrowthMixtureModelRandomStarts` demo.

- http://openmx.psyc.virginia.edu/svn/trunk/models/nightly/GrowthMixtureModelRandomStarts.R

One way to access the starting values in a model is by using the `omxGetParameters` function. This function takes an existing model as an argument and returns the names and values of all free parameters. Using this function on our growth mixture model, which is stored in an objected called `gmm`, gives us back the starting values we specified above.

```
    omxGetParameters(gmm)
#       pclass1 residual    vari1     cov1    vars1   meani1   means1
#           0.2      1.0      1.0      0.4      1.0      0.0     -1.0
#       vari2      cov2    vars2   meani2    means2
#           1.0      1.0      0.5      1.0      5.0
```

A companion function to `omxGetParameters` is `omxSetParameters`, which can be used to alter one or more named parameters in a model. This function can be used to change the values, freedom and labels of any parameters in a model, returning an MxModel object with the specified changes. The code below shows how to change the residual variance starting value from 1.0 to 0.5. Note that the output of the `omxSetParameters` function is placed back into the object `gmm`.

```
gmm <- omxSetParameters(gmm, labels="residual", values=0.5)
```

The MxModel in the object `gmm` can now be run and the results compared with other sets of staring values. Starting values can also be sampled from distributions, allowing users to automate starting value generation, which is demonstrated below. The `omxGetParameters` function is used to find the names of the free parameters and define three matrices: a matrix `input` that holds the starting values for any run; a matrix `output` that holds the converged values of each parameter; and a matrix `fit` that contains the -2 log likelihoods and other relevant model fit statistics. Each of these matrices contains one row for every set of starting values. Starting values are randomly generated from a set of uniform distributions using the `runif` function, allowing the ranges inherent to each parameter to be enforced (i.e., variances are positive, etc). A `for` loop repeatedly runs the model with starting values from the `input` matrix and places the final estimates and fit statistics in the `output` and `fit` matrices, respectively.

```
# how many trials?
trials       <- 20

# place all of the parameter names in a vector
parNames     <- names(omxGetParameters(gmm))
```

```
# make a matrix to hold all of the
input          <- matrix(NA, trials, length(parNames))
dimnames(input) <- list(c(1: trials), c(parNames))

output         <- matrix(NA, trials, length(parNames))
dimnames(output) <- list(c(1: trials), c(parNames))

fit            <- matrix(NA, trials, 5)
dimnames(fit) <- list(c(1: trials), c("Minus2LL", "Status", "Iterations", "pclass1", "time"))

# poulate the class probabilities
input[,"p1"]  <- runif(trials, 0.1, 0.9)
input[,"p1"]  <- input[,"p1"]/(1-input[,"p1"])

# populate the variances
v              <- c("vari1","vars1","vari2","vars2","residual")
input[,v]      <- runif(trials*5, 0, 10)

# populate the means
m              <- c("meani1", "means1", "meani2", "means2")
input[,m]      <- runif(trials*4, -5, 5)

# populate the covariances
r              <- runif(trials*2, -0.9, 0.9)
scale          <- c( sqrt(input[,"vari1"]*input[,"vars1"]), sqrt(input[,"vari2"]*input[,"vars2"]))
input[,c("cov1", "cov2")] <- r * scale

for (i in 1: trials){
    temp1     <- omxSetParameters(gmm, labels=parNames, values=input[i,], name = paste("Starting Value
    temp2     <- mxRun(temp1, unsafe=TRUE, suppressWarnings=TRUE, checkpoint=TRUE)
    output[i,] <- omxGetParameters(temp2)
    fit[i,] <- c(
        temp2@output$Minus2LogLikelihood,
        temp2@output$status[[1]],
        temp2@output$iterations,
        round(temp2$classProbs@result[1,1], 4),
        temp2@output$wallTime
        )
    }
```

Viewing the contents of the `fit` matrix shows the -2 log likelihoods for each of the runs, as well as the convergence status, number of iterations and class probabilities, shown below.

```
fit[,1:4]
#       Minus2LL Status Iterations   pclass1
#    1  8739.050      0         41 0.3991078
#    2  8739.050      0         40 0.6008913
#    3  8739.050      0         44 0.3991078
#    4  8739.050      1         31 0.3991079
#    5  8739.050      0         32 0.3991082
#    6  8739.050      1         34 0.3991089
#    7  8966.628      0         22 0.9990000
#    8  8966.628      0         24 0.9990000
#    9  8966.628      0         23 0.0010000
#    10 8966.628      1         36 0.0010000
#    11 8963.437      6         25 0.9990000
#    12 8966.628      0         28 0.9990000
#    13 8739.050      1         47 0.6008916
```

```
#    14 8739.050      1         36 0.3991082
#    15 8739.050      0         43 0.3991076
#    16 8739.050      0         46 0.6008948
#    17 8739.050      1         50 0.3991092
#    18 8945.756      6         50 0.9902127
#    19 8739.050      0         53 0.3991085
#    20 8966.628      0         23 0.9990000
```

There are several things to note about the above results. First, the minimum -2 log likelihood was reached in 12 of 20 sets of staring values, all with NPSOL statuses of either zero (seven times) or one (five times). Additionally, the class probabilities are equivalent within five digits of precision, keeping in mind that no model as specified contains no restriction as to which class is labeled "class 1" (probability equals .3991) and "class 2" (probability equals .6009). The other eight sets of starting values showed higher -2 log likelihood values and class probabilities at the set upper or lower bounds, indicating a local minimum. We can also view this information using R's `table` function.

```
table(round(fit[,1], 3), fit[,2])
```

```
#               0 1 6
#      8739.05  7 5 0
#      8945.756 0 0 1
#      8963.437 0 0 1
#      8966.628 5 1 0
```

We should have a great deal of confidence that the solution with class probabilities of .399 and .601 is the correct one.

Multiple Runs: Parallel Method

OpenMx supports multicore processing through the `snowfall` library, which is described in the "Multicore Execution" section of the documentation and in the following demo:

- http://openmx.psyc.virginia.edu/svn/trunk/models/passing/BootstrapParallel.R

Using multiple processors can greatly improve processing time for model estimation when a model contains independent submodels. While the growth mixture model in this example does contain submodels (i.e., the class specific models), they are not independent, as they both depend on a set of shared parameters ("residual", "pclass1").

However, multicore estimation can be used instead of the `for` loop in the above section for testing alternative sets of starting values. Instead of changing the starting values in the `gmm` object repeatedly, multiple copies of the model contained in `gmm` must be placed into parent or container model. Either the above `for` loop or a set of "apply" statements can be used to generate the model.

The example below first initializes the `snowfall` library, which also loads the `snow` library. The `sfInit` function initializes parallel; you must supply the number of processors on your computer or grid for the analysis, then reload OpenMx as a snowfall library.

```
require(snowfall)
sfInit(parallel=TRUE, cpus=4)
sfLibrary(OpenMx)
```

From there, parallel optimization requires that a holder or top model (named `Top` in the object *topModel* below) contain a set of independent submodels. In our example, each independent submodel will consist of a copy of the above `gmm` model with a different set of starting values. Using the matrix of starting values from the serial example above (`input`), we can create a function called `makeModel` that can be used to create these submodels. While this function is entirely optional, it allows us to use the `lapply` function to create a list of submodels for optimization. Once those submodels are placed in the `submodels` slot of the object *topModel*, we can run this model just like any other. A second function, `fitStats`, can then be used to get the results from each submodel.

```
topModel     <- mxModel("Top")

makeModel    <- function(modelNumber){
    temp     <- mxModel(gmm, independent=TRUE, name=paste("Iteration", modelNumber, sep=""))
    temp     <- omxSetParameters(temp, labels=parNames, values=input[modelNumber,])
    return(temp)
}

mySubs       <- lapply(1:20, makeModel)
topModel = mxModel(topModel, mySubs)

results <- mxRun(topModel)

fitStats <- function(model){
    retval <- c(
        model@output$Minus2LogLikelihood,
        model@output$status[[1]],
        model@output$iterations,
        round(model$classProbs@result[1,1], 4)
        )
    return(retval)
}

resultsFit <- t(omxSapply(results@submodels, fitStats))
sfStop()
```

This parallel method saves computational time, but requires additional coding. For models as small as the one in this example (total processing time of approximately 2 seconds), the speed-up from using the parallel version is marginal (approximately 35-50 seconds for the serial method against 20-30 seconds for the parallel version). However, as models get more complex or require a greater number of random starts, the parallel method can provide substantial time savings. Regardless of method, re-running models with varying starting values is an essential part of running multivariate models.

3.9 Likelihood, Matrix Specification

This example use an `mxAlgebra` likelihood function to compare the fit of two models for the population frequencies of the A, B and O blood groups. It is based on the textbook Likelihood, by Anthony William Fairbank Edwards (1972; 1984). Human beings may be classified according to which of four ABO blood groups they belong: A, B, AB, or O. These groups can be phenotypically distinguished by which antibodies they produce, and this is important for blood transfusions. Patients receiving blood containing an antigen that they do not produce have an adverse reaction to it.

Two hypotheses were advanced to account for the ABO variation. The single locus model posited that there exists one locus with three alleles, A, B and O. Allele A generates antibody A, allele B generates antibody B, and allele O generates neither. Under the two locus model, there is a diallelic A locus, with one allele that produces antibody A, and another that does not. At a second, unlinked, locus B, antigen B is generated by those who have at least one B allele. We try not to make jokes about people who have the BO genotype. For a more thorough account of the ABO blood group system, the Wikipedia article http://en.wikipedia.org/wiki/ABO_blood_group_system is a good starting point.

The two scripts are available in these files:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/OneLocusLikelihood.R

and

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/TwoLocusLikelihood.R

For (former) users of the first version of Mx, scripts for that software can be found here:

- http://openmx.psyc.virginia.edu/svn/trunk/inst/mx-scripts/one.mx
- http://openmx.psyc.virginia.edu/svn/trunk/inst/mx-scripts/two.mx

3.9.1 Single vs Two Locus Model

Data

Data from the German mathematician *Felix Bernstein <http://en.wikipedia.org/wiki/Felix_Bernstein>* (1924) are used for this example. Their observed frequencies are shown in the table, along with expected proportions in the population under the single locus model. A good exercise is to derive the expected proportions by hand. A key consideration is that several different genotypes (combinations of alleles) may generate a single phenotype (observed antigens in the blood). Due to the genetic dominance of allele A, both AA and AO genotypes produce the blood group A phenotype.

Single Locus Model Specification

The single locus dominant genetic model, with alleles A and B being codominant (resulting in the AB blood group phenotype) predicts that certain proportions of each blood group will occur in the population. These proportions depend on, and can be expressed as functions of, the allele frequencies. Let the frequencies of alleles A, B and O be p, q' and r, respectively. We assume that parents mate at random with respect to their blood groups, so that the likelihood of inheriting two A alleles becomes simply the product of the allele frequencies in the population, i.e., p^2. Thus we are able to draw up a table of the expected proportions of each blood group as a function of the parameters p, q' and r. These unknown parameters will be estimated from the data using numerical optimization, subject to the constraint that $p + q + r = 1$.

Phenotype	Genotypes	Frequency	Proportion
A	AO, AA	234	p(p+2r)
B	BO, BB	261	q(q+2r)
AB	AB	182	2pq
O	OO	94	r^2

The individuals in this sample are considered to be statistically independent. Therefore the likelihood of observing, e.g., 234 individuals with blood type A, is simply $(p(p + 2r))^{234}$ and the log-likelihood is $234 \log p(p + 2r)$. The OpenMx script will use the four log-likelihoods of the four phenotypes, and sum them to obtain the overall log-likelihood. Optimization proceeds by minimization by default, so we minimize the negative log-likelihood in order to maximize the log-likelihood. The following code block specifies this model. There

```
require(OpenMx)

#     Bernstein data on ABO blood-groups
#     c.f. Edwards, AWF (1972)  Likelihood.  Cambridge Univ Press, pp. 39-41

# Matrices for allele frequencies, p, q and r
matP         <- mxMatrix( type="Full", nrow=1, ncol=1,
                          free=TRUE, values=c(.3333), name="P")
matQ         <- mxMatrix( type="Full", nrow=1, ncol=1,
                          free=TRUE, values=c(.3333), name="Q")
matR         <- mxMatrix( type="Full", nrow=1, ncol=1,
                          free=TRUE, values=c(.3333), name="R")
# Matrix of observed data
const        <- mxConstraint(P+Q+R == 1, name="EqualityConstraint")
obsFreq      <- mxMatrix( type="Full", nrow=4, ncol=1,
                          values=c(211,104,39,148), name="ObservedFreqs")
# Algebra for predicted proportions
```

```
expFreq      <- mxAlgebra( expression=rbind(P*(P+2*R), Q*(Q+2*R), 2*P*Q, R*R),
                           name="ExpectedFreqs")
# Algebra for -2logLikelihood
m2ll         <- mxAlgebra( expression=-(sum(log(ExpectedFreqs) * ObservedFreqs)),
                            name="NegativeLogLikelihood")
# User-defined objective
funAl        <- mxFitFunctionAlgebra("NegativeLogLikelihood")

OneLocusModel <- mxModel("OneLocus",
                           matP, matQ, matR, const, obsFreq, expFreq, m2ll, funAl)

OneLocusFit <- mxRun(OneLocusModel)
OneLocusFit$matrices
OneLocusFit$algebras
```

Answers should be 0.2945 0.1540 0.5515 for the allele frequencies p, q and r, respectively, and 627.104 for the negative log-likelihood. We now turn to the alternative two-locus model.

Two Locus Model Specification

Under the two locus model, we allow for two unlinked (i.e. segregating independently of each other) diallelic loci, A and B. We denote the O allele as a at the A locus, and as b at the B locus, so as to distinguish between these two alleles, neither of which generates an antigen. Thus genotypes at the A locus can be **AA**, **Aa**, or **aa**, with genotype frequencies p^2, $2pq$ and q^2, where p is the proportion of allele p in the population, and $q = 1 - p$ is the proportion of allele a. Similarly, genotypes at the B locus can be **BB**, **Bb** or **bb**, with genotype frequencies s^2, $2st$ and t^2, given allele frequencies s and t, respectively. Due to the dominance of A over a and B over b, only those with **aabb** genotypes will belong to blood group O (no antigens). The number the genotype combinations which generate a particular blood group is generally larger than under the single locus model. The combinations, and their expected frequencies in the population, are given in the following table:

Phenotype	Genotypes	Frequency	Proportion
A	AAbb, Aabb	234	$(p^2 + 2pq)$*:math:t^2
B	aaBB, aaBb	261	q^2‘(: $math$: ‘$s^2 + 2st$)
AB	AABB, AABb, AaBB, AaBb	182	$(p^2 + 2pq)(s^2 + 2st)$
O	aabb	94	q^2t^2

The R script to fit this model is very similar to that of the single locus model. Note, however, that it does not feature the `mxConstraint` function. There are in fact two constraints, $q = 1 - p$ and $t = 1 - s$, but these are trivial and easily dealt with using `mxAlgebra` statements. Although one might think that this approach would be suitable for the single locus model, in which $r = 1 - p - q$, a difficulty arises because there is no straightforward way to restrict $p + q \leq 1$ which is necessary for $r \geq 0$. Models specified so that an allele frequency can go negative during optimization are inherently fragile. A negative allele frequency would potentially result in negative likelihoods, and undefined log-likelihoods. Bounding the parameters to lie between 0.0 and 1.0 provides sufficient robustness to this potential problem.

```
require(OpenMx)

#     Bernstein data on ABO blood-groups
#     c.f. Edwards, AWF (1972)  Likelihood.  Cambridge Univ Press, pp. 39-41

# Matrices for allele frequencies, p and s
matP         <- mxMatrix( type="Full", nrow=1, ncol=1,
                          free=TRUE, values=c(.3333), name="P")
matS         <- mxMatrix( type="Full", nrow=1, ncol=1,
                          free=TRUE, values=c(.3333), name="S")
# Matrix of observed data
```

```
obsFreq      <- mxMatrix( type="Full", nrow=4, ncol=1,
                          values=c(211,104,39,148), name="ObservedFreqs")
matQ         <- mxAlgebra( expression=1-P, name="Q")
matT         <- mxAlgebra( expression=1-S, name="T")
# Algebra for predicted proportions
expFreq      <- mxAlgebra( rbind ((P*P+2*P*Q)*T*T, (Q*Q)*(S*S+2*S*T),
                          (P*P+2*P*Q)*(S*S+2*S*T), (Q*Q)*(T*T)), name="ExpectedFreqs")
# Algebra for -2logLikelihood
m2ll         <- mxAlgebra( expression=-(sum(log(ExpectedFreqs) * ObservedFreqs)),
                           name="NegativeLogLikelihood")
# User-defined objective
funAl        <- mxFitFunctionAlgebra("NegativeLogLikelihood")

TwoLocusModel <- mxModel("TwoLocus",
                          matP, matS, matQ, matT, obsFreq, expFreq, m2ll, funAl)

TwoLocusFit<-mxRun(TwoLocusModel)
TwoLocusFit$matrices
TwoLocusFit$algebras
```

Results

The allele frequencies estimated by this script should be $p = 0.2929, s = 0.1532$ with negative log-likelihood of 646.972 units. Comparison of this model with the single locus one shows that although they have the same number of free parameters (the third allele frequency in the single locus model is constrained) the single locus model has much greater support. Investigation of the `$ExpectedFreqs` algebra in the two models helps to illustrate why.

CHAPTER

FOUR

ITEM FACTOR ANALYSIS

4.1 What is Item Factor Analysis?

4.1.1 An Intuitive Appeal

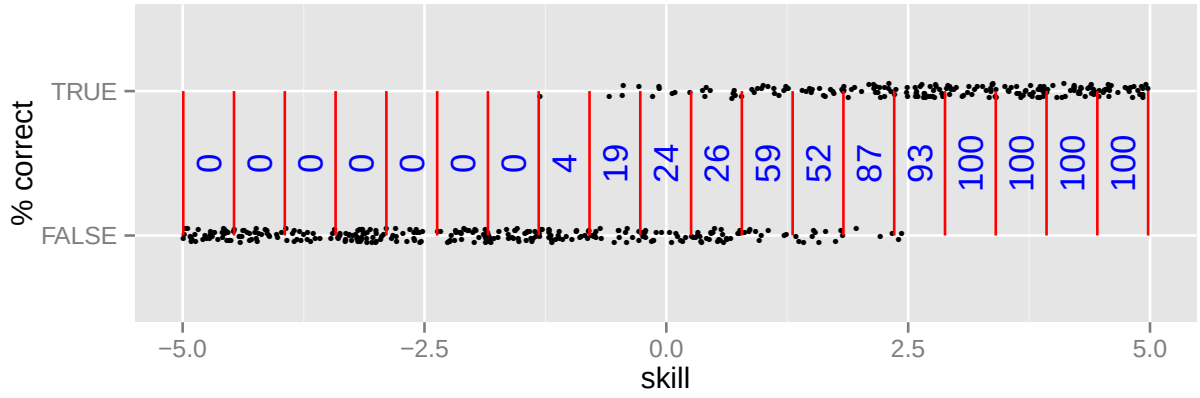


Figure 4.1: Percentage correct of responses by skill bin
Bins are demarcated by red vertical bars. This procedure converts nominal or ordinal data (the black dots) into interval scale data conditional on examinee skill. The blue numbers in the middle of the plot are the interval data.

Educational assessments are often scored *correct* and *incorrect*. Some items may also offer partial credit for partially correct answers. One approach to analyze these ordinal data is Classical Test Theory (CTT). CTT includes statistics like Cronbach's α for estimating reliability, proportion correct as an estimate of item difficulty, and item-total correlation as an estimate of item discrimination. However, a disadvantage of CTT is the need for huge samples to establish test norms. Modern Test Theory, also known as Item Response Theory or Item Factor Analysis (IFA), offers substantial advantages over CTT such as the ability to obtain unbiased item parameter estimates from an unrepresentative sample [Embretson1996].

IFA is based on the idea of fitting models to data. The first step is to convert nominal or ordinal data into interval scale data. To understand how this is accomplished, assume that the true skill (traditionally θ) of participants is known. We can plot item outcome by true skill, partition the skill distribution into bins, and count the proportion of correct responses in each bin (Figure 4.1). We can fit a model to these data. One popular model is the logistic model (also called the 1PL or Rasch model),

$$\begin{aligned} Pr(pick = 0|c, \theta) &= 1 - Pr(pick = 1|c, \theta) \\ Pr(pick = 1|c, \theta) &= \frac{1}{1 + \exp(-(\theta - c))} \end{aligned}$$

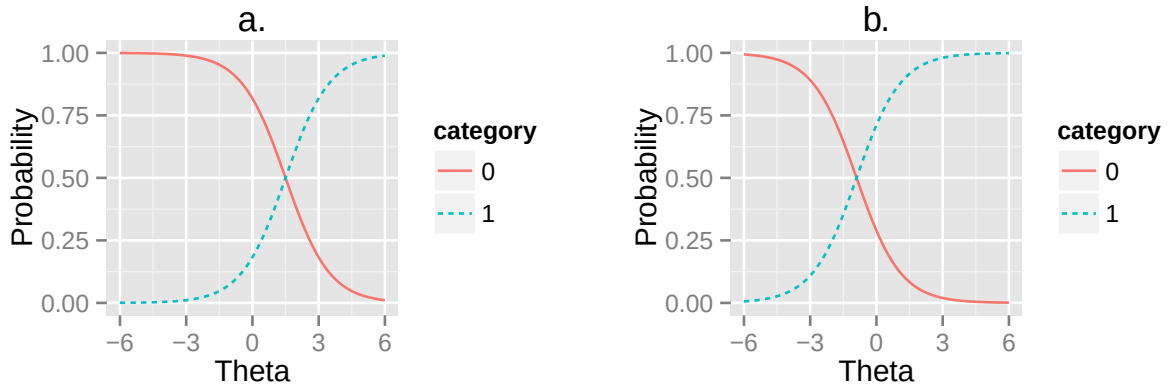


Figure 4.2: Example plots of the logistic curve

Example plots of the logistic curve (blue) and its complement (red). The interval data are fit against the blue curve. The c parameter is set to 1.5 for (a) and -0.9 for (b). Which curve would fit the data better? Answer: Curve (a) is a better fit because the 50% level occurs near skill=1.5 in Figure 4.1.

where θ is the participant's skill and c is the estimated parameter to describe the item's difficulty (Figure 4.2). While the examinee's true skill is assumed known in this simplified introduction, such an assumption is unnecessary. There is a circular dependency. Items parameters depend on examinee skill which, in turn, depend on item parameters. These kinds of models cannot be optimized directly. However, optimization is possible by switching back and forth between item and person parameters [BockAitkin1981].

4.1.2 The Likelihood of Item Models

The best way to appreciate the assumptions involved in Item Factor Analysis is to examine how the likelihood is traditionally computed. The conditional likelihood of response x_{ij} to item j from person i with item parameters ξ_j and latent ability θ_i is

$$L(x_i|\xi,\theta_i) = \prod_j \Pr(\text{pick} = x_{ij}|\xi_j,\theta_i).$$

One implication of this equation is that items are assumed conditionally independent. That is, the outcome of one item does not have any influence on another item after controlling for ξ and θ_i. The unconditional likelihood is obtained by integrating over the latent distribution θ_i,

$$L(x_i|\xi) = \int L(x_i|\xi,\theta_i)L(\theta_i)\mathrm{d}\theta_i.$$

Typically θ_i is distributed as multivariate Normal. Other distributions are possible, but are not implemented in `OpenMx` at the time of this writing. With an assumption that examinees are independently and identically distributed, we can sum the individual log likelihoods,

$$\mathcal{L} = \sum_i \log L(x_i|\xi).$$

One important observation about this model is that there are two kinds of parameters. There are item parameters ξ and latent distribution parameters θ. For didactic purposes, we will start with models for item parameters ξ and neglect the latent distribution, assuming that the latent distribution is standard multivariate Normal. Once there is some experience with item models, we will fix item parameters and focus on estimating latent distribution parameters. Finally, an example will be given of a model that estimates both item and latent distribution parameters simultaneously.

4.2 Item Models

This chapter assumes that you have already read the *Basic Introduction*.

4.2.1 A Rasch model

Suppose you regularly administer the PANAS [WatsonEtal1988], but instead of scoring participants by adding up the item scores, you want to try IFA. Here is how you might do it. Without loss of generality, we will only consider the positive affect part of the scale.

```
library(OpenMx)
library(rpf)

PANASItem <- c("Very Slightly or Not at All",  "A Little",
               "Moderately", "Quite a Bit",    "Extremely")
spec <- list()
spec[1:10] <- rpf.grm(outcomes = length(PANASItem))  # grm="graded response model"

# replace with your own data
data <- rpf.sample(750, spec, sapply(spec, rpf.rparam))

for (cx in 1:10) levels(data[[cx]]) <- PANASItem          # repair level labels
colnames(data) <- c("interested", "excited", "strong", "enthusiastic", "proud",
                    "alert", "inspired", "determined", "attentive", "active")
head(data)  # much easier to understand with labels
origData <- data

startingValues <- matrix(c(1, seq(1,-1,length.out=4)), ncol=length(spec), nrow=5)
imat <- mxMatrix(name="item", values=startingValues,
                 free=TRUE, dimnames=list(names(rpf.rparam(spec[[1]])), colnames(data)))
rownames(imat)[1] <- "posAff"
imat$labels[1,] <- 'slope'

originalPanas1 <- mxModel(model="panas1", imat,
           mxData(observed=data, type="raw"),
           mxExpectationBA81(ItemSpec=spec, item="item"),
           mxFitFunctionML(),
           mxComputeEM('expectation', 'scores', mxComputeNewtonRaphson()))
panas1 <- mxRun(originalPanas1)
```

A PANAS item always has 5 possible outcomes, but best practice is to list the outcomes labels and let R count them for you. If you work on a new measure then you will appreciate the ease with which your script can adapt to adding or removing outcomes. Even for PANAS, we could try collapsing two outcomes and see how the model fit changes.

```
spec <- list()
spec[1:10] <- rpf.grm(outcomes = length(PANASItem))
```

The `rpf.grm` function creates an `rpf.base` class object that represents an item response function. An item response function assigns probabilities to response outcomes. The `grm` in the function name `rpf.grm` stands for *graded response model*. You can inspect the mathematical formula for the graded response model by requesting the manual page with `?rpf.grm`. Experiment with the `rpf.*` functions to get a feel for how they work.

```
rpf.rparam(spec[[1]])     # generates random parameters
rpf.numParam(spec[[1]])   # same as length(rpf.rparam(spec[[1]]))
rpf.paramInfo(spec[[1]]) # type and default upper & lower bounds
rpf.prob(spec[[1]], startingValues[,1], 0)    # probabilities at score=0
rpf.prob(spec[[1]], startingValues[,1], .5)   # probabilities at score=.5
```

```
data <- rpf.sample(750, spec, sapply(spec, rpf.rparam))
```

This last line (`rpf.sample`) creates fake data for 750 random participants based on our list of item models and random item parameters. Instead of this line, you would typically read in your data using `read.csv` and convert it to ordered factors using `mxFactor`.

```
startingValues <- matrix(c(1, seq(1,-1,length.out=4)), ncol=length(spec), nrow=5)
```

We can input particular starting values. That is what we do here. Every column is an item and every row is a different parameter. Regardless of item model, the first rows are factor loadings and the remaining parameters have meanings dependent on the item model. The graded response model is a little finicky; the threshold parameters must be ordered. Alternately, a good way to obtain random starting values is with,

```
startingValues <- mxSimplify2Array(lapply(spec, rpf.rparam))
startingValues[1,] <- 1  # these parameters must be equal
```

This is convenient because it will work for a non-homogeneous list of items. There are various circumstances where you will need to fix some of the starting values to particular values. For example:

- To constrain the nominal model to act like the generalized partial credit model you will set $\alpha_i = 0 \; \forall i > 1$.
- When you have more than 1 factor, you may know a priori that some items do not load on certain factors.

If you need to set some starting values to something specific then you might start with random starting values and then override any rows and columns as needed.

```
imat <- mxMatrix(name="item", values=startingValues,
                 free=TRUE, dimnames=list(names(rpf.rparam(spec[[1]])), colnames(data)))
rownames(imat)[1] <- "posAff"
```

The `item` matrix contains response probability function parameters in columns. This layout can be a little awkward when you estimate a mixed format measure with different numbers of outcomes for each item. Fortunately, all the PANAS items are the same. We must label all the rows and columns. The first row must be labeled with the name of our factor, `posAff`. The remaining rows can take any label. Since all of our items are the same, we can use the default item parameter names. The column names must match the column names of the data.

```
imat$labels[1,] <- 'slope'
```

Here we set the label of every parameter in the first row to `slope`. This is an equality constraint. With this constraint, we assume all items work equally well at measuring the latent trait. This constraint is what makes the difference between a Rasch model and any other kind of IFA model. A Rasch model makes this assumption.

```
panas1 <- mxModel(model="panas1", imat,
           mxData(observed=data, type="raw"),
           mxExpectationBA81(ItemSpec=spec, item="item"),
           mxFitFunctionML(),
           mxComputeEM('expectation', 'scores', mxComputeNewtonRaphson()))
```

Here we put everything together. There are a few things that are new.

```
mxComputeEM('expectation', 'scores', mxComputeNewtonRaphson())
```

The `mxComputeEM` plan is a somewhat more sophisticated version of

```
mxComputeIterate(steps=list(
                 mxComputeOnce('expectation', 'scores'),
                 mxComputeNewtonRaphson(),
                 mxComputeOnce('expectation'),
                 mxComputeOnce('fitfunction', 'fit')))
```

In both compute plans, OpenMx will iterate until the change in the fitfunction is less than some threshold. For every iteration, person scores are predicted by the expectation, a Newton-Raphson optimization takes place to improve the item parameter values, the predicted person scores are discarded, and the fitfunction is evaluated. In comparison to `mxComputeIterate`, the `mxComputeEM` plan offers additional options to speed up convergence and estimate standard errors.

After running the model, we can inspect the parameters estimates,

```
panas1 <- mxRun(originalPanas1)
panas1$item$values
# or
summary(panas1)
```

At this point, you might notice something unsettling about the summary output. No standard errors are reported. How do we know whether our model converged? Excellent question. There are a few things that we can check. We can look at the count of EM cycles and M-step Newton-Raphson iterations.

```
panas1$compute$output
```

If the number of EM cycles is 2 or less then it is likely that the parameters are still sitting at their starting values. Since our starting values were mostly random, that's probably not the solution we were looking for. Instead of digging into the `MxComputeEM` output, we can also re-run the model with extra diagnostics enabled.

```
panas1 <- mxModel(model=originalPanas1,
               mxComputeEM('expectation', 'scores', mxComputeNewtonRaphson(),
                           verbose=2L))
panas1 <- mxRun(panas1)
```

Note the addition of `verbose=2L` to `mxComputeEM`. The `mxComputeNewtonRaphson` object takes a `verbose` parameter as well if you want to examine the progress of the optimization in (much) more detail. From this diagnostic output, we discern that the parameters are changing, but we still do not know whether the solution is a candidate global optimum.

```
info1 <- mxModel(panas1,
              mxComputeSequence(steps=list(
                mxComputeOnce('fitfunction', 'information', "meat"),
                mxComputeStandardError(),
                mxComputeHessianQuality())))
info1 <- mxRun(info1)
```

As a starting point, we can estimate the information matrix by taking the inverse of the covariance of the per-row first derivatives. This estimate is called `meat` because it forms the inside of a sandwich covariance matrix [White1994]. The first thing to look at is the condition number of the information matrix.

```
info1$output$conditionNumber
```

A finite condition number implies that the information matrix is positive definite. Since the condition number is roughly closer to zero than to positive infinity, there is a good chance that the parameters are at a candidate global optimum. We can examine the standard errors.

```
summary(info1)
```

Further diagnostics are available from the RPF package. Many of these diagnostic functions are most convenient to use when all the relevant information is packaged up into an IFA group object. A convenient way to create an IFA group object is to use `as.IFAgroup`.

```
panas1Grp <- as.IFAgroup(panas1)
```

We know from inspection of the likelihood equation that IFA models assume that items are conditionally independent. That is, the outcome on a given item only depends on its item parameters and examinee skill, not on the outcome of

other items. At least some attempt should be made to check this assumption.

```
ChenThissen1997(panas1Grp)
```

Item pairs that exhibit statistically significant local dependence and positively correlated residuals should be investigated. If ignored, local dependence exaggerates the accuracy of measurement. Standard errors will be smaller and items will seem to fit the data better than they otherwise would [Yen1993].

Since we have a single factor Rasch model, the residuals are easily interpretable. We can examine Rasch fit statistics *infit* and *outfit*. Before we do that, however, we need to compute EAP scores.

```
panas1Grp$scores <- EAPscores(panas1Grp)
```

```
rpf.1dim.fit(group=panas1Grp, margin=2)
```

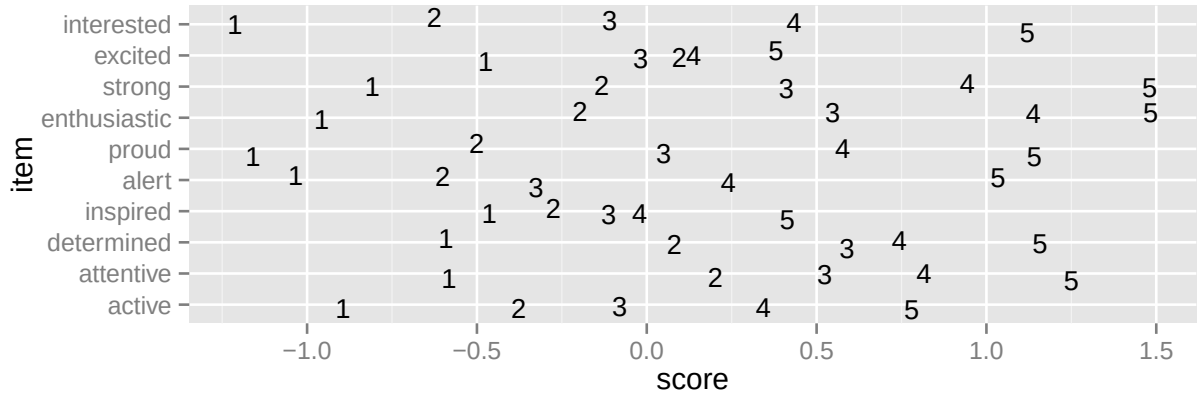


Figure 4.3: Example item map
Outcomes located at the mean of the ability of every examinee who picked that outcome.

This gives us item-wise statistics. For person-wise statistics, we can replace `margin=2` with `margin=1`. For some discussion on the interpretation of these statistics, visit the Infit and Outfit page at the Institute for Objective Measurement. Another way to look at the results is to create an item plot. An item plot assigns to every outcome the mean of the ability of every examinee who picked that outcome (Figure 4.3).

```
sumScoreEAP(panas1Grp)
```

Finally, we can generate a sum-score EAP table. The `posAff` column contains the interval-scale score corresponding to the row-wise sum-score. You could use this table to score the PANAS instead of merely using the sum-score. The EAP sum-score will likely provide higher accuracy measurement of the latent trait *positive affect*.

4.2.2 A 2PL model

Suppose you are skeptical that all positive affect items work equally well at measuring positive affect. We can relax this assumption and let the optimizer estimate how well each item is working. Continuing our previous example, all that is needed is to remove the label from the item parameter matrix.

```
panas2 <- mxModel(model=panas1,
            mxComputeSequence(list(
            mxComputeEM('expectation', 'scores', mxComputeNewtonRaphson()),
            mxComputeOnce('fitfunction', 'information', "meat"),
            mxComputeStandardError(),
            mxComputeHessianQuality())))
```

```
panas2$item$labels[1,] <- NA  # here we remove the label
panas2 <- mxRun(panas2)
```

The compute plan is the same as before except that we combined both the `mxComputeEM` fit step and computation of standard errors in a single `mxComputeSequence`. As usual, we start by inspecting the condition number of the Hessian then look at the parameter estimates with standard errors.

```
panas2$output$conditionNumber
summary(panas2)
```

Since these models are nested, we can conduct a likelihood ratio test to determine whether `panas2` fits the data significantly better than our Rasch constrained model `panas1`.

```
mxCompare(panas2, panas1)
```

Rasch fit statistics are not appropriate for a 2PL model because residuals from different items have different weights. However, we can compare expected item outcome proportions against sum-scores. First we need to create IFA group object for `panas2` then we can run the S test [OrlandoThissen2000].

```
panas2Grp <- as.IFAgroup(panas2)
SitemFit(panas2Grp)
```

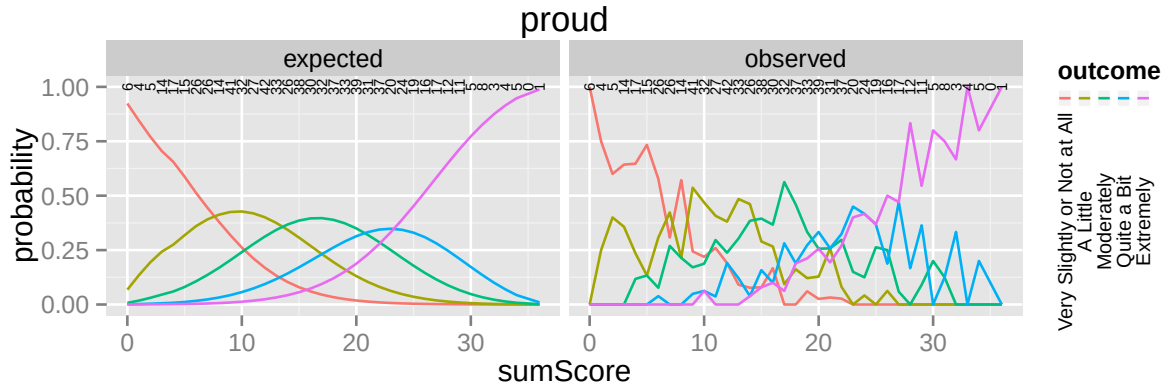


Figure 4.4: Expected vs observed outcome frequencies at each sum-score level

The observed trace lines bounce around because this is a small sample size. The per-sum-score sample size is given along the `probability=1` line. One reason that this is an interesting plot is because the plot is 2 dimensional regardless of the number of latent factors.

The internal tables of `SitemFit` can easily be plotted (Figure 4.4). Sometimes it is easier to diagnose the source of misfit by examining such a plot than by inspection of large tables of probabilities. However, the S test is not a panacea. Poorly fitting items will cause other items to fit poorly. The S test is just one more tool in the toolbox.

4.2.3 A 3PL with Bayesian priors

The 3PL item model is similar to the 2PL model except with an additional lower asymptote to represent the chance of getting the item correct by guessing. Estimation of 3PL models usually requires very large samples or the use of Bayesian priors on the lower asymptote. For an item with TRUE/FALSE outcomes, the chance of guessing correct is .5. In this case, the mode of the prior would be set to .5. In general, the mode is set to the reciprocal of the number of possible outcomes.

Unlike most other IFA software, `OpenMx` can accommodate a prior of any form. This flexibility is marvelous, however, it may require some additional effort to set up priors in comparison to other software. Typically the same family

of priors would be applied to all asymptote parameters. For didactic purposes, here we will implement a beta prior on half of the asymptote parameters and a Gaussian prior on the remainer.

Asymptote parameters are estimated in logit units. The advantage of logit units is that there is no need to complicate the optimization with upper and lower bounds. To interpret the estimates, asymptote parameters should be transformed back into probability units using the logistic function, $(1 + \exp(-g))^{-1}$.

```
spec <- list()
spec[1:8] <- rpf.drm()  # drm="dichotomous response model"

# replace with your own data
simParam <- sapply(spec, rpf.rparam)
simParam['u',] <- logit(1)     # fix upper bound at 1 for a 3PL equivalent model
data <- rpf.sample(750, spec, simParam)

imat <- mxMatrix(name="item", values=c(1,0,logit(.1),logit(1)),
                 free=c(TRUE, TRUE, TRUE, FALSE), nrow=4, ncol=length(spec),
                 dimnames=list(names(rpf.rparam(spec[[1]])), colnames(data)))

# label the pseudo-guessing parameters
imat$labels['g',] <- paste('g',1:length(spec),sep="")

# half of the items get a beta prior
betaRange <- 1:(length(spec)/2)

# the other half get a Gaussian prior
gaussRange <- (1+length(spec)/2):length(spec)

# Create matrices that contain only a subset of the parameters from
# the item matrix so the priors are easier to set up.
betaPrior <- mxMatrix(name="betaPrior", nrow=1, ncol=length(betaRange),
                free=TRUE, labels=imat$labels['g',betaRange],
                values=imat$values['g',betaRange])
gaussPrior <- mxMatrix(name="gaussPrior", nrow=1, ncol=length(gaussRange),
                free=TRUE, labels=imat$labels['g',gaussRange],
                values=imat$values['g',gaussRange])
```

For beta parameters $a = \alpha - 1 > 0$ and $b = \beta - 1 > 0$, the beta density for logit transformed parameter g is

$$\frac{1}{\mathrm{Beta}(\alpha, \beta)} \left[\frac{1}{(1 + \exp(-g))}\right]^a \left[1 - \frac{1}{1 + \exp(-g)}\right]^b.$$

After application of $-2 \log$ and simplifying, we obtain

$$2(b + a) \log(\exp(g) + 1) + ag + \log(\mathrm{Beta}(\alpha, \beta))$$

and derivatives with respect to g are

$$\begin{aligned} \frac{\partial}{\partial g} &= 2\left(b - \frac{(b + a)}{\exp(g) + 1}\right) \\ \frac{\partial^2}{\partial g^2} &= 2\frac{(b + a)\exp(g)}{(\exp(g) + 1)^2}. \end{aligned}$$

The mode of the beta density is $\frac{a}{a+b}$ and we can regard $a + b$ as the informative strength of the prior. For a given mode and strength, we obtain $a = mode * strength$ and $b = strength - a$. It is often helpful to look at a plot (e.g., Figure 4.5) to develop your mathematical imagination.

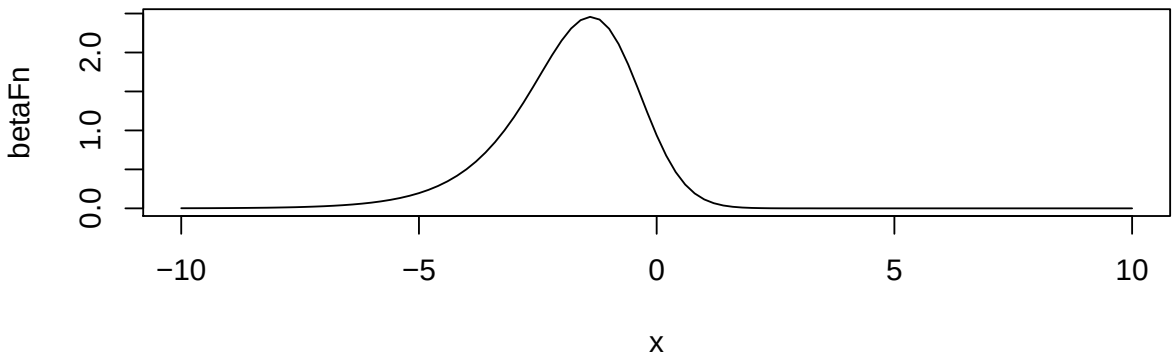


Figure 4.5: Beta prior of strength 5 with mode logit(1/5) on the logit scale
Here we plot `function (x) dbeta(1/(1+exp(-x)), 1+betaParam['a',4], 1+betaParam['b',4])` from logit(-10) to logit(10). An exactly quadratic density, like the Gaussian, is slightly easier for the optimizer to handle in comparison to the beta density.

```
calcBetaParam <- function(mode, strength) {
  a <- mode * strength
  b <- strength - a
  c(a=a, b=b, c=log(beta(a+1,b+1)))
}

guessChance <- c(1/2, 1/3, 1/4, 1/5)
betaParam <- sapply(guessChance, calcBetaParam, strength=5)

# copy our betaParam table into OpenMx row vectors
betaA <- mxMatrix(name="betaA", nrow=1, ncol=length(betaRange), values=betaParam['a',])
betaB <- mxMatrix(name="betaB", nrow=1, ncol=length(betaRange), values=betaParam['b',])
betaC <- mxMatrix(name="betaC", nrow=1, ncol=length(betaRange), values=betaParam['c',])

# implement the math given above
betaFit <- mxAlgebra(2 * sum((betaA + betaB)*log(exp(betaPrior)+1) -
             betaA * betaPrior + betaC), name="betaFit")
betaGrad <- mxAlgebra(2*(betaB-(betaA+betaB) / (exp(betaPrior) + 1)),
             name="betaGrad", dimnames=list(c(),betaPrior$labels))
betaHess <- mxAlgebra(vec2diag(2*exp(betaPrior)*(betaA+betaB) / (exp(betaPrior) + 1)^2), name="betaHe
             dimnames=list(betaPrior$labels, betaPrior$labels))

# Create a model that will evaluate to the log likelihood of the beta prior
# and provide suitable derivatives for the optimizer.
betaModel <- mxModel(model="betaModel", betaPrior, betaA, betaB, betaC,
             betaFit, betaGrad, betaHess,
             mxFitFunctionAlgebra("betaFit", gradient="betaGrad", hessian="betaHess"))
```

In comparison to a beta prior, a Gaussian prior is somewhat easier to set up. A rationale for use of the Gaussian is given in [CaiYangHansen2011]. The mean of the prior is set to the desired mode and the standard deviation can be regarded as the strength of the prior. A standard deviation of 0.5 was suggested by [CaiYangHansen2011].

```
# These are the prior parameter row vectors.
gaussM <- mxMatrix(name="gaussM", nrow=1, ncol=length(gaussRange), values=logit(guessChance))
gaussSD <- mxMatrix(name="gaussSD", nrow=1, ncol=length(gaussRange), values=.5)

# The single variable Gaussian density and derivatives are well known mathematical results.
gaussFit <- mxAlgebra(sum(log(2*pi) + 2*log(gaussSD) +
             (gaussPrior-gaussM)^2/gaussSD^2), name="gaussFit")
gaussGrad <- mxAlgebra(2*(gaussPrior - gaussM)/gaussSD^2, name="gaussGrad",
             dimnames=list(c(),gaussPrior$labels))
gaussHess <- mxAlgebra(vec2diag(2/gaussSD^2), name="gaussHess",
                   dimnames=list(gaussPrior$labels, gaussPrior$labels))

# Create a model that will evaluate to the log likelihood of the Gaussian prior
# and provide suitable derivatives for the optimizer.
gaussModel <- mxModel(model="gaussModel", gaussPrior, gaussM, gaussSD,
                  gaussFit, gaussGrad, gaussHess,
                  mxFitFunctionAlgebra("gaussFit", gradient="gaussGrad", hessian="gaussHess"))

itemModel <- mxModel(model="itemModel", imat,
            mxData(observed=data, type="raw"),
            mxExpectationBA81(spec),
            mxFitFunctionML())

demo3pl <- mxModel(model="demo3pl", itemModel, betaModel, gaussModel,
                 mxFitFunctionMultigroup(groups=c('itemModel.fitfunction', 'betaModel.fitfunction',
                   'gaussModel.fitfunction')),
                 mxComputeSequence(list(
                   mxComputeEM('itemModel.expectation', 'scores', mxComputeNewtonRaphson()),
                   mxComputeNumericDeriv(),
                   mxComputeHessianQuality(),
                   mxComputeStandardError()
                 )))

demo3pl <- mxRun(demo3pl)
```

We cannot use the covariance of the per-row first derivatives to approximate the information matrix because it is not clear how to incorporate the effect of the priors. Instead, we use `mxComputeNumericDeriv()`, an implementation of Richardson extrapolation.

```
mxFitFunctionMultigroup(groups=c('itemModel.fitfunction', 'betaModel.fitfunction',
  'gaussModel.fitfunction'))
```

The fitfunction used in the model (`mxFitFunctionMultigroup`) simply adds the log likelihoods together. Some other IFA software may or may not include the log likelihood of the priors when reporting the log likelihood of the whole model. In `OpenMx`, it is more convenient to compute the complete log likelihood of the model including the priors. Although `OpenMx` models are a bit more work to set up, since you are required to specify the model exactly, you will feel confident that you know what you are doing.

```
betaHess <- mxAlgebra(vec2diag(2*(betaA+betaB)*exp(betaPrior) / (exp(2*betaPrior) +
            2*exp(betaPrior) + 1)), name="betaHess",
            dimnames=list(betaPrior$labels, betaPrior$labels))

gaussHess <- mxAlgebra(vec2diag(2/gaussSD^2), name="gaussHess",
                   dimnames=list(gaussPrior$labels, gaussPrior$labels))
```

In `betaHess` and `gaussHess`, the use of `vec2diag` is recognized and handled specially to facilitate block-wise inversion of the Hessian. Ensure `vec2diag` is the last step of the `mxAlgebra` computation. For example, the Hessian inversion code will not recognized `mxAlgebra(2*vec2diag(...))`. The block-wise inversion code is

one of the main optimizations that makes it practical to estimate models that include hundreds of items.

4.3 Latent Distribution Models

4.3.1 A Single Latent Factor

Suppose an enterprising researcher has administered the PANAS to large sample from the general population, fit an item model to these data, and published item parameters. You are testing an experimental intervention that should increase positive affect. You have 35 participants so far and wish to check whether your sample has significantly more positive affect than the general population mean.

```
# the item parameters you received are population parameters
panas2$item$free[,] <- FALSE

# replace with your own data
trait <- rnorm(35, .75, 1.25)
data <- rpf.sample(trait, grp=panas2Grp)

# set up the matrices to hold our latent free parameters
m.mat <- mxMatrix(name="mean", nrow=1, ncol=1, values=0, free=TRUE)
rownames(m.mat) <- "posAff"
cov.mat <- mxMatrix(name="cov", nrow=1, ncol=1, values=diag(1), free=TRUE)
dimnames(cov.mat) <- list("posAff", "posAff")

panasModel <- mxModel(model=panas2, m.mat, cov.mat,
                      mxData(observed=data, type="raw"), name="panas")

latentModel <- mxModel(model="latent",
                    mxDataDynamic(type="cov", expectation="panas.expectation"),
                    mxExpectationNormal(covariance="panas.cov", means="panas.mean"),
                    mxFitFunctionML())

e1Model <- mxModel(model="experiment1", panasModel, latentModel,
                   mxFitFunctionMultigroup(c("panas.fitfunction", "latent.fitfunction")),
                   mxCI("panas.mean"),
                   mxComputeSequence(list(
                     mxComputeEM('panas.expectation', 'scores',
                                 mxComputeGradientDescent(fitfunction="latent.fitfunction")),
                     mxComputeConfidenceInterval())))

e1Model <- mxRun(e1Model)
summary(e1Model)
```

The optimizer should have no difficulty with this model. Estimation of a mean and variance is one of the easiest problems that an optimizer can be asked to solve. There is no real need to check the quality of the information matrix. Here we are interested in whether the mean is different from 0. Therefore, we use `mxCI` and `mxComputeConfidenceInterval`. This confidence interval is likelihood-based and is equivalent to a likelihood ratio test against a nested model with the mean constrained to 0.

```
mxDataDynamic(type="cov", expectation="panas.expectation")
```

The `mxDataDynamic` in a special adapter to cause the latent model to use `panas.expectation` as a data source of type `cov`. Not any `MxExpectation` can be used in this way. However, `mxExpectationBA81` knows how to provide a Normal distribution as data.

```
mxExpectationNormal(covariance="panas.cov", means="panas.mean")
```

The arguments to `mxExpectationNormal` are the names of the model expected or output matrices. However, recall that the likelihood of an IFA model is conditional on the latent distribution (see *The Likelihood of Item Models*). While these matrices are output from the point of view of `mxExpectationNormal`, they are input from the point of view of `mxExpectationBA81`. During optimization, the `panasModel` likelihood must be recomputed whenever the mean or covariance change until the estimates approach a fixed point. This behavior can be confirmed by passing `verbose=2L` to `mxExpectationBA81`.

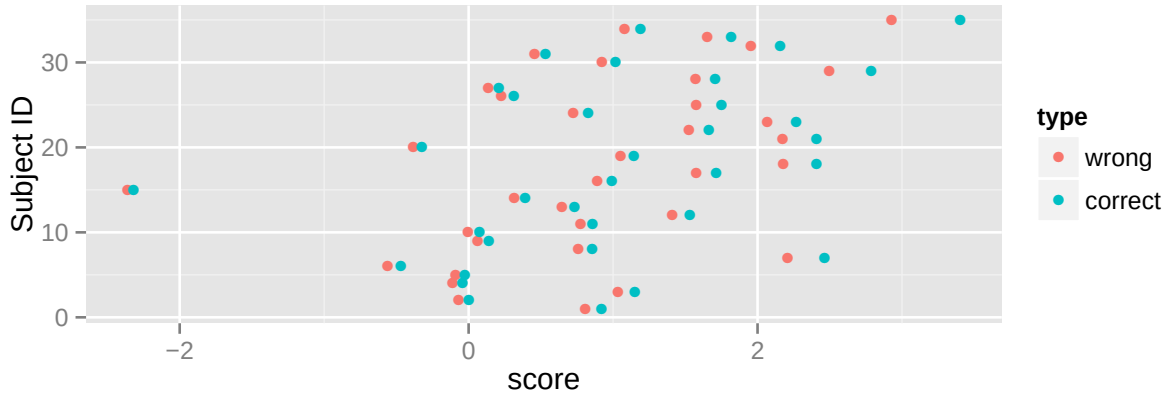


Figure 4.6: EAP scores with a standard Normal latent distribution (wrong) and estimated Normal distribution (correct).

```
e1Grp <- list(spec=panas2Grp$spec,
              param=panas2Grp$param,
              data=data)
e1Grp <- panas2Grp
e1Grp$data <- data
s1 <- EAPscores(e1Grp)[,1]   #wrong

e1Grp <- as.IFAgroup(e1Model$panas)
s2 <- EAPscores(e1Grp)[,1]   #correct
```

It is instructive to see what happens when an `e1Grp` object is created that omits the estimated latent distribution. Without an explicit latent distribution, the standard Normal is assumed. Examine the change in EAP scores with and without the estimated latent distribution (Figure 4.6).

4.3.2 Two-Tier Latent Covariance

Suppose a music researcher published item parameters for a measure of music perception accuracy. Items load to some extent on a tonal factor and a rhythm factor. In addition, there are 3 items associated with each stimulus. Since questions asking about the same stimulus are expected to be more correlated than items about different stimulus, these groups of items share extra variance. You have administered this measure to a few classes of music students and wish to know about the distribution of their tonal and rhythmic perception accuracy.

```
# replace with published item parameters
spec <- list()
spec[1:21] <- rpf.grm(factors=9, outcomes=5)
factors <- c('tonal', 'rhythm', paste('s', 1:7, sep=""))
imat <- mxMatrix(name="item", values=simplify2array(lapply(spec, rpf.rparam)),
                 dimnames=list(c(factors, paste('b', 1:4, sep="")),
```

```
                              paste("i", 1:length(spec), sep="")))

# arrange the per-stimulus covariance structure
for (stimulus in 1:7) {
  imat$values[2 + stimulus, -(((stimulus - 1) * 3 + 1) : (stimulus*3))] <- 0
}

# replace with your own data
require(MASS)  # for mvrnorm
simCov <- matrix(0, length(factors), length(factors))
diag(simCov) <- rlnorm(length(factors), .5, .5)
simCov[1:2, 1:2] <- c(1.2, .4, .4, .8)
skill <- mvrnorm(200, c(1.1, .7, runif(7)), simCov)
data <- rpf.sample(t(skill), spec, imat$values)

# set up the matrices to hold our latent free parameters
mMat <- mxMatrix(name="mean", nrow=length(factors), ncol=1, values=0, free=TRUE)
rownames(mMat) <- factors
covMat <- mxMatrix(name="cov", values=diag(length(factors)), free=FALSE)
covMat$labels[1,2] <- covMat$labels[2,1] <- 'cov1'  # ensure symmetric
dimnames(covMat) <- list(factors, factors)
covMat$free[1:2, 1:2] <- TRUE
diag(covMat$free) <- TRUE

trModel <- mxModel(model="tr", mMat, covMat, imat,
             mxExpectationBA81(spec, qpoints=15, qwidth=5),
             mxFitFunctionML(),
             mxData(observed=data, type="raw"))

latentModel <- mxModel(model="latent",
                   mxDataDynamic(type="cov", expectation="tr.expectation"),
                   mxExpectationNormal(covariance="tr.cov", means="tr.mean"),
                   mxFitFunctionML())

m1Model <- mxModel(model="music1", trModel, latentModel,
                 mxFitFunctionMultigroup(c("tr.fitfunction", "latent.fitfunction")),
                 mxComputeEM('tr.expectation', 'scores',
                         mxComputeGradientDescent(fitfunction="latent.fitfunction"),
                         tolerance=1))

m1Model <- mxRun(m1Model)
```

There are 9 factors which usually would entail 9 dimensional integration over the latent density. Such high dimensional integration is either intractable or takes a long time to run. However, this model happens to have a two-tier covariance structure that permits analytic reduction to 3 dimensional integration. Formally, a two-tier covariance matrix is restricted to

$$\Sigma_{\text{two-tier}} = \begin{pmatrix} G & 0 \\ 0 & \text{diag}(\tau) \end{pmatrix},$$

where the covariance sub-matrix G is unrestricted (subject to identification), covariance sub-matrix $\text{diag}(\tau)$ is diagonal, and τ is a vector of variances. The factors that make up G are called primary factors and the factors that comprise τ are called specific factors. Furthermore, each item is permitted to load on at most one specific factor.

```
mxExpectationBA81(spec, qpoints=15, qwidth=5)
```

The default quadrature uses 49 points per dimension. That works out to $49^3 = 117649$ points for 3 dimensions. To speed things up at the cost of some accuracy, we reduced the equal-interval quadrature to 15 points, ranging from Z score -5 to 5. This reduces the number of quadrature points to $15^3 = 3375$.

```
mxComputeEM('tr.expectation', 'scores',
            mxComputeGradientDescent(fitfunction="latent.fitfunction"),
            tolerance=1)
```

You may have noticed that latent parameter models have used `mxComputeGradientDescent` instead of `mxComputeNewtonRaphson`. That is because the analytic derivatives for the multivariate normal required by the Newton-Raphson optimizer had not been coded into `OpenMx` at the time of writing. At least for item parameters, the availability of two optimizers offers a way to verify that the optimization algorithm is working. You can always replace `mxComputeNewtonRaphson` by `mxComputeGradientDescent`. The optimization will take more time, but you can check whether you arrive at the same optimum. The ability to easily swap-out and replace components of a model is invaluable for debugging unexpected behavior. Finally, the option `tolerance=1` is there to terminate optimization early. This is meant to be a quick demonstration and requesting higher accuracy slows down estimation substantially.

- multiple group models
- special features to cope with missing data
- automatic-ish item construction (especially for the nominal model)
- simulation studies

4.4 Future Extensions

- In addition to `mxExpectationNormal`, it should be possible to fit the latent distribution to an arbitrary structural model created using `RAM` or `LISREL` notation. Currently, the dimensionality of the latent space is limited by the use of quadrature integration. However, the Metropolis-Hastings Robbins-Monro (MH-RM) algorithm can efficiently fit high-dimensional models [Cai2010]. The MH-RM algorithm would make a useful addition to `OpenMx`.
- The `item` matrix could be provided as an arbitrary algebra. This would be a generalization of the linear latent trait model.

CHAPTER

FIVE

ADVANCED CONCEPTS

5.1 OpenMx Internal Architecture

OpenMx is a language plugin designed to perform linear algebra. OpenMx is too limited to be seen as a general purpose programming language, but it is also more expressive than a typical C library. OpenMx was developed as an R package. It borrows R's syntax parser to encode algebra expressions and the internals are rich with the R procedures needed to smoothly integrate with R. However, it is not inconceivable that OpenMx could eventually be split into a pure C backend with an R or Python or Julia front end.

The fundamental data structure in OpenMx is the 2 dimensional matrix of double precision real numbers. This is inflexible. Data of dimensionality greater than 2 is cumbersome to model. However, the benefit of this inflexibility is speed. Internal operators can make assumptions and go faster because matrices all have the same structure.

Algebras are reified functions of matrices. Information flows through algebras, in one direction, during model fitting. Algebras are not compiled to assembly. Algebra use a simple byte code language to represent algebraic operators. Dependencies between algebras are recorded such that recalculation is kept to a minimum. What transformations cannot easily be expressed in algebras can be put into algebra-like objects such as MxExpectation or MxFitFunction.

5.1.1 MxModel Lifecycle

MxModels are created in R, consisting of R data structures. An MxModel can only have 1 MxData and 1 MxExpectation. This implies that multigroup and multilevel models require more than 1 MxModel. MxRun translates all the information contained in an R MxModel into corresponding C data structures. At this stage, as much as possible is checked to make sure that the model is correctly specified to avoid errors during optimization.

- MxMatrices are torn apart. Free variable labels are matched to create a list of free variables recording locations where they are stored in which matrices. Starting values and upper and lower bounds are collected for each free variable.
- The model tree is flattened. The hierarchy is preserved in MxBaseExpectation container and submodel slots, but otherwise, the whole model tree is flattened into lists of matrices, algebras, expectations, and fitfunctions. This treatment is applied separately to MxModels marked as independent.
- MxCompute controls what happens in the backend.
- After the backend returns, the model is updated with everything that changed.
- The model@runstate slot is suppose to preserve the post-backend state of the model so that summary output doesn't change after the user changes something in the model after mxRun.

Note that MxEval is almost pure R code.

5.1.2 Parallelization Facilities

There 2 facilities for OpenMx to take advantage of parallelism. The first facility is OpenMP. At the time of writing, only the FitFunctions FIMLFitFunction and RowFitFunction use OpenMP. To allow threads to work without interfering with each other, both of these MxFitFunctions request that the complete model (C data structures) be copied such that each thread has its own copy. This model copying is optional. With adequate care, it is also possible to write OpenMP optimizations without the added complexity of duplicating the model (see sadmvn.f). The second facility is Snowfall. Snowfall works at the granularity of machines or processes with a machine while OpenMP is only concerned with exploiting all the CPU within a single machine.

5.2 File Checkpointing

This section will cover how to periodically save the state of the optimizer to a file on disk. In the event of a system crash, the checkpoint file can be loaded back into the model when the system has been restored. The checkpoint file can also be used as a log to trace the state of the free parameters as they change during optimization. The simplest form of file checkpointing is to use the argument `checkpoint = TRUE` in the call to the `mxRun()` function.

```
require(OpenMx)

data(demoOneFactor)
manifestVars <- names(demoOneFactor)

factorModel <- mxModel("One Factor",
    mxMatrix(type="Full", nrow=5, ncol=1, values=0.2, free=TRUE, name="A",
        labels=letters[1:5]),
    mxMatrix(type="Symm", nrow=1, ncol=1, values=1, free=FALSE, name="L"),
    mxMatrix(type="Diag", nrow=5, ncol=5, values=1, free=TRUE, name="U"),
    mxAlgebra(expression=A %*% L %*% t(A) + U, name="R"),
    mxExpectationNormal(covariance="R", dimnames=manifestVars),
    mxFitFunctionML(),
    mxData(observed=cov(demoOneFactor), type="cov", numObs=500)
)

factorFit <- mxRun(factorModel, checkpoint = TRUE)
```

With no extra options, a checkpoint file will be created in the current working directory with the filename: "<modelname>.omx". The checkpoint file is a data.frame object such that each row contains all the values of the free parameters at a particular instance in time. By default, a row is added to the file every 10 minutes. The `mxOption()` function can be used to set the directory of the checkpoint file, to specify an optional prefix to the checkpoint filename, to select whether to save based on minutes or number of optimizer iterations, and to specify the checkpoint interval in units of minutes or optimizer iterations. Below is an example that modifies some of the checkpoint options:

```
directory <- tempdir()

factorModel <- mxOption(factorModel, "Checkpoint Directory", directory)
factorModel <- mxOption(factorModel, "Checkpoint Units", "iterations")
factorModel <- mxOption(factorModel, "Checkpoint Count", 10)
```

After a checkpoint file has been created, it can be loaded into a MxModel object using the `mxRestore()` function. It is necessary to specify the checkpoint directory and checkpoint filename prefix if they were declared using mxOption() when the checkpoint file was created:

```
factorFit <- mxRun(factorModel, checkpoint = TRUE)
factorRestore <- mxRestore(factorModel, chkpt.directory = directory)
```

The checkpoints will extend to independent submodels in a collection of models. Each independent submodel will be saved in a separate file. See `?mxOption` or `getOption('mxOptions')` for a list of options that modify the behavior of file checkpointing. See `?mxRestore` for more information on restoring a checkpoint file.

5.3 Multicore Execution

This section will cover how take advantage of multiple cores on your machine. To use the multicore mode in OpenMx, you must declare independent submodels. A model is declared independent by using the argument 'independent=TRUE' in the `mxModel()` function. An independent model and all of its dependent children are executed in a separate optimization environment. An independent model shares **no** free parameters with either its sibling models or its parent model. An independent model may **not** refer to matrices or algebras in either its sibling models or its parent model. A parent model may access the final results of optimization from an independent child model.

To use the snowfall library, you must start your R environment with the following commands:

```
library(OpenMx)
library(snowfall)
sfInit(parallel = TRUE, cpus = 8)
sfLibrary(OpenMx)
```

`sfInit` will initialize the snowfall cluster. You must specify either the number of CPUs on your machine or the cluster environment (see snowfall package documentation). `sfLibrary` exports the OpenMx library to the client nodes in the cluster. At the end of your script, use the command:

```
sfStop()
```

5.3.1 To Improve Performance

Any sequential portions of your script will quickly become the performance bottleneck (Amdahl's Law). Avoid iteration over large data structures. Use the functions `omxLapply()` and `omxSapply()` instead of iteration. These two functions invoke the snowfall `sfLapply()` and `sfSapply()` functions if the snowfall library has been loaded. Otherwise they invoke the sequential functions `lapply()` and `sapply()`. To hunt for bottelnecks in your script, run your script with multicore settings enabled and use Rprof to profile a reasonable size test case. Ignore the calls to sfLapply() and sfSapply() in the results of profiling. Any other time-consuming calls represent potential sequential bottlenecks.

Some of the functions provided by the OpenMx library can be bottlenecks. Iterative use of the `mxModel()` function in order to add submodels can be time consuming. Use the following unsafe idiom to improve performance:

```
topModel <- mxModel('container')
# generate a list of independent submodels
submodels <-  omxLapply(1:100, generateNewSubmodels)
names(submodels) <- imxExtractNames(submodels)
topModel@submodels <- submodels
```

5.3.2 An Example

The following script can be found with `demo(BootstrapParallel)`

```
# parameters for the simulation: lambda = factor loadings,
# specifics = specific variances
lambda <- matrix(c(.8, .5, .7, 0), 4, 1)
nObs <- 500
```

```
nReps <- 10
nVar <- nrow(lambda)
specifics <- diag(nVar)
chl <- chol(lambda %*% t(lambda) + specifics)

# indices for parameters and hessian estimate in results
pStrt <- 3
pEnd <- pStrt + 2*nVar - 1
hStrt <- pEnd + 1
hEnd <- hStrt + 2*nVar - 1

# dimension names for OpenMx
dn <- list()
dn[[1]] <- paste("Var", 1:4, sep="")
dn[[2]] <- dn[[1]]

# function to get a covariance matrix
randomCov <- function(nObs, nVar, chl, dn) {
    x <- matrix(rnorm(nObs*nVar), nObs, nVar)
    x <- x %*% chl
    thisCov <- cov(x)
    dimnames(thisCov) <- dn
    return(thisCov)
}

createNewModel <- function(index, prefix, model) {
    modelname <- paste(prefix, index, sep='')
    data <- mxData(randomCov(nObs, nVar, chl, dn), type="cov", numObs=nObs)
    model <- mxModel(model, data)
    model <- mxRename(model, modelname)
    return(model)
}

getStats <- function(model) {
    retval <- c(model@output$status[[1]],
        max(abs(model@output$gradient)),
        model@output$estimate,
        sqrt(diag(solve(model@output$hessian))))
    return(retval)
}


# initialize obsCov for MxModel
obsCov <- randomCov(nObs, nVar, chl, dn)

# results matrix: get results for each simulation
results <- matrix(0, nReps, hEnd)
dnr <- c("inform", "maxAbsG", paste("lambda", 1:nVar, sep=""),
    paste("specifics", 1:nVar, sep=""),
    paste("hessLambda", 1:nVar, sep=""),
    paste("hessSpecifics", 1:nVar, sep=""))
dimnames(results)[[2]] <- dnr

# instantiate MxModel
template <- mxModel("stErrSim",
                    mxMatrix(name="lambda", type="Full", nrow=4, ncol=1,
                             free=TRUE, values=c(.8, .5, .7, 0)),
                    mxMatrix(name="specifics", type="Diag", nrow=4,
```

```
                    free=TRUE, values=rep(1, 4)),
                mxAlgebra(lambda %*% t(lambda) + specifics,
                    name="preCov", dimnames=dn),
                mxData(observed=obsCov, type="cov", numObs=nObs),
                mxExpectationNormal(covariance='preCov'),
                mxFitFunctionML(),
                independent = TRUE)

topModel <- mxModel("container")

submodels <- lapply(1:nReps, createNewModel, "stErrSim", template)

names(submodels) <- imxExtractNames(submodels)
topModel@submodels <- submodels

modelResults <- mxRun(topModel, silent=TRUE, suppressWarnings=TRUE)

results <- t(omxSapply(modelResults@submodels, getStats))

# get rid of bad covergence results
results2 <- data.frame(results[which(results[,1] <= 1),])

# summarize the results
means <- mean(results2)
stdevs <- sd(results2)
sumResults <- data.frame(matrix(dnr[pStrt:pEnd], 2*nVar, 1,
                        dimnames=list(NULL, "Parameter")))
sumResults$mean <- means[pStrt:pEnd]
sumResults$obsStDev <- stdevs[pStrt:pEnd]
sumResults$meanHessEst <- means[hStrt:hEnd]
sumResults$sqrt2meanHessEst <- sqrt(2) * sumResults$meanHessEst

# print results
print(sumResults)
```

5.4 Full Information Maximum Likelihood, Row Fit Specification

This example will show how full information maximum likelihood (FIML) can be implemented using a row-by-row evaluation of a likelihood function. **Note: You do not have to implement your own FIML! The method of FIML used in this example is for didactic purposes only.** If you are looking for how to get your model to use FIML, then look for any model that uses raw data. For example

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/OneFactorModel_PathRaw.R

This document seeks to inform users about how they can implement their own row fit functions using FIML as an example. The example is in two parts. The first part is a discussion of full information maximum likelihood. The second part is an example implementation of full information maximum likelihood in a row-wise fit function that estimates the saturated model in two variables. The second part refers to the following file:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/RowObjectiveFIMLBivariateSaturated.R

There is an analogous version of this example that uses the standard full information maximum likelihood implementation here:

- http://openmx.psyc.virginia.edu/docs/OpenMx/2.6.7/_static/demo/BivariateCorrelation.R

The goal of the current document is twofold: to increase users' understanding of full information maximum likelihood, and to assist users in implementing their own row fit functions.

5.4.1 Full Information Maximum Likelihood

Full information maximum likelihood is almost universally abbreviated FIML, and it is often pronounced like "fimmle" if "fimmle" was an English word. FIML is often the ideal tool to use when your data contains missing values because FIML uses the raw data as input and hence can use all the available information in the data. This is opposed to other methods which use the observed covariance matrix which necessarily contains less information than the raw data. An observed covariance matrix contains less information than the raw data because one data set will always produce the same observed covariance matrix, but one covariance matrix could be generated by many different raw data sets. Mathematically, the mapping from a data set to a covariance matrix is not one-to-one (i.e. the function is non-injective), but rather many-to-one.

Although there is a loss of information between a raw data set and an observed covariance matrix, in structural equation modeling we are often only modeling the observed covariance matrix and the observed means. We want to adjust the model parameters to make the observed covariance and means matrices as close as possible to the model-implied covariance and means matrices. Therefore, we are usually not concerned with the loss of information from raw data to observed covariance matrix. However, when some raw data is missing, the standard maximum likelihood method for determining how close the observed covariance and means matrices are to the model-expected covariance and means matrices fails to use all of the information available in the raw data. This failure of maximum likelihood (ML) estimation, as opposed to FIML, is due to ML exploiting for the sake of computational efficiency some mathematical properties of matrices that do not hold true in the presence of missing data. The ML estimates are not wrong per se and will converge to the FIML estimates, rather the ML estimates do not use all the information available in the raw data to fit the model.

The intelligent handling of missing data is a primary reason to use FIML over other estimation techniques. The method by which FIML handles missing data involves filtering out missing values when they are present, and using only the data that are not missing in a given row.

Likelihood of a Row of Data

If *X* is the entire data set then the minus two log likelihood of row *i* of the data is

$$\mathcal{L}_i = k_i \ln(2\pi) + \ln(|\Sigma_i|)) + (X_i - M_i)\Sigma_i^{-1}(X_i - M_i)^\mathsf{T} \tag{5.1}$$

where

- k_i is the number of non-missing observed variables in row *i* of the data set
- $\ln()$ is the natural logarithm function (i.e. logarithm base $e = 2.718...$)
- $\pi = 3.14159...$
- $|*|$ is the determinant of $*$
- Σ_i is the *filtered* model-implied manifest covariance matrix ($k_i \times k_i$ matrix)
- X_i is the *filtered* row *i* of the data set ($1 \times k_i$ matrix)
- M_i is the *filtered* model-implied manifest means row vector ($1 \times k_i$ matrix)
- Σ_i^{-1} is the inverse of Σ_i
- $(*)^\mathsf{T}$ is the transpose of $*$

Equation (5.1) is identically equal to -2 times the logarithm of the probability density function of the multivariate normal distribution. The minus two log likelihood is quite literally minus two times the log of the probability of the data given the model.

There are several important things to note about Equation (5.1).

First, the model-implied means vector and the model-implied covariance matrix are for the the manifest variables only. Although your structural equation model may involve both latent and manifest variables, the latent variables are only present to explain the means and covariances of the observed variables in a meaningful and parsimonious way. The free parameters of your model are adjusted to make the model-implied means vector and the model-implied covariance matrix as close as possible to the observed means vector and the observed covariance matrix.

Filtering

Second, there are several references to filtered vectors and matrices. Filtering is how FIML handles missing data. All filtering is performed based on the pattern of missingness observed in a given row of data. If the observed values of a given row, i, of data are $x_i = (1, 14, NA, 2, NA, NA)$ where NA denotes a missing value, then the filtered data row is $X_i = (1, 14, 2)$. The filtered data row is merely the original data row with the missing entries removed. For this row, entries 3, 5, and 6 are removed. Alternatively, entries 1, 2, and 4 are kept.

The filtered, model-implied means row vector is similar. If the original model-implied means row vector is $M = (2, 15, 0, 3, 1, 2)$, then the filtered model-implied means row vector for row i is $M_i = (2, 15, 3)$, keeping only entries 1, 2, and 4.

The filtered, model-implied covariance matrix is marginally more complicated. It must be selected on both rows and columns. If Σ is the model-implied covariance matrix and Σ is given by $\Sigma = \begin{pmatrix} 1 & 3 & 1 & 2 & 1 & 2 \\ 3 & 13 & 3 & 6 & 3 & 6 \\ 1 & 3 & 2 & 2 & 3 & 2 \\ 2 & 6 & 2 & 8 & 2 & 4 \\ 1 & 3 & 3 & 2 & 14 & 2 \\ 2 & 6 & 2 & 4 & 2 & 5 \end{pmatrix}$ then the filtered covariance matrix selects rows 1, 2, and 4 $\Sigma = \begin{pmatrix} \mathbf{1} & \mathbf{3} & \mathbf{1} & \mathbf{2} & \mathbf{1} & \mathbf{2} \\ \mathbf{3} & \mathbf{13} & \mathbf{3} & \mathbf{6} & \mathbf{3} & \mathbf{6} \\ 1 & 3 & 2 & 2 & 3 & 2 \\ \mathbf{2} & \mathbf{6} & \mathbf{2} & \mathbf{8} & \mathbf{2} & \mathbf{4} \\ 1 & 3 & 3 & 2 & 14 & 2 \\ 2 & 6 & 2 & 4 & 2 & 5 \end{pmatrix}$ and columns 1, 2, and 4. $\Sigma = \begin{pmatrix} \mathbf{1} & \mathbf{3} & 1 & \mathbf{2} & 1 & 2 \\ \mathbf{3} & \mathbf{13} & 3 & \mathbf{6} & 3 & 6 \\ \mathbf{1} & \mathbf{3} & 2 & \mathbf{2} & 3 & 2 \\ \mathbf{2} & \mathbf{6} & 2 & \mathbf{8} & 2 & 4 \\ \mathbf{1} & \mathbf{3} & 3 & \mathbf{2} & 14 & 2 \\ \mathbf{2} & \mathbf{6} & 2 & \mathbf{4} & 2 & 5 \end{pmatrix}$ The selection on both rows and columns yields the following filtered expected covariance matrix. $\Sigma_i = \begin{pmatrix} 1 & 3 & 2 \\ 3 & 13 & 6 \\ 2 & 6 & 8 \end{pmatrix}$ In practical implementations of FIML, the data are first sorted based on their pattern of missingness, so that all the rows missing on variables 3, 5, and 6 are computed together followed by all the rows with a different missingness pattern. This sorting allows fewer filterings to be performed and often accelerates the likelihood computation. In the row fit implementation shown below there is no data sorting because it is for demonstration purposes only. The implementation of FIML in the backend of OpenMx uses this data sorting and other techniques to provide speed ups. The details are in the source code at http://openmx.psyc.virginia.edu/svn/trunk/R/MxFIMLObjective.R and http://openmx.psyc.virginia.edu/svn/trunk/src/omxFIMLFitFunction.cpp .

Quadratic Products

There is one final note to discuss about Equation (5.1). A very important component to Equation (5.1) is $(X_i - M_i)\Sigma_i^{-1}(X_i - M_i)^\mathsf{T}$. It is a quadratic form. Any expression of the form xAx^T where x is a row-vector and A is a

matrix is called a *quadratic form*. Equivalently, a quadratic form can be stated as $x^\mathsf{T} Ax$ where x is a column-vector. In mathematical circles it is typical to express quadratic forms in terms of columns vectors as $x^\mathsf{T} Ax$, whereas in statistical circles it is common to express quadratic forms in terms of row vectors as xAx^T. The difference is completely arbitrary and due to tradition and convenience. Quadratic forms arise in many disciplines: in engineering as linear quadratic regulators, in physics as potential and kinetic energy, and in economics as utility functions. Quadratic form appear in many optimization problems, so it is no surprise that they appear in the FIML equation.

A quadratic form xAx^T can also be thought of as a quadratic product of x and A, so that $x \bigotimes A = xAx^\mathsf{T}$.

The particular quadratic form in Equation (5.1) has a special meaning and interpretation. It is the squared Mahalanobis distance from the data row X_i to the mean vector M_i in the multivariate space defined by the covariance matrix Σ_i. Intuitively, the Mahalanobis distance from the mean vector tells you how far an observation is from the center of the distribution, taking into account the spread of the distribution in all directions.

For well-behaved covariance matrices, the value of the quadratic form in equation (5.1) (i.e. the squared Mahalanobis distance) is always greater than or equal to zero, and equal to zero only when the observation row vector is exactly equal to the mean vector. The likelihood functions in maximum likelihood (ML) and in FIML are not defined when this is not the case. In general for any row-vector x and square, symmetric matrix A, if $xAx^\mathsf{T} > 0$ for any $x \neq 0$, then the quadratic form xAx^T and the matrix A are called *positive definite*.

Because ML and FIML are not defined when the model-implied covariance matrix is not positive definite, frequent and often cryptic error message that users of any structural equation modeling program receive is something like ERROR: EXPECTED COVARIANCE MATRIX IS NOT POSITIVE DEFINITE. A number of different problems could induce this error. The model may be unidentified; a variable may have zero variance, i.e. be a constant; one variable might be a linear combination of another variable or equal to another variable; the starting values might imply an impossible covariance matrix; a variable may have zero or negative error (i.e. residual) variance. In any case, it is a good idea to check your model specification for theoretical and typographical errors, and if you are expecting a parameter like an error variance to be greater than zero then set zero as that parameter's lower bound.

Now that the FIML equation for a single row of data has been discussed, it is relevant to see how the full information maximum likelihood of the entire data set is computed.

Likelihood of the Entire Data

The minus two log likelihood of the entire data set is the sum of the minus two log likelihoods of the rows.

$$\mathcal{L} = \sum_{i=1}^{N} \mathcal{L}_i$$

where there are N rows in the data.

5.4.2 Row Fit Example

We will now implement FIML using a row-wise fit function. The `mxFitFunctionRow()` function evaluates an `mxAlgebra` for each row of a data set. It then stores the results of this row-wise evaluation in an `mxAlgebra` which is by default called "rowResults". Finally, the row results must be collapsed into a single number. Another `mxAlgebra` called the "reduceAlgebra" takes the row results and reduces them to a single number which is then minimized.

Data

For this example we will simulate our own data. We will use the `mvrnorm()` function which lives is the `MASS` package. The `mvrnorm()` function generates a multivariate random normal sample with a given vector of means and a given covariance matrix. The following code generates the data.

```
require(MASS)
set.seed(200)
rs <- .5
xy <- mvrnorm (1000, c(0,0), matrix(c(1, rs, rs, 1), nrow=2, ncol=2))
```

The data have 2 variables with 1000 rows. The true means are 0. Each variable has a true variance of 1.0, and a covariance of 0.5.

Some further data processing will prove helpful. First, we recast the generated data as a `data.frame` object in R. Second, we tell R that what we want the variables names to be. Finally, we look at a summary of the data set and the observed covariance matrix which differs slightly from the covariance matrix used to generate the data.

```
testData <- as.data.frame(xy)
testVars <- c('X','Y')
names(testData) <- testVars
summary(testData)
cov(testData)
```

Now the data has been generated and we can specify the saturated model.

Model Specification

We generate an `mxModel`, give it data, and two `mxMatrix` objects. The first `mxMatrix` is a row-vector or completely free parameters and is the model-implied means vector. Because we are specifying the saturated model, the means are freely estimated. The second `mxMatrix` gives the model-implied covariance matrix. Because we are specifying the saturated model, the covariance matrix is freely estimated, however it is still constrained to by symmetric and the starting values are picked so that the variances on the diagonal are in general larger than the covariances.

```
bivCorModelSpec <- mxModel(
    name="FIML Saturated Bivariate",
    mxData(
        observed=testData,
        type="raw",
    ),
    mxMatrix(
        type="Full",
        nrow=1,
        ncol=2,
        free=TRUE,
        values=c(0,0),
        name="expMean"
    ),
    mxMatrix(
        type="Symm",
        nrow=2,
        ncol=2,
        values=c(.21, .2, .2, .21),
        free=TRUE,
        name='expCov'
    )
)
```

Filtering

We create a new `mxModel` that has everything from the previous model. We then create `mxAlgebra` objects that filter the expected means vector and the expected covariance matrix. We also create an `mxAlgebra` that keeps track

of the number of variables that are not missing in a given row.

```
bivCorFiltering <- mxModel(
    model=bivCorModelSpec,
    mxAlgebra(
        expression=omxSelectRowsAndCols(expCov, existenceVector),
        name="filteredExpCov",
    ),
    mxAlgebra(
        expression=omxSelectCols(expMean, existenceVector),
        name="filteredExpMean",
    ),
    mxAlgebra(
        expression=sum(existenceVector),
        name="numVar_i")
)
```

Calculations

We create a new `mxModel` that has everything from the previous models.

```
bivCorCalc <- mxModel(
    model=bivCorFiltering,
    mxAlgebra(
        expression = log(2*pi),
        name = "log2pi"
    ),
    mxAlgebra(
        expression=log2pi %*% numVar_i + log(det(filteredExpCov)),
        name ="firstHalfCalc",
    ),
    mxAlgebra(
        expression=(filteredDataRow - filteredExpMean) %&% solve(filteredExpCov),
        name = "secondHalfCalc",
    )
)
```

Row Fit Specification

We create a new `mxModel` that has everything from the previous models.

```
bivCorRowObj <- mxModel(
    model=bivCorCalc,
    mxAlgebra(
        expression=(firstHalfCalc + secondHalfCalc),
        name="rowAlgebra",
    ),
    mxAlgebra(
        expression=sum(rowResults),
        name = "reduceAlgebra",
    ),
    mxFitFunctionRow(
        rowAlgebra='rowAlgebra',
        reduceAlgebra='reduceAlgebra',
        dimnames=c('X','Y'),
    )
)
```

```
bivCorTotal <- bivCorRowObj
```

Model Fitting

```
bivCorFit <- mxRun(bivCorTotal)
```

5.5 CSOLNP Documentation

CSOLNP is an open-source optimizer engine for the OpenMx package. It is a C++ translation of solnp function from the Rsolnp package, available on CRAN. The algorithm solves nonlinear programming problems in general form of:

$$\begin{gathered}\min f(x)\\ \text{subject to:}\\ g(x) = 0\\ l_h \leq h(x) \leq u_h\\ l_x \leq x \leq u_x\end{gathered}$$

Where:
x : Vector of decision variables ($x \in R^n$).
$f(x)$: Objective function ($f(x) : R^n \rightarrow R$).
$g(x)$: Equality constraint function ($g(x) : R^n \rightarrow R^{m_e}$).
$h(x)$: Inequality constraint function ($h(x) : R^n \rightarrow R^{m_i}$).
l_h, u_h: Lower and upper bounds for Inequality constraints.
l_x, u_x: Lower and upper bounds for decision variables.

$f(x)$, $g(x)$ and $h(x)$ are all smooth functions.

Each major iteration of the optimization algorithm solves an augmented Lagrange multiplier method in the form of:

$$\begin{gathered}\min f(x) - y^k g(x) + (\tfrac{\rho}{2})\|g(x)\|^2\\ \text{subject to:}\\ J^k(x - x^k) = -g(x^k)\\ l_x \leq x \leq u_x\end{gathered}$$

Where ρ is the penalty parameter. $g(x)$ denotes all the constraints including equality constraints and inequality constraints, which are converted to equalities by adding slack variables. J^k is the Jacobian of first derivatives of $g(x)$, and y^k is the vector of Lagrange multipliers. The superscript k denotes the k^{th} major iteration.

Each major iteration starts with a feasibility check of the decision variables x^k, and continues by implementing a Sequential Quadratic Programming (SQP) method, which calculates the gradient and the Hessian for the augmented Lagrange multiplier method. The criteria for moving to the next major iteration is satisfied when a Quadratic Pro-

gramming problem of the form:

$$\min (\tfrac{1}{2})(x - x^k)^T H(x - x^k) + g^T(x - x^k)$$
$$\text{subject to:}$$
$$J^k(x - x^k) = -g(x^k)$$
$$l_x \leq x \leq u_x$$

results in a feasible and optimal solution to the augmented Lagrange multiplier problem. If the solution is not feasible or optimal, a new QP problem is called (minor iteration) for updating the gradient and the Hessian. The stop criterion for the optimization is either when the optimal solution is found, or when the maximum number of iterations is reached.

5.5.1 Input

Before providing an explanation of each input for CSOLNP, it is necessary to look at the "Matrix" structure defined for the use of any vector or matrix needed throughout CSOLNP.
"Matrix" is a structure containing three fields being:
int rows: Number of rows of the matrix.
int cols: Number of columns of the matrix.
double *t: Pointer to an array of doubles storing the matrix elements.

The arguments are:
- solPars: A Matrix of starting values for decision variables.
- solFun: Pointer to the objective function which takes the decision variable Matrix as input and returns the objective value.
- solEqB: A Matrix of equality constraints.
- solEqFun: Pointer to the equality constraint function which takes the decision variable Matrix as its argument, and returns a Matrix of evaluated constraints.
- solIneqFun: Pointer to the inequality constraint function with the Matrix of decision variables as input, and a Matrix of evaluated inequality constraints as output.
- solIneqLB: Matrix of lower bounds for inequality constraints.
- solIneqUB: Matrix of upper bounds for inequality constraints.
- solLB: Matrix of lower bounds for decision variables.
- solUB: Matrix of upper bounds for decision variables.
- solctrl: Matrix of control parameters containing:
  - ρ: The penalty parameter in the augmented objective function with the default value of 1.
  - maxit: Number of major iterations with the default value of 400.
  - minit: Number of minor iterations with the default value of 800.
  - δ: Step size in numerical gradient calculation with the default value of 1e-7.
  - tol: Relative tolerance on feasibility and tolerance with the default value of 1e-8.
- verbose: An integer variable with 3 levels (1, 2, 3) for printing throughout CSOLNP. verbose = 3 prints every calculation within CSOLNP.

5.5.2 Output

A structure containing the following values:
- Final objective value.
- Optimal estimations of decision variables.

- Hessian at the optimal solution.
- Gradient at the optimal solution.
- A variable named inform reporting the result of the optimization (same as inform variable returned by NPSOL optimizer). The following scenarios are reported by different values of inform:

inform = 0: Optimal solution is found.
inform = 1: The optimal solution is found but not to the requested accuracy.
inform = 4: Maximum number of major iterations is reached.
inform = 6: No improvement can be made to the current point (no convergence).

5.5.3 Example

To be provided

5.5.4 Restoring NPSOL

To use NPSOL, your OpenMx must be compiled to include it. If NPSOL is available, you can make it the default optimizer with

mxOption(NULL, "Default optimizer", "NPSOL")

You can also control this setting with the IMX_OPT_ENGINE environment variable.

5.5.5 Comparing the performances of CSOLNP, and NPSOL

Running the test suite of the package with both optimizers resulted in the following average running times:

NPSOL
real 3m58.2688s
user 3m56.4598s
sys 0m1.5264s

CSOLNP
real 4m13.9032s
user 4m11.8268s
sys 0m1.7012s

CHAPTER

SIX

CHANGES IN OPENMX

6.1 trunk

- Prevent Varadhan2008 from failing near convergence
- Throw error on attempt to invert incomplete Hessian
- CSOLNP: Correct exclusion of inequality constraints from gradient/Hessian
- When checkpointing fit, record who requested it
- Restore parallel processing for CIs
- Add "Checkpoint Fullpath" to override output destination
- Make mxOption(model, val) return the global setting or the model's override
- Add WLS standard error and chi-square functions
- Consolidate starting value nudging logic in ComputeGD
- Try harder to keep CIs ordered properly vs estimate
- Add partial identification information to ID check. Tells which parameters are not identified.
- CSOLNP: Avoid reuse of obm for more than 1 purpose
- CSOLNP: Simplify CI calculation
- CSOLNP: Tidy optimizer reporting
- CSOLNP: Count the number of major iterations
- Add Makefile rule to run the failing tests
- CSOLNP: Return stuff via CSOLNPFit; isolate Matrix inside of subnp
- CSOLNP: Remove some redundent information
- Remove template argument from CSOLNP::solnp
- Store equality & inequality results in CSOLNPFit
- Reduce variable lifetime
- Tidy some minor valgrind issues
- Remove q() from test
- CSOLNP: Convert control param to Eigen vector
- CSOLNP: Move bounds to CSOLNPFit; remove lots of deadcode
- CSOLNP: Move inequality bounds to CSOLNPFit

- CSOLNP: Remove Matrix arg from CSOLNPFit::solFun
- CSOLNP: Remove **GLOB_** prefix since variables are no longer global
- CSOLNP: Eliminate more globals
- CSOLNP: Continue refactoring API
- CSOLNP: Pass parameter vector using Eigen
- CSOLNP: Rewrite omxProcessConstraintsCsolnp in Eigen
- CSOLNP: Don't bother creating the unused solEqB matrix
- CSOLNP: Eliminate reliance on all-zero solEqB
- Simplify setup of constraints
- CSOLNP: Put in temporary fix for conformability of derivs with constraints
- CSOLNP: Fix some problems revealed by Eigen conformability checking
- Rework calculation of compiler args; add IMX_SAFE switch
- Add test for omxGetParameters
- omxGetParameters: Omit duplicated parameters even when there are no submodels
- Tidy up MxMatrix-related error and warning messages; re-write MatrixErrorDetection.R
- BA81: Prohibit all NA rows
- Small but important MxMatrix-related adjustments.
- Add Jacobian-based model identification check.
- Added method for getting cov, means, and thresh from LISREL expectation.
- Added method for getting cov, means, and thresh from RAM expectation.
- Added generic for getting expected covariance, means, and thresholds.
- Small test for LISREL type, to be expanded
- LISREL model initialization.
- Better MxMatrix verification
- Conditionally "condense" the 'labels', 'free', 'lbound', and 'ubound' slots of MxMatrix objects.
- CSOLNP: Convert ind to Eigen; make easier to understand
- Enable ubsan for gcc 4.9+
- Make gcc happier
- CSOLNP: Remove more deadcode
- Avoid reuse of condif[123]
- test if OpenMx disorders thresholds
- CSOLNP: Remove bounds check; all Eigen ops are already bounds checked
- CSOLNP: Remove superfluous assignment
- CSOLNP: Remove deadcode
- CSOLNP: Converting function add to Eigen
- CSOLNP: Converting functions subtract and timess to Eigen

- CSOLNP: Converting functions subset, copy, multiply, and divide to Eigen
- CSOLNP: reapplying revisions 3976-3996
- Add gcc 4.9 support (same binary as gcc 4.8)
- WLS maybe working for joint.
- Helpful error when SaturatedLikelihood is given a list of two.
- Change the fake definition variable value for easier indexing
- CSOLNP: global variables moved to a struct
- Fix saturated DoF for raw data when only some of the variables are used.
- Bug fix for refs models with cov data
- CSOLNP: reverted version checked in
- Refrain from evaluating MxMatrix in the front-end
- Refactor computation of CFI, TLI, & RMSEA
- Move demos to nightly
- Prevent SEGV on ordinal columns with no thresholds
- Ensure continuous all-NA column is not run through mxFactor
- Add comment about multinomial degrees of freedom
- Enable –force-biarch for Windows fat binaries & remove obsolete rules
- various doc cleanup

6.2 Release 2.0.0-4004 (Oct 24, 2014)

- Fix calculation of the saturated -2LL for IFA models
- Add some warning about CSOLNP
- Add hint about infeasible starting values (per Mike Neale)
- Rework capture of errors & warnings
- Permit running test from top dir
- CSOLNP: Unravel some confused Matrix resizing
- CSOLNP: Allow 0 coeff matrices; init to NaN instead of 1.0
- CSOLNP: Avoid allocation to report gradient
- CSOLNP: Add a varient of subset that copies out to Eigen::MatrixBase
- CSOLNP: Eliminate 2 instances of fillMatrix
- CSOLNP: Shorten lifetime of index variable
- CSOLNP: Convert a little bit of subnp to Eigen
- CSOLNP: For set{Row|Column}InPlace, treat 2nd arg as a vector
- Don't abort build if NPSOL is not found for a particular compiler
- CSOLNP: Switch Bcolj to stack allocation

- CSOLNP: Remove negate deadcode
- CSOLNP: Re-express findMax(matrixAbs()) as matrixMaxAbs()
- CSOLNP: Simplify divideByScalar2D & multiplyByScalar2D
- CSOLNP: Covert a few more copyInto -> copyIntoInplace
- CSOLNP: Move sob to stack allocated storage
- CSOLNP: Move alp to stack allocated storage
- CSOLNP: Move subnp_ctrl from Matrix to stack allocated Eigen::Array
- CSOLNP: Change setColumn to setColumnInplace
- CSOLNP: Change some setRow to setRowInplace
- Once constraints are turned into algebras, eliminate objectives from those new algebras and replace them with fitfunctions.
- CSOLNP: Convert some copyInto into copyIntoInplace
- CSOLNP: options added
- Update citation
- Drop psych dependency
- Document that mxLog will fail in Rgui on Windows
- Stop with error message if mxLog fails
- Updates to documentation
- Report CIs with NA as NA instead of throwing an error
- CSOLNP: void functions and memory allocation checks added to minor iteration loop
- Correctly collapse levels regardless of order (mxFactor)
- Repair derivatives of the beta density
- Avoid exception when only 1 optimizer has errors in the test suite
- Add mxFactor(..., collapse=TRUE)
- Fix bug that failed to print optimizer messages again in summary. Add test.
- Fix mxEval(..., defvar.row) for when data is sorted
- Check for duplicated level names in factors in MxData
- Simple category collapsing for mxFactor
- Change mxStandardizeRAMpaths() to say if the model hasn't been run yet; amend test model accordingly.
- Add mxMakeNames & test
- Add eval by name wrapper for mxEval created by Spiegel
- 'make test' should record errors optimizer-wise
- Don't INSTALLMAKEFLAGS=""; allow setting from the environment
- Report 'make test' errors as they happen
- Throw warning from ref models when definition variables are present.
- Cope with zero length components in mxSimplify2Array

- Make mxFactor preserve rownames
- Update dims slot of expectations for models in mxRun. Adjust saturated model accordingly.
- Make mxSimplify2Array preserve colnames
- Added threshold deviation labels for saturated models
- Fix bug in saturated model for raw matrix data.
- Implement qnorm() and lgamma() in mxAlgebras.
- Make models with row fitfunctions re-runnable from an initial fit and end name collisions with filteredDataRow, existenceVector, and rowResults.
- Modify quoted formula error catch to only use match.call()$expression once.
- Smarter way to catch mxAlgebra formulas that are character strings.
- Catch quoted formula error in mxAlgebra
- Fix mxEval error and add test for square bracket with blank entry
- Modify wall time printing as per Dev mailing list discussion
- Small change so that mxStandardizeRAMpaths() handles independent submodels as intended.
- Adding correctly named Holzinger 1939 data set
- mxFitFunctionMultigroup: Fix error for unmatched submodel; add test
- omxQuotes should do something sensible when passed nothing
- Permit models with no matrices (to avoid masking other errors)
- Coerce x argument of mxFactor to character type
- Fix array indexing of ordinal thresholds (when there are unused columns in observed data)
- Add warning for data.type='cor'
- removing link to non-existant function in OpenMx [doc]
- Avoid stringification of non-finite numbers
- Charlie Driver pointed out that mxTryHard() fit attempts that result in npsolstatus -1 should be treated similarly to fit attempts ending in error
- Fix bug in summary for AIC/BIC with cor data.
- Drop duplicates from CI list
- Prevent acceleration of worsening fit
- BA81: Make E-step optimization less fragile

6.3 Release 2.0.beta3-3838 (Sep 26, 2014)

- Constrain GRM item parameters more sensibly
- Newton-Raphson: Restore feasible parameter vector when fit obtains NaN
- Hopefully, fix performance decrement in FIML.
- Begin EM acceleration 1 cycle earlier
- Switch default EM acceleration to varadhan2008

- Reorg EM acceleration; add Varadhan & Roland (2008)
- Return SEs as "not requested" strings when argument SE=FALSE in mxStandardizeRAMpaths()
- Cope with observed data matrix with no NAs (observed data.frame not affected)
- Mention fomulation of IFA independence model [doc]
- Drop support for non-double matrices
- CSOLNP: memory issue and reporting starting values as optimal parameters
- (Re?)enable parallel processing for confidence intervals
- Grab Eigen from RcppEigen
- Add ordinal ML/WLS test with both model identifications
- For saturated type=cov model, set TLI=1 and RMSEA=0
- Recompute thresholds (when provided)
- Rewrite algebra/fitfunction lookup in MxFitFunctionMultigroup
- Downgrade check of algebra dimnames to a warning for backward compat
- Edit 'make clean' to also remove src/*.dll (Windows shared library).
- Add mxTryHard() and its man page.
- Add check for data type of algebra dimnames
- Implemented smarter tab-completion for MxObjects.
- Make the global freeGroup vector private
- Update SE study for relative tolerance
- SE simulation studies (IFA)
- Depend on package parallel (in R core since 2.14)
- Check threshnames for duplicates
- Store omxThresholdColumn in a std::vector
- LAD–CheckCode6.R moved to models/passing
- CSOLNP: status code report corrected
- Change the 'running xxxmodel' output from mxRun to a message
- Toggle silent= back to FALSE as preference for running Reference/Saturated models.
- Add multilevel model example in state space form. Data is grabbed from the web.
- Report optimizer in summary for default compute plan
- Rename nullModels -> refModels per dev discussion
- Add column of raw SEs to output of mxStandardizeRAMpaths(); edit its test model and man page appropriately.
- Fix 'unknown macro 't" warning I was getting from mxRestore man page
- Document Number of Threads option.
- Modify .svnignore files. WLS branch and trunk changes. Added continuous only wls test. Found and added state space algebra test.
- Include numObs in data for mxNullModels

- First draft of mxDataWLS
- Fixed documentation for reading checkpoint file via read.table
- Permit BA81 mean & variance specified with algebras
- Rename omxSaturatedModel to mxNullModels per dev discussion
- Refuse to clear model slots by assignment to NULL and say why
- Remove minItemsPerScore option
- Fix state space in FIML joint.

6.4 Release 2.0.beta2-3751 (Aug 20, 2014)

- Switch default optimizer back to CSOLNP
- Made MxSummary have prettier printing of Chi-Square and RMSEA with CI. Also added more checks for SummaryCheck.R
- Extinguish globalState
- NLOPT and Simulated annealing added
- Report condition number when standard erorrs are enabled
- Improve destruction order
- Merge OMX_VERBOSE to OMX_DEBUG
- Simplify copyParamToModel API
- Move childList from omxState to FitContext
- Track whether a model has been run for submodels
- Modify FIML Single Iteration Joint to accomodate State Space expectations for continuous vars.
- Distinguish between stale models and models which has not been run
- Improve reporting of CIs for non-free parameters
- Ignore request for CIs if the label exists and is not a free parameter
- Add comment about RMSEA formula
- Reorganize omxSaturatedModel for better modularity
- Factor out processing of the run argument to omxSaturatedModel
- Remove our own version of std::max
- Consolidate FIML to omxFIMLSingleIterationJoint (except for state space). NOTE: This introduced a serious performance regression.
- Remove a bunch of duplicated code in FIML
- Add Oakes1999 EM information matrix method
- Set up setVarGroup handler for MxAlgebraFitFunction
- Add independent flag to mxComputeSequence
- Implement new unprotect strategy
- For summary(verbose=F), suppress SE and bounds if all NA

- minor error msg change: show user the verboten list of names
- Various optimizations to improve frontend performance
- Don't blindly cast all NA columns to double. When the column is a factor, it needs to stay integer
- Only warn once about the model being modified since run
- Prohibit clashes between model and matrix names in type='RAM' models
- Clear modified-since-run flag on submodels
- Permit more than 1 fitfunction in mxComputeOnce
- The backend no longer resizes matrices that will be exported to R
- Switch to more accurate matrix log/exp
- Adjust pretty-printing of MxData to match slot names
- Print model name in summary
- Change error message in saturated model helper when algebra fit function detected.
- Improvements to mxSummary: RMSEA conf intervals, Information table adjustments, guidance to help page, statement about missing fit indices.
- CSOLNP: out of bound starting values issue solved. Mode added
- Revision to saturated model helper that allows multigroup and saturated models
- ComputeHessianQuality should not SEGV when no Hessian is available
- Add fix and test for mxFitFunctionR Hessian
- modified mxVersion to include R and plaform + optimizer
- Add a few unprotects to reduce stack usage of big algebra
- Manual update from Hermine H Maes
- Extensive rework of the conformability checking pass
- Provide more helpful error messages when an expectation fails to initialize
- CSOLNP: potential fix for CI and non-linear constraint issues
- Add checks for AIC, BIC
- Add a conformability checking step to the backend
- better error when definition variable is missing
- Added names() support to the base R object types. Changed the return value of MxModel names(). [For those who enjoy tab-completion]
- Make NPSOL the default if available
- Added verbose=FALSE argument to mxSummary to change amount of information that is printed.
- Add matrix logarithm algebra operator
- Move more CI code from frontend to backend
- Better way to label and reference figures (like APA style)
- Visually decorate figures so they stand out from the rest of the text [manual, HTML version]
- Recompute fit at the end of mxComputeGradientDescent
- CRAN prohibits variable length arrays

- Better error message when a CI is not found. Let user know what to check and do
- Fix for a minor bug that dates back to version 1.3.2 (at least). Now, mxVersion() reports the version number of the OpenMx package loaded into R's workspace, not the version number of the OpenMx package installed in the first directory in .libPaths()
- Added check of many (but not all) of the parts of MxSummary.
- Added joint ordinal continuous example and an example with means to LISREL man page.
- Improve safety of mxFactor
- Automatically extract and run code from the manual to ensure that it works
- Added reporting of Chi square DoF.
- Minor bug fix in saturated model helper.
- Fixed bug in Chi square degrees of freedom calculation.
- modified error msg in mxRun and mxOption to tell user how to change default optimizer, and how to add a fit function
- Refuse to set "Default optimizer" on models
- Reinstate R_CheckUserInterrupt (got commented out by mistake)
- Fail if a model has an expectation, no fitfunction, and no custom compute plan
- Fix signature mismatch in displayCompute
- Remove non-ascii em dashes
- Fix our signature for logLik S3 method
- If an MxMatrix is constant, avoid copy
- Advertise our version as 2.0.0 instead of 999.0.0
- Reimplement omxData storage to facilitate dynamic data
- omxAssignFirstParameters should not explode with 0 free parameters
- Move packageStartupMessage to .onAttach
- Reject vector=TRUE as part of MxFitFunctionMultigroup
- Make FIML respect vector=FALSE

6.5 Release 2.0.beta1-3473 (May 30, 2014)

- Fix for R 3.1
- Add packageStartupMessage if compiled without OpenMP
- Remove extra copy of # of evaluations
- Fix algebra dependency tracking
- Name anonymous algebras to aid debugging
- numThreads is always 1 without openmp
- MxRAMModel should not assume MxExpectationRAM
- Permit optimization directly on mxFitFunctionML(vector=TRUE)

- Remove unimplemented and crashing omxSetFinalReturnsFIMLFitFunction
- Added a warning about using "one" as a label in mxPaths.
- minor error msg mod to suggest action if non-square matrix declared as cov
- Warn if summary is called on a model that was modified after mxRun
- Bug Fix: the backend resized 1x1 matrices to MxN when used in scalar/elementwise multiplication, but then did not resize them back in the frontend.
- Fix mxRename with constraints bug.
- Learn mxMatrix dimnames from values if explicit dimnames omitted
- fix issue 'summary() of fitted mxModel object returns error'
- Confidence Intervals now give the name of the parameter if they are from an MxMatrix instead of the Matrix-Name[row,col].
- Exterminate strncmp
- Distinguish between sorted/unsorted and whether sorting is requested
- Remove unneeded parameter from isErrorRaised()
- Fixed Saturated Likelihood Bug by not populating that attribute for FIML.
- Support NPSOL warmStart
- Add option to checkpoint every evaluation
- Bug fix the number of observed statistics in non-IFA models with raw data.
- In addition to the mxOption, let mxData(sort=FALSE) request unsorted data
- Add a git bisect script
- Fixed bug where mxOption 'Default optimizer', and other options as well, get overwritten to the global defaults
- logLik for mxModel (contributed by Andreas Brandmaier)
- mxOption without a value to show the current setting
- Place likelihood-based CI code into a separate MxCompute step
- Add rownames to standardErrors
- Do not resetDataSortingFlags
- Remove unused arguments from omxRaiseError, omxRaiseErrorf
- Remove unused argument from omxResizeMatrix
- Handle omxRemoveRowsAndColumns for row-major order; add tests
- omxAliasMatrix, omxResetAliasedMatrix are essentially omxCopyMatrix
- Simplify implementation of aliased matrices
- Fix memory corruption in omxResetAliasedMatrix
- Checkpoint rewrite
- Allow standard errors without Richardson extrapolation
- Preserve dimnames in assignments to MxMatrix slots
- Rework reporting of iterations and optimizer status; add failing test
- Fix memory corruption in matrix populate lists

- Remove unused omxInitMatrix argument
- Add more gdb hints
- Fix misuse of stdargs macros
- Enable gdb for the regular 'make test' variants
- Hook up recordIterationError for FIML
- Add FitContext::recordIterationError
- Enumerate errors if there is more than one
- Report all the errors instead of only the most recent one
- mxCI should not interfere with SEs (or fit)
- Add dimnames to @output$hessian and @output$ihessian
- Fix omxMatrix memory model
- Remember how many cores we detected
- Fix mxCI for a vector of parameter names; add test
- Hook up numerical integration precision parameters
- Improve detection of CPU architecture
- Minor bug fix in saturated model generator
- Pull libnpsol.a from our website (linux only, so far)
- Hint the correct way to customize the compiler/compiler options
- Add info on using omxAssignFirstParameters with SetParameters
- Add mxOption for max stack depth
- Factor out lots of calls to Rf_mkChar
- Remove "at iteration" from error messages (simplifies log diff)
- Make elementwise algebra ops conformable for the scalar-matrix case
- Continue with omxMatrix API simplification
- Enable R_NO_REMAP for a cleaner namespace
- Add check for algebra dimnames
- Don't use R to calculate our algebra result matrix dimensions
- Initial compute protocol for the whole tree of dependencies
- Don't rely on R to evaluate our algebra
- Remove non-reproducable pointer addresses from logs (makes it possible to diff logs)
- Change backend initialization order
- Set up the usual gdb breakpoints automatically
- Update references to mxFitFunctionAlgebra and correct dot multiplication explanation in mxAlgebra.Rd
- Improve some conformability checks by showing the dimension mismatch
- Always report @output$fit but don't report @output$minimum unless it is

- Using the name "stderr" with #pragma omp critical can cause conflicts when building multithread binaries under Windows.
- Permit rescaling of log-likelihood
- Compute condition number of information matrix
- Control some ComputeEstimatedHessian knobs from R
- Do not use roxygen to create Collate field
- Reduce use of protect stack
- Suggest how to debug protect stack overflow
- Permit easy specification of the default optimizer in R code
- Keep parameters within bounds
- Rename most PPML functions to keep them unexported
- Allow summary to work with unnamed estimates
- Fixed a bug where ML fit functions with raw data (FIML) and vector=TRUE set were returning a single value instead of a vector.
- Updating twinData.Rd to document the reuse of zyg 6:10. Also added Nick Martin reference
- Document how to find & adjust R's default compiler flags
- Add back some UNPROTECTs
- Be more paranoid about importing mxData to backend
- Avoid final copyParamToModel when not needed
- Remove obsolete performance counters
- Fix/remove improper printf style formats
- Change how dirty matrices are indicated
- Rewrite dependency tracking
- Grab expectation names in the backend
- Disable NPSOL gradient verification by default (this is a developer feature)
- Permit better control over whether NPSOL uses gradients
- Remove most instances of setFinalReturns, mark deprecated
- Don't communicate unused dependency information to the backend
- Store fit function name (regression fix?)
- Free parameter groups
- Store omxGlobal as a pointer to ensure proper init & destruction
- Threads don't help RFitFunction, force single threaded
- NPSOL linear constraints are unused (deadcode)
- Set up a build rule for installing without NPSOL
- Replace bloated autoconf script with a simple shell script
- Prevent developers from using Rprintf
- Split omxState into truly global stuff and per-thread stuff

- Modified mxSaturatedModel to take either a model or a data set, and added a run= argument with FALSE as the default.
- mxSaturated model support for ordinal data.
- Skip recomputing thresholds when they don't exist
- Replace completely broken PBS cpus detection
- Add thread-safe logging functions
- Changed all objective functions to return NAMED lists of expectation and fitfunction.
- Switch over to new structured Compute system
- Added manual page for mxFitFunctionML
- Encapsulate NPSOL into omxComputeGD; rework error status reporting
- Pass blank gradient and hessian to omxNPSOLConfidenceIntervals
- Reorder matrix name processing
- Dump matrices using valid R syntax (debugging)
- Remove unimplemented OMX_SOCKET_CHECKPOINT
- Simplify management of omxData.type
- Switched backend from C to C++
- Fixed maximum number of dimensions bug so that only dimensions other than those -inf to +inf are counted
- Add API for internal expectations; remove redundent copy of rObj
- C-side support for submodels
- Add deps on MASS and mvtnorm
- Remove lots of UNPROTECTs
- Refactor allocation of omxAlgebra.args
- Teach MxExpectation to store its relationship in the model tree
- Add default A,S,F matrix names for mxExpectationRAM
- Process submodels concurrently with manifest & latent variables (early)
- Simplify error reporting of unrecognized arguments to mxModel
- Factor out interpretation of mxPath's connect argument
- fix insertMeansPathRAM to catch arrows=2
- minor fix to insertMeansPathRAM() 1. catch arrows != 1 in means paths; 2. reword error
- Automatically balance PROTECT/UNPROTECT
- Added suggestion for solution when mxData are cov/cor and not symmetric.
- Improved mxData error messages
- Added cov2cor and chol to supported functions.
- Update mxCompare() to handle missing comparison parameter
- update showFitStatistics() to handle empty compareSummaries list
- Added comment to RAM model error, prompting user to view ?mxData when their model has no data

- Add varargs replacement for omxRaiseError
- Make omxCheckCloseEnough compare missingness pattern too
- Complain if any starting values are missing
- Fix uninitialised memory access in omxSelectRows & omxSelectCols, Bug only affected non-square matrices
- changed FIML fit functions to report all elements when vector=TRUE
- added mxThreshold() function
- migrated from MxObjective* to mxFitFunction* and mxExpectation*
- bug fix in multiple-iteration upper confidence interval estimation
- bug fix in mxRun() where a model with grandchildren submodels can have its internal state corrupted
- Added vech2full and vechs2full mxAlgebras: inverses of vech and vechs
- detect non-conformable arrays for elementwise division in the front-end
- fix crash in Hessian calculation when no objective function is specified
- warn if standard errors are enabled and the numeric Hessian calculation is disabled
- added confidence interval calculations to checkpoint output

6.6 Release 1.3.0-2168 (September 17, 2012)

- added mxOption() for "Analytic Gradients" with possible values "Yes"/"No"
- added 'cache' and 'cacheBack' arguments to mxEval()
- added omxLocateParameters() function (see ?omxLocateParameters)
- type='RAM' allowing 'manifestVars' argument to appear in a different order than in the observed covariance matrix.
- the configuration mxRun(model, checkpoint = TRUE) writes a line in the checkpoint file at the conclusion of model optimization.
- added "Always Checkpoint" to mxOptions() with values "Yes" or "No"
- header for the checkpoint file will identify anonymous free parameters with the string modelName.matrixName[row,col]
- omxGetParameters() and omxSetParameters() support anonymous free parameters
- implemented cov2cor in the OpenMx backend
- mxOption "Major iterations" accepts either a value or a function
- now tracking the MxAlgebra and MxMatrix objects that need to be updated when populating free parameters or updating definition variables.
- performance improvements in mxModel() when building RAM models
- performance improvements in mxRun() frontend for large matrices
- rewrite on the processing of objective functions in the frontend
- added R functions omxCbind(), omxRbind(), and omxTranspose()
- added 'fetch' argument to omxGetParameters()

6.7 Release 1.2.5-2156 (September 5, 2012)

- bugfix to omxRAMtoML() when input model has covariance data
- fixing memory leak in cleaning up algebras when optimization is complete
- fixing memory leak in omxImaginaryEigenvalues()
- fixed a bug that manifests in confidence intervals with definition variables (see http://openmx.psyc.virginia.edu/thread/1505)
- fixed a bug in the identification of NA definition variables (see http://openmx.psyc.virginia.edu/thread/1521)

6.8 Release 1.2.4-2063 (May 22, 2012)

- added argument "name" to omxSetParameters()
- error checking for 0-length arguments to mxPath()
- fixing several memory leaks

6.9 Release 1.2.3-2011 (April 10, 2012)

- bugfix when (I-A)^-1 speedup is disabled
- bugfix in mxAlgebra() detection of missing operator or function
- bugfix in joint FIML when: 1) there are no definition variables, 2) the first (sorted) row with a new number of ordinal variables, and 3) has all continuous variables missing

6.10 Release 1.2.2-1986 (March 22, 2012)

- setting default value for mxOption("UsePPML") to "No" until the feature is adequately tested. Added test enforcing default value to test suite.

6.11 Release 1.2.1-1979 (March 21, 2012)

- bug fix for interaction of matrix transpose and square bracket substitutions
- renaming mxLISRELObjective() to imxLISRELObjective(), LISREL objective function is not yet implemented.
- improved error messages when a free parameter has multiple lbounds/ubounds
- improved error messages when passing strings into mxModel()
- improved error messages in mxEval()

6.12 Release 1.2.0-1926 (February 04, 2012)

- new interface such that 'objective@info$expMean', 'objective@info$expCov' and 'objective@info$likelihoods' are available in FIML or RAM objective functions.
- bug fix for omxSetParameters with lbound or ubound of NA.
- catching unknown matrix operator or function in mxAlgebra() or mxConstraint() declaration
- deprecated 'all' argument from mxPath function and replaced it with 'connect' argument. Also updated demos that used 'all' argument and documentation.
- removed 'excludeself' argument from mxPath() function
- added deprecation warning for argument 'all' = TRUE in mxPath()
- added support for OpenMP in hessian calculation
- added support for OpenMP in continuous FIML objective function
- added support for OpenMP in ordinal FIML objective function
- added support for OpenMP in joint ordinal/continuous FIML objective function
- added more descriptive error message to mxOption()
- serializing the sum operation on likelihoods - floating point addition is not associative

6.13 Release 1.1.2-1818 (October 24, 2011)

- fixed bug in summary() function, see http://openmx.psyc.virginia.edu/thread/1104
- fixed bug in independence model likelihood calculation for ML objective
- included support for gcc 4.5 and 4.6 under 32-bit x86

6.14 Release 1.1.1-1784 (September 11, 2011)

- fixed several bugs in joint ordinal-continuous integration
- fixed several typos in User Guide

6.15 Release 1.1.0-1764 (August 22, 2011)

- Joint and Ordinal documentation included and up to date.
- added deprecation warning for argument 'all'=TRUE in mxPath()
- added omxSelectRowsAndCols, omxSelectRows and omxSelectCols documentation
- fixed model flattening to keep track of confidence intervals in submodels
- added error checking for dimnames on MxMatrix and MxAlgebra objects
- reformated comments and heading style for all demos
- fixed segmentation fault on backend error condition
- turn off jiggling of free parameters with starting values of 0.0 when useOptimizer=FALSE

- allow non-RAM objective functions in RAM model
- change model type names to 'default' and 'RAM'
- no longer explicitly transforming RAM + raw data models into FIML models
- fixed bug in MxMatrix indexing operator
- added argument 'threshnames' to mxFIMLObjective() and mxRAMObjective()
- renaming majority of omx* functions to imx* functions. See http://openmx.psyc.virginia.edu/thread/761
- added error checking to mxOption() function
- added "Optimality tolerance" to mxOption() selection
- added 'lbound' and 'ubound' columns to summary() output of free parameters
- added asterisks to the 'lbound' and 'ubound' columns when feasibility tolerance is met
- added "omxNot" function to the set of available mxAlgebra() function
- added "omxSelectRows", "omxSelectCols", and "omxSelectRowsAndCols" as mxAlgebra operators()
- added "mean" function to the set of available mxAlgebra() functions
- added slots "expCov" and "expMean" to the MxRAMObjective function
- added useOptimizer option to mxRun.
- added error checking in frontend and backend for non-positive-definite observed covariance matrices
- added "omxGreaterThan", "omxLessThan", "omxApproxEquals", "omxAnd", and "omxOr" operators to the set of mxAlgebra() operators
- error checking for model[[1]] or model[[TRUE]]
- error checking in the front-end whether more than 20 ordinal columns are present in a data set
- improved performance in the front-end in mxModel() for adding paths to RAM models
- print name of algebra when operator has too few or too many arguments
- added mxErrorPool() function and R documentation.
- added Apache license information to all R documentation files.
- new implementation of mxEval().
- new argument 'defvar.row' to mxEval(). See ?mxEval.
- handling definition variables for (I - A) ^ - 1 speedup
- handling square bracket labels for (I - A) ^ - 1 speedup
- added argument 'free' to omxGetParameters. See ?omxGetParameters.
- added argument 'strict' to omxSetParameters. See ?omxSetParameters.
- eliminated warnings for confidence interval optimization codes
- added "..." argument to mxRObjectiveFunction()
- fix memory leak in RAM objective function
- removed dependency to MBESS library in R documentation
- added more descriptive error message when thresholds are not sorted
- incorporated NaN unsafe matrix-matrix multiplication (dgemm) from R <= 2.11.1

- incorporated NaN unsafe matrix-vector multiplication (dgemv) from R <= 2.11.1
- return NA in mxVersion() if "OpenMx" cannot be found
- fix infinite loop in objective function transformations
- added initialization to load OpenMx on swift workers
- implemented omxParallelCI() to calculate confidence intervals in parallel
- only calculating CIs for upper triangle of symmetric matrices
- cleanup appearance of transient MxMatrix objects in error messages
- fix bug with very large number of omxUntitledName() objects
- added optional argument CPUS=n to "make test" target
- change snowfall interface to use sfClusterApplyLB()
- storing raw data in row-major order, and copying contiguous data rows
- fix bug in mxModel() when using remove = TRUE

6.16 Release 1.0.7-1706 (July 6, 2011)

- error checking in front-end for non-positive-definite observed covariance matrices
- fix bug in MxMatrix indexing operator
- added deprecation warning for argument 'all'=TRUE in mxPath()

6.17 Release 1.0.6-1581 (March 10, 2011)

- Bug fix corner case in sorting data with definition variables

6.18 Release 1.0.5-1575 (March 8, 2011)

- added error checking in ML objective to match expected covariance and observed covariance matrices
- updated BootstrapParallel.R demo to use mxData() instead of model@data
- added the dataset from the Psychometrika article (www.springerlink.com/content/dg37445107026711)

6.19 Release 1.0.4-1540 (January 16, 2011)

- added initialization to load OpenMx on swift workers
- only calculating CIs for upper triangle on symmetric matrices
- fix bug with very large number of omxUntitledName() objects
- incorporated NaN unsafe matrix-vector multiplication (dgemv) from R <= 2.11.1
- fix bug in mxModel() when using remove = TRUE
- Added intervals to MxModel class documentation

6.20 Release 1.0.3-1505 (November 10, 2010)

- return NA in mxVersion() if "OpenMx" cannot be found
- eliminate infinite loop in objective function transformations

6.21 Release 1.0.2-1497 (November 5, 2010)

- missing <errno.h> include
- fix memory leak in RAM objective function
- incorporated NaN unsafe matrix-matrix multiplication (dgemm) from R <= 2.11.1

6.22 Release 1.0.1-1464 (October 8, 2010)

- bugfix for mxEval() and MxData objects
- handling definition variables for (I - A) ^ - 1 speedup
- handling square bracket labels for (I - A) ^ - 1 speedup
- added argument 'free' to omxGetParameters. See ?omxGetParameters.
- added argument 'strict' to omxSetParameters. See ?omxSetParameters.
- eliminated warnings for confidence interval optimization codes

6.23 Release 1.0.0-1448 (September 30, 2010)

- added missing entries to demo 00INDEX file

6.24 Release 0.9.2-1446 (September 26, 2010)

- added growth mixture models to user guide
- added initial Swift hook in omxLapply() - currenly activated only for mxRun calls
- fixed a bug in mxRename() when encountering symbol of missingness
- bugfix for crash when '*' is used instead of '%*%'
- feature removal: square brackets in MxMatrix labels now accept only literal values
- bugfix for definition variables used in mxRowObjective (which is still experimental)
- bugfix for (I - A) ^ -1 speedup with FIML optimization

6.25 Release 0.9.1-1421 (September 12, 2010)

- fixed a bug in -2 LL calculation in RAM models with definition variables and raw data

6.26 Release 0.9.0-1417 (September 10, 2010)

- improved error messages for non 1 x n means vectors in FIML and ML
- fixed a performance bug that was forcing too many recalculations of the covariance matrix in FIML optimizations.
- defualt behavior is to disable standard error calculations when model contains nonlinear constraints
- improved error messages for NA values in definition variables
- added error message when expected covariance dimnames and threshold dimnames do not contain the same elements.
- fixed a bug when mxRename() encounters a numeric or character literal.
- new chapters added to OpenMx user guide.
- fixed a bug in -2 log likelihood calculation with missing data

6.27 Release 0.5.2-1376 (August 29, 2010)

- improved error messages when identical label is applied to free and fixed parameters
- added 'onlyFrontend' optional argument to mxRun() function. See ?mxRun.
- disabling cbind() and rbind() transformations as they are broken.

6.28 Release 0.5.1-1366 (August 22, 2010)

- added error detection when multiple names are specified in mxMatrix(), mxAlgebra(), etc.
- removed 'digits' argument from mxCompare. Target behavior of argument was unclear. See ?mxCompare
- more informative error messages for 'dimnames' argument of objective functions
- more informative error messages when constraints have wrong dimensions
- improved error detected for 'nrow' and 'ncol' arguments of mxMatrix() function
- fixed a bug in ordinal FIML objective functions with non-used continuous data

6.29 Release 0.5.0-1353 (August 08, 2010)

- calculating cycle length of RAM objective functions
- bugfix: preserve rownames when converting data.frame columns to numeric values
- 'nrow' and 'ncol' arguments now supercede matrix dimensions in mxMatrix()
- add boolean argument 'vector' to mxRAMObjective() for returning the vector of likelihoods
- added demo(OneFactorModel_LikelihoodVector) as example of 'vector=TRUE' in RAM model
- cbind() and rbind() inside MxAlgebra expressions with all arguments as MxMatrix objects are themselves transformed into MxMatrix objects
- bugfix with square bracket substitution

- finishing implementing sorting of raw data in mxFIMLObjective()
- added 'RAM Optimization' and 'RAM Max Depth' to model options. See ?mxOption
- added support for linux x86_64 with gcc 4.1.x

6.30 Release 0.4.1-1320 (June 12, 2010)

- confidence interval optimizations now jitter if they can't get started
- added error checking for dimensions of expected means in ML + FIML objectives
- check for missing observed means when using (optional) expected means in ML
- added citation("OpenMx") information

6.31 Release 0.4.0-1313 (June 09, 2010)

- fixed bug in calculation of confidence intervals around non-objective values
- checking for partial square bracket references on input
- fixed error reporting for non-positive-definite covariances in FIML
- implemented initial sorting-based speedup for FIML objectives
- added 'No Sort Data' to mxOptions()
- eliminated getOption('mxOptimizerOptions') and getOption('mxCheckpointOptions')
- added getOption('mxOptions')
- error messages for illegal names provide function call information
- added 'estimates' column to confidence intervals in summary() of model
- fixed bug so checkpointing will work in R 2.9.x series
- fixed bug so mxRename() works on confidence interval specification
- renaming 'estimates' column of confidence interval summary output to 'estimate'

6.32 Release 0.3.3-1264 (May 24, 2010)

- confidence interval frontend was requesting nonexistent matrices
- omxNewMatrixFromMxMatrix() assumed input was always integer vector S-expression
- confidence intervals mislabeling free parameter names

6.33 Release 0.3.2-1263 (May 22, 2010)

- added 'newlabels' argument to omxSetParameters() function
- now throwing errors to the user when detected from the backend in mxRun()
- checkpointing mechanism implemented - mxRun(model, checkpoint = TRUE)

- never computing confidence intervals for matrix cells where free = FALSE (all bets are off on algebras)
- added mxOption(model, "CI Max Iterations", value)
- added documentation for mxRestore() function
- default expected means vectors are no longer generated

6.34 Release 0.3.1-1246 (May 09, 2010)

- new arguments to mxRun() for checkpointing and socket communication (doesn't work yet)
- throw an error if FIML objective has thresholds but observed data is not a data.frame object.
- bugfix for R version 2.11.0 is detecting as.symbol("") character as missing function parameter
- mxFactor() function accepts data.frame objects
- added mxCI() function to calculate likelihood-based confidence intervals
- mxCompare() shows model information for base models
- added flag all=[TRUE|FALSE] to mxCompare() function
- 'SaturatedLikelihood' argument to summary() function will accept MxModel object

6.35 Release 0.3.0-1217 (Apr 20, 2010)

6.35.1 new features

- implemented new ordinal data interface http://openmx.psyc.virginia.edu/thread/416#comment-1421 (except for 'means=0' component)
- added mxFactor() function (see ?mxFactor for help)
- added R documentation for rvectorize() and cvectorize()
- implemented eigenvalues and eigenvectors
- added 'numObs', 'numStats' arguments to mxAlgebraObjective()
- added 'numObs', 'numStats' arguments to summary()

6.35.2 changes to interface

- mxConstraint("A", "=", "B") is now written as mxConstraint(A == B)
- renaming 'cov' argument to 'covariance' in omxMnor()
- renaming 'lbounds' argument to 'lbound' in omxMnor()
- renaming 'ubounds' argument to 'ubound' in omxMnor()
- renaming 'cov' argument to 'covariance' in omxAllInt()
- argument 'silent = TRUE' to mxRun() will no longer suppress warnings
- argument 'suppressWarnings = TRUE' to mxRun() will suppress warnings

6.35.3 bug fixes

- added error checking for mxBounds() with undefined parameter names
- improving error messages in mxMatrix() function
- fixed aliasing bug in mxAlgebra() with external variables and constant values

6.35.4 internal

- added directory to repository for nightly tests ("make nightly")
- performance improvement to namespace conversion
- changing MxPath data structure from a list to an S4 object
- new signature for omxSetParameters(model, labels, free, values, lbound, ubound, indep)
- checked in implementation of mergesort
- changed mxCompare() signature to mxCompare(base, comparison, digits = 3)

6.36 Release 0.2.10-1172 (Mar 14, 2010)

- bugfix for assigning default data name when default data does not exist
- bugfix for sharing/unsharing data to a three-level hierarchy
- bugfix for error reporting in bad matrix access, uncalculated std. errors, and poor omxAllint thresholds
- implemented rvectorize, cvectorize algebra functions: vectorize by row, and vectorize by column
- not allowing the following forbidden characters in names or labels: "+-!~?:*/^%<>=&|$"
- added timestamp to summary() output
- reimplemented summary() function to handle unused data rows, and independent submodels
- reimplemented names(model), model$foo, and model$foo <- value to return all components of a model tree
- started work on an "OpenMx style guide" section to the User Guide.
- fix documentation + demo errors brought to our attention by dbishop

6.37 Release 0.2.9-1147 (Mar 04, 2010)

- bugfix for ordinal FIML with columns that are not threshold columns
- bugfix for detection of algebraic cycles when multiple objective functions are present
- test cases included for standard error calculation
- enabled standard error calculation by default

6.38 Release 0.2.8-1133 (Mar 02, 2010)

- bugfix for memory leak behavior in kronecker product calculation
- implemented kronecker exponentiation operator %^%.
- actually updating means calculations in ordinal FIML models
- removing standard errors from summary() until they are computed correctly

6.39 Release 0.2.7-1125 (Feb 28, 2010)

- added 'unsafe' argument to mxRun function. See ?mxRun for more information.
- bugfix for filtering definition variable assignment to current data source.
- bugfix for using data.frame with integer type columns that are not factors.

6.40 Release 0.2.6-1114 (Feb 23, 2010)

- implemented omxAllInt, use ?omxAllInt for R help on this function
- implemented omxMnor, use ?omxMnor for R help on this function
- added option to calculate Hessian after optimization. See ?mxOption for R help.
- added option to calculate standard errors after optimization. See ?mxOption for R help.
- added R documentation for vech, vechs, vec2diag, and diag2vec
- performance improvements for independent submodels
- added 'silent=FALSE' argument to mxRun() function
- added R documentation for omxApply(), omxSapply(), and omxLapply()
- wrote mxRename() and added R documentation for function
- added 'indep' argument to summary() to ignore independent submodels (see ?summary)
- added 'independentTime' to summary() output. Wall clock time for independent submodels.
- added 'wallTime' and 'cpuTime' to summary() output. Total wall clock time and total cpu time.
- implemented ':' operator for MxAlgebra expressions. 1:5 returns the vector [1,2,3,4,5]
- implemented subranges for '[' operator in MxAlgebra expressions. foo[1:5,] is valid inside algebra.
- added omxGetParameters, omxSetParameters, omxAssignFirstParameters. Use ? for documentation.
- renamed all objective function generic functions from omxObj* to genericObj*
- enumerated OpenMx and NPSOL options in ?mxOption documentation
- implemented foo[x,y] and foo[x,y] <- z for MxMatrix objects

6.41 Release 0.2.5-1050 (Jan 22, 2010)

- added 'mxVersion' slot to output of summary() function.
- added documentation of summary() function.
- set default function precision for ordinal FIML evaluation to "1e-9"
- not throwing an error on mxMatrix('Full', 3, 3, labels = c(NA, NA, NA))
- applying identical error checking to single thresholds matrix and single thresholds algebra
- implemented generic method names() for MxModel objects.
- implemented vec2diag() and diag2vec() matrix algebra functions.

6.42 Release 0.2.4-1038 (Jan 15, 2010)

- definition variables can now be used inside algebra expressions
- definition variables inside of MxMatrices will populate to the 1st row before conformability checking. In plain english: you do not need to specify the starting values for definition variables.
- the square-bracket operator when used in MxMatrix labels is no longer restricted to constants for the row and column. The row and column arguments will accept any term that evaluates to a scalar value or a (1 x 1) matrix.
- summary() on a model returns a S3 object. Behaves like summary() in stats package.
- eliminated UnusualLabels.R test case. Too many problems with windows versus OS X versus linux.
- implemented vech() and vechs() functions: half-vectorization and strict half-vectorization
- fixed bug in ordinal FIML when # of data columns > # of thresholds
- added 'frontendTime' and 'backendTime' values to summary() output. They store the elapsed time of a model in the R front-end and C back-end, respectively.
- created a name space for the OpenMx library. Only mx**() and omx**() functions should be exported to the user, plus several miscellaneous matrix functions and S4 generic functions.
- corrected 'observedStatistics' output of summary() to exclude definition variables
- corrected 'observedStatistics' output of summary() count the number of equality constraints

6.43 Release 0.2.3-1006 (Dec 04, 2009)

- added 'vector' argument to mxFIMLObjective() function. Specifies whether to return the likelihood vector (if TRUE) or the sum of log likelihoods (if FALSE). Default value is FALSE.
- renamed omxCheckEquals() to omxCheckIdentical(). omxCheckIdentical() call "identical" so that NAs can be compared.
- added checking of column names of F and M matrices in RAM objective functions.
- added 'dimnames' argument to mxRAMObjective() function. Populates the column names of F and M matrices.
- added square bracket operator to MxAlgebra expressions. A[x,y] or A[,y] or A[x,] or A[,] are valid.
- square bracket operator supports row and column string arguments.
- mxModel(remove=TRUE) accepts both character names or S4 named entities.

- added support for x86_64 on OS X 10.6 (snow leopard)
- fixed support for x86_64 on Ubuntu 9.10 (gcc 4.4)
- throw error message when inserting a named entity into a model with an identical name
- added Anthony William Fairbank Edwards "Likelihood" (1972; 1984) A, B, O blood group example to online documentation

6.44 Release 0.2.2-951 (Oct 29, 2009)

- omxGraphviz() either prints to stdout or to a filename
- updated omxGraphviz() to draw an arrow if (value != 0 || free == TRUE || !is.na(label))
- omxGraphviz() returns a character string invisibly
- error checking for bogus definition variables
- summary() uses matrix dimnames by default, use options('mxShowDimnames'=FALSE) to disable
- added support for gcc 4.4 (Ubuntu 9.10) on x86 and x86_64 architectures
- created R documention for omxGraphviz() function
- generalized dependency specification for objective functions (omxObjDependencies)
- fixed cross-reference links in User Guide

6.45 Release 0.2.1-922 (Oct 10, 2009)

- checked observed data for dimnames on ML objective functions
- using dimnames= argument to mxMLObjective() and mxFIMLObjective() propagates to MxMatrix objects on output.
- bug fix for error message where model name is incorrect
- updated user guide in response to feedback from beta testers
- incremented version number to 0.2.1-922 to sync demos and user guide

6.46 Release 0.2.0-905 (Oct 06, 2009)

- several of the twin model demo examples have been recoded.
- fixed bug with non-floating point matrices.
- more error checking for mxPath().
- tools/mxAlgebraParser.py will convert Mx 1.0 algebra expressions (Python PLY library is required).
- renamed "parameter estimate" column to "Estimate" and "error estimate" to "Std.Error" in summary().
- added 'dimnames' argument to mxFIMLObjective() and mxMLObjective().
- error checking for RAM models with non-RAM objective functions.

6.47 Release 0.1.5-851 (Sep 25, 2009)

- improved error messages on unknown identifier in a model (beta tester issue)
- fixed bug in mxMatrix() when values argument is matrix and byrow=TRUE
- implemented square-bracket substitution for MxMatrix labels
- fixed a bug in computation of omxFIMLObjective within an algebra when definition variables are used
- significant alterations to back-end debugging flags
- tweaked memory handling in back-end matrix copying
- added support for x86_64 linux with gcc 4.2 and 4.3

6.48 Release 0.1.4-827 (Sep 18, 2009)

- added checking and type coercion to arguments of mxPath() function (a beta tester alerted us to this)
- moved matrices into submodels in UnivariateTwinAnalysis_MatrixRaw demo
- added Beginners Guide to online documentation
- mxRun() issues an error when the back-end reports a negative status code
- named entities and free or fixed parameter names cannot be numeric values
- constant literals are allowed inside mxAlgebra() statements, e.g. mxAlgebra(1 + 2 + 3)
- constant literals can be of the form 1.234E+56 or 1.234e+56.
- type checking added to mxMatrix arguments (prompted by a forum post)
- mxPath() issues an error if any of the arguments are longer than the number of paths to be generated
- data frames are now accepted at the back-end
- FIML ordinal objective function is now working. Still a bit slow and inelegant, but working
- FIML ordinal now accepts algebras and matrices. dimnames of columns must match data elements
- implemented free parameter and fixed parameter substitution in mxAlgebra statements
- implemented global variable substitution in mxAlgebra statements
- turned off matrix and algebra substitution until a new proposal is decided
- snow and snowfall are no longer required packages
- added cycle detection to algebra expressions
- mxEval() with compute = TRUE will assign dimnames to algebras
- added dimnames checking of algebras in the front-end before optimization is called
- added 'make rproftest' target to makefile

6.49 Release 0.1.3-776 (Aug 28, 2009)

- mxEvaluate() was renamed to mxEval() after input from beta testers on the forums.
- new function mxVersion that prints out the current version number (beta tester request).

- When printing OpenMx objects, the @ sign is used where it is needed if you would want to print part of the object (beta tester request).
- now supports PPC macs.
- implemented AIC, BIC and RMSEA calculations.
- mxMatrix documentation now talks about lower triangular matrices (beta tester request).
- fixed bugs in a number of demo scripts.
- added chi-square and p-value patch from beta tester Michael Scharkow.
- added comments to demo scripts.
- fixed a bug in the quadratic operator (a beta tester alerted us to this).
- means vectors are now always 1xn matrices (beta tester request).
- added an option "compute" to mxEval() to precompute matrix expressions without going to the optimizer.
- Matrix algebra conformability is now tested in R at the beginning of each mxRun().
- named entities (i.e. mxMatrices, mxAlgebras, etc.) can no longer have the same name as the label of a free parameter. (This seems obscure, but you will like what we do with it in the next version!)
- can use options(mxByrow=TRUE) in the R global options if you always read your matrices in with the by-row=TRUE argument. Saves some typing. (beta tester request)
- fixed the standard error estimates summary.
- added mxVersion() function to return the version number (as a string).

6.50 Release 0.1.2-708 (Aug 14, 2009)

- **Added R help documentation for omxCheckCloseEnough(), omxCheckWithinPercentError(),** omxCheckTrue(), omxCheckEquals(), and omxCheckSetEquals()
- **(mxMatrix) Fixed a bug in construction of symmetric matrixes.**
  - now supports lower, standardized, and subdiagonal matrices.

6.51 Release 0.1 (Aug 03, 2009)

- (mxEvaluate) mxEvaluate translates MxMatrix references, MxAlgebra references, MxObjectiveFunction references, and label references.
- (mxOptions) added 'reset' argument to mxOptions()
- **(mxPath) renamed 'start' argument of mxPath() to 'values'**
  - renamed 'name' argument of mxPath() to 'labels'
  - renamed 'boundMin' argument of mxPath() to 'lbound'
  - renamed 'boundMax' argument of mxPath() to 'ubound'
  - eliminated 'ciLower' argument of mxPath()
  - eliminated 'ciUpper' argument of mxPath()
  - eliminated 'description' argument of mxPath()

- **(dimnames) implemented dimnames(x) for MxMatrix objects**
  - implemented dimnames(x) <- value for MxMatrix objects
  - implemented dimnames(x) for MxAlgebra objects
  - implemented dimnames(x) <- value for MxAlgebra objects
- (mxMatrix) added 'dimnames' argument to mxMatrix()
- (mxData) renamed 'vector' argument of mxData() to 'means'

CHAPTER

SEVEN

REFERENCE

- OpenMx R documentation

CHAPTER

EIGHT

INDICES AND TABLES

- *search*

[LI1986] Li, C.C. (1986). Path Analysis - A Primer. The Boxwood Press, Pacific Grove, CA.

[RAM1990] McArdle, J.J. & Boker, S.M. (1990). RAMpath: Path diagram software. Denver: Data Transforms Inc.

[BockAitkin1981] Bock, R. D. & Aitkin, M. (1981). Marginal maximum likelihood estimation of item parameters: Application of an EM algorithm. Psychometrika, 46, 443–459.

[Cai2010] Cai, L. (2010). High-dimensional exploratory item factor analysis by a Metropolis–Hastings Robbins–Monro algorithm. Psychometrika, 75(1), 33-57.

[CaiYangHansen2011] Cai, L., Yang, J. S., & Hansen, M. (2011). Generalized full-information item bifactor analysis. Psychological Methods, 16(3), 221-248.

[Embretson1996] Embretson, S. E. (1996). The new rules of measurement. Psychological Assessment, 8(4), 341-349.

[OrlandoThissen2000] Orlando, M. and Thissen, D. (2000). Likelihood-Based Item-Fit Indices for Dichotomous Item Response Theory Models. Applied Psychological Measurement, 24(1), 50-64.

[WatsonEtal1988] Watson, D., Clark, L. A., & Tellegen, A. (1988). Development and validation of brief measures of positive and negative affect: The PANAS scales. Journal of Personality and Social Psychology, 54 (6), 1063.

[White1994] Estimation, Inference and Specification Analysis. Cambridge University Press, Cambridge.

[Yen1993] Yen, W. M. (1993). Scaling performance assessments: Strategies for managing local item dependence. Journal of Educational Measurement, 30, 187-213.