OpenMx Documentation

Release 1.0.0-1448

OpenMx Development Team

September 30, 2010

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

1.1 Beginners Guide to OpenMx

This document will walk the reader through the basic concepts used in the OpenMx library. It will assume that you have successfully installed the R statistical programming language [http://www.r-project.org/] and the OpenMx library for R [http://openmx.psyc.virginia.edu]. Before we begin, let us start with a mini-lecture on the R programming language. Our experience has found that this exercise will greatly increase your understanding of subsequent sections of the introduction. Detailed introductions to R can be found on the internet.

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

1.1.1 Pass By Value

```
1 addone <- function(number) {
2     number <- number + 1
3     return(number)
4 }
5
6 avariable <- 5
7
8 print(addone(avariable))
9 print(avariable)
```

In the previous code block, the variables `addone` and `avariable` are defined. The value assigned to `addone` is a function, while the value assigned to `avariable` is the number 5. The function `addone` takes a single argument, adds one to the argument, and returns the argument back to the user. What is the result of executing this code block? Try it. The correct result is 6 and 5. But why is the variable `avariable` still 5, 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 *variables* and *values*. The *variables* declared in this example are `addone`, `avariable`, and `number`. The *values* refer to the things that are stored by the *variables*. 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 variable `avariable` cannot be modified by the function `addone`. If I wanted to update the value stored in the variable, I would have needed to replace line 8 with the expression `print(avariable <- addone(avariable))`. Try it. The updated example prints out 6

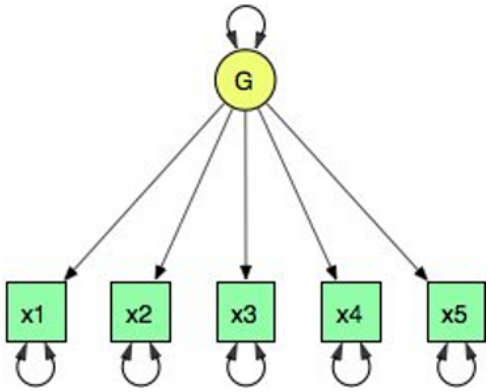
and 6. The lesson from this exercise is that the only way to update a variable in a function call is to capture the result of the function call [1]. This lesson is sooo important that we'll repeat it:

- the only way to update a variable 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).

1.1.2 Path Model Specification

Below is a figure of a one factor model with five indicators. 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.




```
require(OpenMx)

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

factorModel <- 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)
)

factorFit <- mxRun(factorModel)
summary(factorFit)
```

This example uses a RAM-style specification technique. Let's break down what is happening in each section of this example.

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

Preamble

Every OpenMx script must begin with either `library(OpenMx)` or `require(OpenMx)`. These commands will load the OpenMx library.

Reading Data

The `data` function can be used to read sample data that has been pre-packaged into the R library. 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.

Model Creation

The `mxModel` function is used to create a model. 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 for latent variables (`latentVars=`). In this case the manifest variables are `c("x1", "x2", "x3", "x4", "x5")` and the latent variable is `c("G")`.

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. By default, 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.

Objective Function Creation

When using a path specification of the model, the objective function is always `RAM`.

Data Source Creation

A `mxData` function is used to construct a data source for the model. In this example, we are specifying a covariance matrix. 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 if required the number of observations (`numObs`).

Model Population

The `mxModel` function is somewhat of a swiss-army knife. The first argument to the `mxModel` function can be a `name`, in case it is a newly generated model or a previously defined model. In the latter case, the new model 'inherit's all the characteristics (arguments) of the old model, which can be changed with additional arguments. An `mxModel` can contain `mxPath`, `mxData`, `mxObjective` and other `mxModel` statements as arguments.

Model Execution

The `mxRun` function will run a model through the optimizer. The return value of this function is an identical model, with all the free parameters in the elements of the matrices of the model assigned to their final values. The summary function (`summary(modelname)`) is a convenient method for displaying the highlights of a model after it has been executed.

1.1.3 Matrix Model Specification

```
require(OpenMx)

data(demoOneFactor)

factorModel <- mxModel(name="One Factor",
    mxMatrix(type="Full", nrow=5, ncol=1, free=TRUE, values=0.2, name="A"),
    mxMatrix(type="Symm", nrow=1, ncol=1, free=FALSE, values=1, name="L"),
    mxMatrix(type="Diag", nrow=5, ncol=5, free=TRUE, values=1, name="U"),
    mxAlgebra(expression=A %*% L %*% t(A) + U, name="R"),
    mxMLObjective(covariance="R", dimnames = names(demoOneFactor)),
    mxData(observed=cov(demoOneFactor), type="cov", numObs=500)
)

factorFit <- mxRun(factorModel)
summary(factorFit)
```

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. Let's break down what is happening in each section of this example.

Preamble

Every OpenMx script must begin with either `library(OpenMx)` or `require(OpenMx)`. These commands will load the OpenMx library.

Reading Data

The `data` function can be used to read sample data that has been pre-packaged into the R library. 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.

Model Creation

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, objective 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
algebra	MxAlgebra
objective function	MxObjectiveFunction
constraint	MxConstraint
data source	MxData

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 variable that is used to identify the entity. In our example, the variable `factorModel` is created with the `mxModel` function and stores a value that is a "MxModel" object with the name `One Factor`.

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 a 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.

Each `MxMatrix` object is a container that stores five matrices of equal dimensions. The five matrices stored in a `MxMatrix` object are: `values`, `free`, `labels`, `lbound`, and `ubound`. `Values` stores the current values of each element in the matrix. `Free` stores a boolean that determines whether a element is free or fixed. `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, i.e. 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 their names.

Objective Function Creation

`MxObjective` constructs an objective function for the model. For this example, we are using a maximum likelihood objective function and specifying an expected covariance algebra and omitting an expected means algebra. The expected covariance algebra is referenced according to its name. The objective function for a particular model is given the name `objective`. Consequently there is no need to specify a name for objective function objects. We need to assign `dimnames` for the rows and columns of the covariance matrix, such that a correspondence can be determined between the expected and the observed mean vectors / covariance matrices.

Data Source Creation

An `mxData` function provides a data source for the model. In this example, we are specifying a covariance matrix. The data source for a particular model is given the name `data`. Consequently there is no need to specify a name for data objects.

Model Population

The mxModel function is somewhat of a swiss-army knife. If the first argument to the `mxModel` function is an existing model, then the result of the function call is a new model with the remaining arguments to the function call added or removed from the model (depending on the 'remove' argument, which defaults to FALSE). Alternatively, we can give it a `name` and populate the model with matrices, algebras, an objective function, and a data source, which are all arguments of the `mxModel`.

Model Execution

The `mxRun` function will run a model through the optimizer. The return value of this function is an identical model, with all the free parameters in the elements of the matrices of the model assigned to their final values. The summary function (`summary(modelname)`) is a convenient method for displaying the highlights of a model after it has been executed.

Alternative Formulation

Rather than adding the paths/matrices/algebras, objective function and data as arguments to the `mxModel`, which we will use primarily throughout the documentation, we can also create 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 variable that is used to identify the entity. In our example, the variable `matrixA` stores a value that is a MxMatrix object with the name "A".

```
1  require(OpenMx)
2
3  data(demoOneFactor)
4
5  factorModel <- mxModel(name="One Factor")
6
7  matrixA <-  mxMatrix(type="Full", nrow=5, ncol=1, free=TRUE, values=0.2, name="A")
8  matrixL <-  mxMatrix(type="Symm", nrow=1, ncol=1, free=FALSE, values=1, name="L")
9  matrixU <-  mxMatrix(type="Diag", nrow=5, ncol=5, free=TRUE, values=1, name="U")
10
11 algebraR <- mxAlgebra(expression=A %*% L %*% t(A) + U, name="R")
12
13 objective <- mxMLObjective(covariance="R", dimnames = names(demoOneFactor))
14 data <- mxData(observed=cov(demoOneFactor), type="cov", numObs=500)
15
16 factorModel <- mxModel(
17                 factorModel, matrixA, matrixL, matrixU, algebraR, objective, data)
18
19 factorFit <- mxRun(factorModel)
20 summary(factorFit)
```

Note that lines 5 and 16 could have been combined with the following call:

```
factorModel <- mxModel(
                matrixA, matrixL, matrixU, algebraR, objective, data, name="One Factor")
```

1.2 Quick Overview

This document provides a quick overview of the capabilities of OpenMx in handling everything from simple calculations to optimization of complex multiple group models. Note that this overview focuses on matrix specification of OpenMx models, and that all these models can equally well be developed using path specification. We start with a matrix algebra example, followed by an optimization example. We end this quick overview with a twin analysis example, as the majority of prior Mx users will be familiar with this type of analysis. We plan to expand this overview with examples more relevant to other applications of structural equation modeling. While we provide the scripts here, we will not discuss every detail, as that is done more systematically in the next chapters with examples shown in two model styles and two data styles. We describe in detail a series of models using path specification in chapter two, followed by the same examples in matrix specification in chapter three.

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

- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/MatrixAlgebra.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/BivariateCorrelation.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/UnivariateTwinAnalysis.R

1.2.1 Simple OpenMx Script

We will start by showing some of the main features of OpenMx using simple examples. For those familiar with Mx, it is basically a matrix interpreter combined with a numerical optimizer to allow fitting statistical models. Of course you do not need OpenMx to perform matrix algebra as that can already be done in R or by hand. However, to accommodate flexible statistical modeling of the type of models typically fit in Mx, Mplus or other SEM packages, special kinds of matrices and functions are required which are bundled in OpenMx. We will introduce key features of OpenMx using a matrix algebra example. Remember that R is object-oriented, such that the results of operations are objects, rather than just matrices, with various properties/characteristics attached to them. We will describe the script line by line; a link to the complete script is here.

Say, we want to create two matrices, **A** and **B**, each of them a 'Full' matrix with 3 rows and 1 column and with the values 1, 2 and 3, as follows:

$$A = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad B = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

we use the `mxMatrix` command, and define the type of the matrix (`type=`), number of rows (`nrow=`) and columns (`ncol=`), its specifications (`free=`) and starting values (`values=`), optionally labels (`labels=`), upper (`ubound=`) and lower (`lbound=`) bounds, and a name (`name=`).

```
mxMatrix(
    type="Full",
    nrow=3,
    ncol=1,
    values=c(1,2,3),
    name='A'
)
```

The matrix **A** can be stored as the R object `A` by assigning the MxMatrix to the string A

```
A <- mxMatrix(
    type="Full",
    nrow=3,
    ncol=1,
```

```
    values=c(1,2,3),
    name='A'
)
```

or we can include the mxMatrix statement as an argument in an `mxModel` - here saved as the R object `exercise` and given the name `declare matrices` - together with other matrices and possibly calculations based on those matrices using `mxAlgebra` statements. All the arguments of an `mxModel` are separated by a comma. The model below contains just two matrices, **A** and **B**. Note that the last argument is not followed by a comma, but rather by an right bracket to close the `mxModel`.

```
exercise <- mxModel("declare matrices",
    mxMatrix(
        type="Full",
        nrow=3,
        ncol=1,
        values=c(1,2,3),
        name='A'
    ),
    mxMatrix(
        type="Full",
        nrow=3,
        ncol=1,
        values=c(1,2,3),
        name='B'
    )
)
```

Assume we want to calculate the (q1) the sum of the matrices **A** and **B**, (q2) the element by element multiplication (Dot product) of **A** and **B**, (q3) the transpose of matrix **A**, and the (q4) outer and (q5) inner products of the matrix **A**, using regular matrix multiplication, i.e.:

$$q1 = A + B \tag{1.1}$$
$$q2 = A.A \tag{1.2}$$
$$q3 = t(A) \tag{1.3}$$
$$q4 = A * t(A) \tag{1.4}$$
$$q5 = t(A) * A \tag{1.5}$$

we invoke the `mxAlgebra` command which performs an algebra operation between previously declared matrices, which are all included within an `mxModel`. Note that in R, transpose is represented by `t()`, regular matrix multiplication `\%*\%` and dot multiplication as `*`. We also assign the algebras a name to refer back to them later:

```
mxAlgebra(
    expression=A + B,
    name='q1'
)

mxAlgebra(
    expression=A * A,
    name='q2'
)

mxAlgebra(
    expression=t(A),
    name='q3'
)
```

```
mxAlgebra(
    expression=A %*% t(A),
    name='q4'
)

mxAlgebra(
    expression=t(A) %*% A,
    name='q5'
)
```

For the algebras to be evaluated, they become arguments of the `mxModel` command, as do the declared matrices, each separated by comma's. The model, which is here saved as the R object `algebraExercises` and given the name `perform algebra on matrices`, is then executed by the `mxRun` command, as shown in the full code below, saved in an R file `matrixAlgebra.R`:

```
require(OpenMx)

algebraExercises <- mxModel("perform algebra on matrices",
    mxMatrix(type="Full", nrow=3, ncol=1, values=c(1,2,3), name='A'),
    mxMatrix(type="Full", nrow=3, ncol=1, values=c(1,2,3), name='B'),
    mxAlgebra(expression=A+B, name='q1'),
    mxAlgebra(expression=A*A, name='q2'),
    mxAlgebra(expression=t(A), name='q3'),
    mxAlgebra(expression=A%*%t(A), name='q4'),
    mxAlgebra(expression=t(A)%*%A, name='q5')
)

answers <- mxRun(algebraExercises)
answers@algebras
result <- mxEval(list(q1,q2,q3,q4,q5),answers)
```

The resulting R object `answers` from running the OpenMx script `algebraExercises` contains the same matrices and algebras with the values in the algebras being the result of the calculations on the matrices.

As you notice, we added some lines at the end to generate the desired output. As the resulting matrices and algebras are stored in `answers`; we can refer back to them by specifying `answers@matrices` or `answers@algebras`. We can also calculate any additional quantities or perform extra matrix operations on the results using the `mxEval` command. For example, if we want to see a list of all the answers to the questions in `matrixAlgebra.R`, the results would look like this:

```
[[1]]
     [,1]
[1,]    2
[2,]    4
[3,]    6

[[2]]
     [,1]
[1,]    1
[2,]    4
[3,]    9

[[3]]
     [,1] [,2] [,3]
[1,]    1    2    3

[[4]]
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    4    6
[3,]    3    6    9

[[5]]
     [,1]
[1,]   14
```

So far, we have introduced five new commands: `mxMatrix`, `mxAlgebra`, `mxModel`, `mxRun` and `mxEval`. These commands allow us to run a wide range of jobs, from simple matrix algebra to rather complicated SEM models. Let's move to an example involving optimizing the likelihood of observed data.

1.2.2 Optimization Script

When collecting data to test a specific hypothesis, one of the first things one typically does is to check the basic descriptive statistics, such as the means, variances and covariances/correlations. We can use basic functions in R, i.e., *summary(Data)* - or the alternative *describe(Data)* from the package `psych` - *meanCol(Data)*, *cov(Data)* or *cor(Data)* to perform these operations. We can even get R to provide significance levels for the correlations etc. However, if we want to test specific hypotheses about the data by maximum likelihood (ML), for example, test whether the correlation between two variables is significantly different from zero, we need to compare the likelihood of the data when the correlation is freely estimated with the likelihood of the data when the correlation is fixed to zero. Let's work through a specific example.

Say, we have collected data on two variables **X** and **Y** in 1000 individuals, and R descriptive statistics has shown that the correlation between them is 0.5. For the sake of this example, we used another built-in function in the R package `MASS`, namely `mvrnorm`, to generate multivariate normal data for 1000 individuals with means of 0.0, variances of 1.0 and a correlation (`rs`) of 0.5 between **X** and **Y**. Note that the first argument of `mvrnorm` is the sample size, the second the vector of means, and the third the covariance matrix to be simulated. We save the data in the object `xy` and create a vector of labels for the two selected variables, hence `selVars`, which is used in the `dimnames` statement later on. The *dimnames'* are expected to be a list with rows corresponding to subjects and columns to variables. The rows are typically not labeled thus we use `NULL`, the columns get the labels of the variables from `selVars`. The R functions `summary()` and `cov()` are used to verify that the simulations appear OK.

```
#Simulate Data
require(MASS)
set.seed(200)
rs=.5
xy <- mvrnorm (1000, c(0,0), matrix(c(1,rs,rs,1),2,2))
testData <- xy
selVars <- c('X','Y')
dimnames(testData) <- list(NULL, selVars)
summary(testData)
cov(testData)
```

To evaluate the likelihood of a set of correlated data using SEM, we estimate a saturated model with free means, free variances and a covariance. It is called a saturated model as estimated as many parameters as there are observed statistics (including both means and (co)variances). Let's start with specifying the mean vector. We use the `mxMatrix` command, provide the `type`, here `Full`, the number of rows and columns (`nnow=` and `ncol=`), respectively `1` and `2`, the specification of free/fixed parameters (`free=T/F`), the starting values (`values=`), and a name. Given all the elements of this **1x2** matrix are free, we can use `free=TRUE`. The starting values are provided using a list, i.e. `c(0,0)`. Finally, we are explicit in naming this matrix `expMean`. Thus the matrix command looks like this. Note the soft tabs to improve readability.

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

Next, we need to specify the expected covariance matrix. As this matrix is symmetric, we could estimate it directly as a symmetric matrix. However, to avoid solutions that are not positive definite (and get us into optimization trouble), we will use a Cholesky decomposition. Thus, we specify a lower triangular matrix (matrix with free elements on the diagonal and below the diagonal, and zero's above the diagonal), and multiply it with its transpose to generate a symmetric matrix. We will use a `mxMatrix` command to specify the lower triangular matrix and a `mxAlgebra` command to set up the symmetric matrix. The matrix is a **2x2** free lower matrix with starting values of `0.5` and the name "Chol". We can now refer back to this matrix by its name in the `mxAlgebra` statement. We use a regular multiplication of `Chol` with its transpose `t(Chol)`, and name this as `expCov`. Note that we do not directly estimate the two variances and one covariance, but rather the three elements of a lower triangle of a matrix of the same size. The number of elements in the lower triangle of a matrix are always the same as the number of elements in a symmetric matrix.

```
mxMatrix(
    type="Lower",
    nrow=2,
    ncol=2,
    free=TRUE,
    values=0.5,
    name="Chol"
),
mxAlgebra(
    expression=Chol %*% t(Chol),
    name="expCov"
),
```

Now that we have specified our 'model', we need to supply the data. This is done with the `mxData` command. The first argument includes the actual data, in the type given by the second argument. Type can be a covariance matrix (cov), a correlation matrix (cor), a matrix of cross-products (sscp) or raw data (raw). We will use the latter option and read in the raw data directly from the simulated dataset `testData`.

```
mxData(
    observed=testData,
    type="raw"
),
```

Next, we specify which objective function we wish to use to obtain the likelihood of the data. Given we fit to the raw data, we use the full information maximum likelihood (FIML) objective function `mxFIMLObjective`. Its arguments are the expected covariance matrix (`covariance=`), generated using the `mxMatrix` and `mxAlgebra` commands as `expCov`, and the expected means vector (`means=`), generated using the `mxMatrix` command as `expMeans`, and `dimnames`. The `dimnames` are a type of label that is required to recognize the expected mean vector and expected covariance matrix and match up the rows and columns of the model with those of the data. For a mean vector, the first element is always `NULL` given mean vectors always have one row. The second element of the list should have the labels for the two variables `c('X','Y')` which we have previously assigned to the object `selVars`. For a covariance matrix, both elements are the labels for the two variables, thus `selVars`. Given the key piece of information for the

columns of the mean vector and the rows and columns of the covariance matrix is the list of variables, that is the only element required for `dimnames`.

```
mxFIMLObjective(
    covariance="expCov",
    means="expMean",
    dimnames=selVars)
)
```

All these elements become arguments of the `mxModel` command, seperated by comma's. The first argument can be a name, as in this case "bivCor" or another model (see below). The model is saved in an R object 'bivCorModel'. This `MxModel` object - note the capital `M` in `MxModel` for the resulting R object compared to the lower case `m` in mxModel for the command - becomes the argument of the `mxRun` command, which evaluates the model and provides output - if the model ran successfully - using the following command. Note that we have shrunk every command to one line to provide a better overview of the script here.

```
bivCorModel <- mxModel("bivCor",
    mxMatrix( type="Full", nrow=1, ncol=2, free=TRUE, values=c(0,0), name="expMean" ),
    mxMatrix( type="Lower", nrow=2, ncol=2, free=TRUE, values=0.5, name="Chol" ),
    mxAlgebra( expression=Chol %*% t(Chol), name="expCov", ),
    mxData( observed=testData, type="raw" ),
    mxFIMLObjective( covariance="expCov", means="expMean", dimnames=selVars)
)

bivCorFit <- mxRun(bivCorModel)
```

We can request various final values of the output using `summary(bivCorFit)` or the `mxEval` command. In the following example, the simplest use case of `mxEval` is seen. The name of a matrix or algebra is used, and `mxEval` returns the current value of that matrix or algebra. See the *OpenMx Style Guide* for more advanced uses of the `mxEval` command.

```
EM <- mxEval(expMean, bivCorFit)
EC <- mxEval(expCov, bivCorFit)
LL <- mxEval(objective,bivCorFit)
```

These commands generate the following output:

```
EM
              X            Y
[1,] 0.03211646 -0.004883803

EC
          X         Y
X 1.0092847 0.4813501
Y 0.4813501 0.9935387

LL
         [,1]
[1,] 5415.772
```

Standard lists of data summary, parameter estimates and goodness-of-fit statistics can be obtained with the `summary` command.:

```
> summary(bivCorFit)
         X                    Y
Min.   :-2.942561   Min.   :-3.296159
```

```
1st Qu.:-0.633711   1st Qu.:-0.596177
Median :-0.004139   Median :-0.010538
Mean   : 0.032116   Mean   :-0.004884
3rd Qu.: 0.739236   3rd Qu.: 0.598326
Max.   : 4.173841   Max.   : 4.006771

name  matrix row col parameter estimate error estimate
1 <NA> expMean   1   1        0.032116456     0.02228409
2 <NA> expMean   1   2       -0.004883803     0.02235021
3 <NA>    Chol   1   1        1.004631642     0.01575904
4 <NA>    Chol   2   1        0.479130899     0.02099642
5 <NA>    Chol   2   2        0.874055066     0.01376876

Observed statistics:  2000
Estimated parameters:  5
Degrees of freedom:  1995
-2 log likelihood:  5415.772
Saturated -2 log likelihood:
Chi-Square:
p:
AIC (Mx):  1425.772
BIC (Mx):  -4182.6
adjusted BIC:
RMSEA:  0
```

If we want to test whether the covariance/correlation between **X** and **Y** is significantly different from zero, we could fit a submodel and compare it with the previous saturated model. Given that this model is essentially the same as the original, except for the covariance, we create a new mxModel (named `bivCorModelSub`) with as first argument the old model (named `bivCorModel`). Then we only have to specify the matrix that needs to be changed, in this case the lower triangular matrix becomes essentially a diagonal matrix, obtained by fixing the off-diagonal elements to zero in the `free` and `values` arguments

```
#Test for Covariance=Zero
bivCorModelSub <-mxModel(bivCorModel,
    mxMatrix(
        type="Diag",
        nrow=2,
        ncol=2,
        free=TRUE,
        name="Chol"
    ))
```

Or we can write it more succintly as follows:

```
bivCorModelSub <-mxModel(bivCorModel,
    mxMatrix( type="Diag", nrow=2, ncol=2, free=TRUE, name="Chol" ))

bivCorFitSub <- mxRun(bivCorModelSub)
```

We can output the same information as for the saturated job, namely the expected means and covariance matrix and the likelihood, and then use R to calculate other statistics, such as the Chi-square goodness-of-fit.

```
EMs <- mxEval(expMean, bivCorFitSub)
ECs <- mxEval(expCov, bivCorFitSub)
LLs <- mxEval(objective, bivCorFitSub)
Chi= LLs-LL;
LRT= rbind(LL,LLs,Chi); LRT
```

1.2.3 More in-depth Example

Now that you have seen the basics of OpenMx, let us walk through an example in more detail. We decided to use a twin model example for several reasons. Even though you may not have any background in behavior genetics or genetic epidemiology, the example illustrates a number of features you are likely to encounter at some stage. We will present the example in two ways: (i) path analysis representation, and (ii) matrix algebra representation. Both give exactly the same answer, so you can choose either one or both to get some familiarity with the two approaches.

We will not go into detail about the theory of this model, as that has been done elsewhere (refs). Briefly, twin studies rely on comparing the similarity of identical (monozygotic, MZ) and fraternal (dizygotic, DZ) twins to infer the role of genetic and environmental factors on individual differences. As MZ twins have identical genotypes, similarity between MZ twins is a function of shared genes, and shared environmental factors. Similarity between DZ twins is a function of some shared genes (on average they share 50% of their genes) and shared environmental factors. A basic assumption of the classical twin study is that the MZ and DZ twins share environmental factors to the same extent.

The basic model typically fit to twin data from MZ and DZ twins reared together includes three sources of latent variables: additive genetic factors (**A**), shared environmental influences (**C**) and unique environmental factors (**E**), We can estimate these three sources of variance from the observed variances, the MZ and the DZ covariance. The expected variance is the sum of the three variance components (**A + C + E**). The expected covariance for MZ twins is (**A + C**) and that of DZ twins is (**.5A + C**). As MZ and DZ twins have different expected covariances, we have a multiple group model.

It has been standard in twin modeling to fit models to the raw data, as often data are missing on some co-twins. When using FIML, we also need to specify the expected means. There is no reason to expect that the variances are different for twin 1 and twin 2, neither are the means for twin 1 and twin 2 expected to differ. This can easily be verified by fitting submodels to the saturated model, prior to fitting the **ACE** model.

Let us start by simulating twin data followed by fitting a series of models. The code includes both the twin data simulation and several OpenMx scripts to analyze the data. We will describe each of the parts in turn and include the code for the specific part in the code blocks. Note that a more extensive example is discussed later in *Genetic Epidemiology, Matrix Specification*.

First, we simulate twin data using the `mvrnorm` R function. If the additive genetic factors (**A**) account for 50% of the total variance and the shared environmental factors (**C**) for 30%, thus leaving 20% explained by specific environmental factors (**E**), then the expected MZ twin correlation is `a^2 + c^2` or 0.8 in this case, and the expected DZ twin correlation is 0.55, calculated as `.5*a^2 + c^2`. We simulate 1000 pairs of MZ and DZ twins each with zero means and a correlation matrix according to the values listed above. We run some basic descriptive statistics on the simulated data, using regular R functions.

```
require(OpenMx)
require(psych)
require(MASS)

set.seed(200)
a2<-0.5      #Additive genetic variance component (a squared)
c2<-0.3      #Common environment variance component (c squared)
e2<-0.2      #Specific environment variance component (e squared)
rMZ <- a2+c2
rDZ <- .5*a2+c2
DataMZ <- mvrnorm (1000, c(0,0), matrix(c(1,rMZ,rMZ,1),2,2))
DataDZ <- mvrnorm (1000, c(0,0), matrix(c(1,rDZ,rDZ,1),2,2))

selVars <- c('t1','t2')
    colnames(DataMZ) <- selVars
```

```
    colnames(DataDZ) <- selVars
describe(DataMZ)
describe(DataDZ)
colMeans(DataMZ,na.rm=TRUE)
colMeans(DataDZ,na.rm=TRUE)
cov(DataMZ,use="complete")
cov(DataDZ,use="complete")
```

We typically start with fitting a saturated model, estimating means, variances and covariances separately by order of the twins (twin 1 vs twin 2) and by zygosity (MZ vs DZ pairs), to establish the likelihood of the data. This is essentially similar to the optimization script discussed above, except that we now have two variables (same variable for twin 1 and twin 2) and two groups (MZ and DZ). Thus, the saturated model will have two matrices for the expected means of MZs and DZs, and two for the expected covariances, generated from multiplying a lower triangular matrix with its transpose, one for each group. The raw data are read in using the `mxData` command, and the corresponding objective function `mxFIMLObjective` applied.

```
mxModel("MZ",
    mxMatrix(
        type="Full",
        nrow=1,
        ncol=2,
        free=TRUE,
        values=c(0,0),
        name="expMeanMZ"),
    mxMatrix(
        type="Lower",
        nrow=2,
        ncol=2,
        free=TRUE,
        values=.5,
        name="CholMZ"),
    mxAlgebra(
        expression=CholMZ %*% t(CholMZ),
        name="expCovMZ"),
    mxData(
        observed=DataMZ,
        type="raw"),
    mxFIMLObjective(
        covariance="expCovMZ",
        means="expMeanMZ",
        dimnames=selVars)
    )
```

Note that the `mxModel` statement for the DZ twins is almost identical to that for MZ twins, except for the names of the objects and data. If the arguments to the OpenMx command are given in the default order (see i.e. `?mxMatrix` to open the help/reference page for that command), then it is not necessary to include the name of the argument. Given we skip a few optional arguments, such as `lbound` and `ubound`, the argument name `name=` is included to refer to the right argument. For didactic purposes, we prefer the formatting used for the MZ group, with soft tabs and each argument on a separate line, etc. (see list of formatting rules). However, the experienced user may want to use a more compact form, as the one used for the DZ group.

```
mxModel("DZ",
    mxMatrix("Full", 1, 2, T, c(0,0), name="expMeanDZ"),
    mxMatrix("Lower", 2, 2, T, .5, name="CholDZ"),
    mxAlgebra(CholDZ %*% t(CholDZ), name="expCovDZ"),
```

```
    mxData(DataDZ, "raw"),
    mxFIMLObjective("expCovDZ", "expMeanDZ", selVars))
```

The two models are then combined in a 'super'model which includes them as arguments. Additional arguments are an `mxAlgebra` statement to add the objective funtions/likelihood of the two submodels. To evaluate them simultaneously, we use the `mxAlgebraObjective` with the previous algebra as its argument. The `mxRun` command is used to start optimization.

```
twinSatModel <- mxModel("twinSat",
    mxModel("MZ",
        mxMatrix("Full", 1, 2, T, c(0,0), name="expMeanMZ"),
        mxMatrix("Lower", 2, 2, T, .5, name="CholMZ"),
        mxAlgebra(CholMZ %*% t(CholMZ), name="expCovMZ"),
        mxData(DataMZ, type="raw"),
        mxFIMLObjective("expCovMZ", "expMeanMZ", selVars)),
    mxModel("DZ",
        mxMatrix("Full", 1, 2, T, c(0,0), name="expMeanDZ"),
        mxMatrix("Lower", 2, 2, T, .5, name="CholDZ"),
        mxAlgebra(CholDZ %*% t(CholDZ), name="expCovDZ"),
        mxData(DataDZ, type="raw"),
        mxFIMLObjective("expCovDZ", "expMeanDZ", selVars)),
    mxAlgebra(MZ.objective + DZ.objective, name="minus2loglikelihood"),
    mxAlgebraObjective("minus2loglikelihood")
)
twinSatFit <- mxRun(twinSatModel)
```

It is always helpful/advised to check the model specifications before interpreting the output. Here we are interested in the values for the expected mean vectors and covariance matrices, and the goodness-of-fit statistics, including the likelihood, degrees of freedom, and any other derived indices, such as i.e. Akaike's Information Criterion, which can be obtained by `summary(twinSatFit)`.

```
ExpMeanMZ <- mxEval(MZ.expMeanMZ, twinSatFit)
ExpCovMZ <- mxEval(MZ.expCovMZ, twinSatFit)
ExpMeanDZ <- mxEval(DZ.expMeanDZ, twinSatFit)
ExpCovDZ <- mxEval(DZ.expCovDZ, twinSatFit)
LL_Sat <- mxEval(objective, twinSatFit)
```

Before we move on to fit the ACE model to the same data, we may want to test some of the assumptions of the twin model, i.e. that the means and variances are the same for twin 1 and twin 2, and that they are the same for MZ and DZ twins. This can be done as an omnibus test, or stepwise. Let us start by equating the means for both twins, separately in the two groups. We accomplish this by using the same label (just one label which will be reused by R) for the two free parameters for the means per group. As the majority of the previous script stays the same, we start by copying the old model into a new one. We then include the arguments of the model that require a change.

```
twinSatModelSub1 <- mxModel(twinSatModel, name = "twinSatSub1")
twinSatModelSub1$MZ$expMeanMZ <- mxMatrix("Full", 1, 2, TRUE, 0, "mMZ")
twinSatModelSub1$MZ$expMeanMZ <- mxMatrix("Full", 1, 2, TRUE, 0, "mDZ")
twinSatFitSub1 <- mxRun(twinSatModelSub1)
```

If we want to test if we can equate both means across twin order and zygosity at once, we will end up with the following specification. Note that we use the same label across models for elements that need to be equated.

```
twinSatModelSub2 <- mxModel(twinSatModelSub1, name = "twinSatSub2")
twinSatModelSub2$MZ$expMeanMZ <- mxMatrix("Full", 1, 2, TRUE, 0, "mean")
twinSatModelSub2$DZ$expMeanDZ <- mxMatrix("Full", 1, 2, TRUE, 0, "mean")
twinSatFitSub2 <- mxRun(twinSatModelSub2)
```

We can compare the likelihood of this submodel to that of the fully saturated model or the previous submodel using the results from `mxEval` commands with regular R algebra. A summary of the model parameters, estimates and goodness-of-fit statistics can also be obtained using `summary(twinSatFit)`.

```
LL_Sat <- mxEval(objective, twinSatFit)
LL_Sub1 <- mxEval(objective, twinSatFitSub1)
LRT1 <- LL_Sub1 - LL_Sat
LL_Sub2 <- mxEval(objective, twinSatFitSub1)
LRT2 <- LL_Sub2 - LL_Sat
```

One assumption of the classical twin study is that the variances of twin 1 and twin 2 are not significantly different, nor that they differ between MZ and DZ twins. Although the principle of testing equality of variances across twin order and zygosity are the same as those of testing equality of means, in practice the test of variances are more complicated, because we do not directly estimate the variances, but rather use a Cholesky decomposition. We thus first have to use algebra to extract the expected variances and then use constraints to equate the correct elements of the expected covariance matrices. As this is an introductory chapter, we will leave that treatment for a later example and move to the latent variable model example.

Now, we are ready to specify the ACE model to test which sources of variance significantly contribute to the phenotype and estimate their best value. The structure of this script is going to mimic that of the saturated model. The main difference is that we no longer estimate the variance-covariance matrix directly, but express it as a function of the three sources of variance, **A**, **C** and **E**. As the same sources are used for the MZ and the DZ group, the matrices which will represent them are part of the 'super'model. As these sources are variances, which need to be positive, we typically use a Cholesky decomposition of the standard deviations (and effectively estimate **a** rather then **a^2**, see later for more in depth coverage). Thus, we specify three separate matrices for the three sources of variance using the `mxMatrix` command and 'calculate' the variance components with the `mxAlgebra` command. Note that there are a variety of ways to specify this model, we have picked one that corresponds well to previous Mx code, and has some intuitive appeal.

```
#Specify ACE Model
twinACEModel <- mxModel("twinACE",
    mxModel("ACE",
        # Matrix expMean for expected mean vector for MZ and DZ twins
        mxMatrix( type="Full", nrow=1, ncol=2, free=TRUE, values=20, label="mean",
            name="expMean"),
        # Matrices a, c, and e to store the a, c, and e path coefficients
        mxMatrix( type="Full", nrow=1, ncol=1, free=TRUE, values=.6, label="a11",
            name="a"),
        mxMatrix( type="Full", nrow=1, ncol=1, free=TRUE, values=.6, label="c11",
            name="c"),
        mxMatrix( type="Full", nrow=1, ncol=1, free=TRUE, values=.6, label="e11",
            name="e"),
        # Matrixes A, C, and E to compute A, C, and E variance components
        mxAlgebra( expression=a * t(a), name="A"),
        mxAlgebra( expression=c * t(c), name="C"),
        mxAlgebra( expression=e * t(e), name="E"),
        # Matrix expCovMZ for expected covariance matrix for MZ twins
        mxAlgebra( expression=  rbind(  cbind(A+C+E, A+C),
                                        cbind(A+C,   A+C+E)),
            name="expCovMZ"),
        # Matrix expCovMZ for expected covariance matrix for DZ twins
        mxAlgebra( expression=  rbind(  cbind(A+C+E,   .5%x%A+C),
                                        cbind(.5%x%A+C, A+C+E)),
            name="expCovDZ")
    ),
    mxModel("MZ",
        mxData( observed=DataMZ, type="raw"),
```

```
        mxFIMLObjective( covariance="ACE.expCovMZ", means="ACE.expMean",
            dimnames=selVars)
    ),
    mxModel("DZ",
        mxData( observed=DataDZ, type="raw"),
        mxFIMLObjective( covariance="ACE.expCovDZ", means="ACE.expMean",
            dimnames=selVars)
    ),
    # Algebra to combine objective function of MZ and DZ groups
    mxAlgebra(MZ.objective + DZ.objective, name="minus2loglikelihood"),
    mxAlgebraObjective("minus2loglikelihood")
)
twinACEFit <- mxRun(twinACEModel)
```

Relevant output can be generated with `print` or `summary` statements or specific output can be requested using the `mxEval` command. Typically we would compare this model back to the saturated model to interpret its goodness-of-fit. Parameter estimates are obtained and can easily be standardized. A typical analysis would likely include the following output.

```
LL_ACE <- mxEval(objective, twinACEFit)
LRT_ACE= LL_ACE - LL_Sat

#Retrieve expected mean vector and expected covariance matrices
    MZc <- mxEval(ACE.expCovMZ, twinACEFit)
    DZc <- mxEval(ACE.expCovDZ, twinACEFit)
    M   <- mxEval(ACE.expMean, twinACEFit)
#Retrieve the A, C, and E variance components
    A <- mxEval(ACE.A, twinACEFit)
    C <- mxEval(ACE.C, twinACEFit)
    E <- mxEval(ACE.E, twinACEFit)
#Calculate standardized variance components
    V <- (A+C+E)
    a2 <- A/V
    c2 <- C/V
    e2 <- E/V
#Build and print reporting table with row and column names
    ACEest <- rbind(cbind(A,C,E),cbind(a2,c2,e2))
    ACEest <- data.frame(ACEest, row.names=c("Variance Components","Standardized VC"))
    names(ACEest)<-c("A", "C", "E")
    ACEest; LL_ACE; LRT_ACE
```

Similarly to fitting submodels from the saturated model, we typically fit submodels of the ACE model to test the significance of the sources of variance. One example is testing the significance of shared environmental factors by dropping the free parameter for `c` (fixing it to zero). We call up the previous model and include the new specification for the matrix to be changed, and rerun.

```
    twinAEModel <- mxRename(twinACEModel, "twinAE")
    twinAEModel$ACE.c <- mxMatrix(type="Full", nrow=1, ncol=1,
            free=FALSE, values=.6, label="c11", name="c")

twinAEFit <- mxRun(twinAEModel)
```

We discuss twin analysis examples in more detail in the detailed example code. We hope we have given you some idea of the features of OpenMx.

1.3 Two Model Styles - Two Data Styles

In this first detailed example, we introduce the different styles available to specify models and data, which were briefly discussed in the 'Beginners Guide'. There are currently two supported approaches to specifying models: (i) path specification and (ii) 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; the 'matrix specification' model style relies on matrices and matrix algebra to produce OpenMx code. For either approach you must specify data, and the two supported data styles are (a) summary format, i.e. covariance matrices and possibly means, and (b) raw data format. We will illustrate both, as arguments of functions may differ. Thus, we will describe each example four ways:

- i.a Path Specification - Covariance Matrices (optionally including means)
- i.b Path Specification - Raw Data
- ii.a Matrix Specification - Covariance Matrices (optionally including means)
- ii.b Matrix Specification - Raw Data

Our first example fits a simple model to one variable, estimating its variance and then both the mean and the variance - a so called saturated model. We start with a univariate example, and also work through a more general bivariate example which forms the basis for later examples.

1.3.1 Univariate Saturated Model

The four univariate examples are available here, and you may wish to access them while working through this manual. The last file includes all four example in one.

- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/UnivariateSaturated_PathCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/UnivariateSaturated_PathRaw.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/UnivariateSaturated_MatrixCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/UnivariateSaturated_MatrixRaw.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/UnivariateSaturated.R

The bivariate examples are available in the following files:

- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/BivariateSaturated_PathCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/BivariateSaturated_PathRaw.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/BivariateSaturated_MatrixCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/BivariateSaturated_MatrixRaw.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/BivariateSaturated_MatrixCovCholesky.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/BivariateSaturated_MatrixRawCholesky.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/BivariateSaturated.R

Note that we have additional versions of these matrix-style examples which use a Cholesky decomposition to estimate the expected covariance matrices, which is preferable to directly estimating the symmetric matrices as it avoids the possibility of non positive-definite matrices.

Data

To avoid reading in data from an external file, we simulate a simple dataset directly in R, and use some of its 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)
testData <- as.matrix(x)
selVars <- c("X")
dimnames(testData) <- list(NULL, selVars)
summary(testData)
mean(testData)
var(testData)
```

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 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 `testData` 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. We also define `selVars` here with the names of the variable(s) to be analyzed.

1.3.2 Covariance Matrices and Path-style Input

Model Specification

The model estimates the mean and 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. Below is the path diagram and the complete script:


```
require(OpenMx)

#example 1: Saturated Model with Cov Matrices and Path-Style Input
univSatModel1 <- mxModel("univSat1",
            type="RAM",
    manifestVars= selVars,
    mxPath(
        from=c("X"),
        arrows=2,
        free=T,
        values=1,
        lbound=.01,
        labels="vX"
    ),
    mxData(
```

```
        observed=var(testData),
        type="cov",
        numObs=1000
    )
)
```

Each of of the commands are discussed separately beside excerpts of the OpenMx code. We use the `mxModel` command to specify the model. Its first argument is a name. All arguments are separated by commas.

```
univSatModel1 <- mxModel("univSat1",
```

When using the path specification, it is easiest to work from an existing path diagram. Assuming you are familiar with path analysis (*for those who are not, there are several excellent introductions, see refs*), we have a box for the observed/manifest variable *x*, specified with the `manifestVars` argument, and one double headed arrow on the box to represent its variance, specified with the `mxPath` command. 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.

```
        type="RAM",
manifestVars=selVars,

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

We specify which data the model is fitted to with the `mxData` command. Its first argument, `observed=`, reads in the data from an R matrix or data.frame, with the `type=` given in the second argument. Given we read a covariance matrix here, 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.

```
mxData(
    observed=var(testData),
    type="cov",
    numObs=1000
))
```

With the path specification, the 'RAM' objective function is used by default, as indicated by the `type` argument. Internally, OpenMx translates the paths into RAM notation in the form of the matrices **A**, **S**, and **F** [see refs].

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 object as its argument.

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

There are 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` name. The `expression` can be a matrix or algebra name defined in the model, new calculations using any of these matrices/algebras, the objective function, etc. We can then use any regular R function to generate derived fit statistics, some of which will be 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.

```
EC1 <- mxEval(S, univSatFit1)    #univSatFit1[['S']]@values
LL1 <- mxEval(objective, univSatFit1)
SL1 <- univSatFit1@output$other$Saturated
Chi1 <- LL1-SL1
```

The output of these objects like as follows:

```
> EC1
             [,1]
    [1,] 1.062112
> LL1
             [,1]
    [1,] 1.060259
> SL1
    [1] 1.060259
> Chi1
                  [,1]
    [1,] 2.220446e-16
```

In addition to providing a covariance matrix as input data, we could add a means vector. As this requires a few minor changes, lets highlight those. We have one additional `mxPath` command for the means. In the path diagram, the means are specified by a triangle which as a fixed value of one, reflected in the `from="one"` argument, with the `to=` argument referring to the variable which mean is estimated. Note that paths for means are always single headed.

```
univSatModel1m <- mxModel(univSatModel1, name = "univSat1m",
    mxPath(
        from="one",
        to="X",
        arrows=1,
        free=T,
        values=0,
        labels="mX"
    ),
```

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 `mean` command.

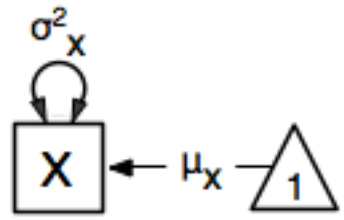
```
    mxData(
        observed=var(testData),
        type="cov",
        numObs=1000,
        means=mean(testData)
    )
)
```

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 referred to in the output in a similar way as the expected variance before.

```
univSatFit1m <- mxRun(univSatModel1m)
EM1m <- mxEval(M, univSatFit1m)
```

1.3.3 Raw Data and Path-style Input

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 path diagram for this model, now including means (path from triangle of value 1) is as follows:



The only change required is in the `mxData` command, which now takes either an R matrix or a data.frame with the observed data as first argument, and the `type="raw"` as the second argument.

```
mxData(
    observed=testData,
    type="raw"
)
```

A nice feature of OpenMx is that an existing model can be easily modified. So `univSatModel1` can be modified as follows:

```
univRawModel1 <- mxModel(univSatModel1,
    mxData(
        observed=testData,
        type="raw"
    )
)
```

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

```
univRawFit1 <- mxRun(univSatModel1)
```

Note that the output now includes the expected means, as well as the expected covariance matrix and -2 x log-likelihood of the data.:

```
> EM2
               [,1]
    [1,] 0.01680498
> EC2
             [,1]
    [1,] 1.061049
> LL2
             [,1]
    [1,] 2897.135
```

1.3.4 Covariance Matrices and Matrix-style Input

The next example replicates these models using matrix-style coding. The code to specify the model includes four commands, (i) `mxModel`, (ii) `mxMatrix`, (iii) `mxData` and (iv) `mxMLObjective`.

Starting with the model fitted to the summary covariance matrix, we need to 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 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 following examples. The `mxData` is identical to that used in path stlye models. A different objective function is used, however, namely the `mxMLObjective` command which takes two arguments, `covariance` to hold the expected covariance matrix (which we specified above using `mxMatrix` as `expCov`), and `dimnames` which allow the mapping of the observed data to the expected covariance matrix, i.e. the model.

```
univSatModel3 <- mxModel("univSat3",
    mxMatrix(
        type="Symm",
        nrow=1,
        ncol=1,
        free=T,
        values=1,
        name="expCov"
    ),
    mxData(
        observed=var(testData),
        type="cov",
        numObs=1000
    ),
    mxMLObjective(
        covariance="expCov",
        dimnames=selVars
    )
)

univSatFit3 <- mxRun(univSatModel3)
```

A means vector can also be added 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 `1` row and column, is assigned `free=T` with start value `0`, and the name `expMean`. The second change is an additional argument `mean` to the `mxMLObjective` function for the expected mean, here `expMean`.

```
mxMatrix(
    type="Full",
    nrow=1,
    ncol=1,
    free=TRUE,
    values=0,
    name="expMean"
)
mxData(
    observed=var(testData),
    type="cov",
    numObs=1000,
    means=mean(testData)
```

```
)
mxMLObjective(
    covariance="expCov",
    means="expMean",
    dimnames=selVars
)
```

1.3.5 Raw Data and Matrix-style Input

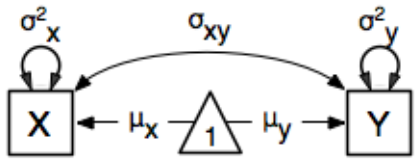
Finally, if we want to use the matrix specification with raw data, we specify matrices for the means and covariances using `mxMatrix()`. The `mxData` command now, however takes a matrix (or data.frame) of raw data and the `mxFIMLObjective` function replaces `mxMLObjective` to evaluate the likelihood of the data using FIML (Full Information Maximum Likelihood). This function takes three arguments: the expected covariance matrix `covariance`; the expected mean vector, `means`; and a third for the `dimnames`.

```
univSatModel4 <- mxModel("univSat4",
    mxMatrix(
        type="Symm",
        nrow=1,
        ncol=1,
        free=T,
        values=1,
        name="expCov"
    ),
    mxMatrix(
        type="Full",
        nrow=1,
        ncol=1,
        free=T,
        values=0,
        name="expMean"
    ),
    mxData(
        observed=testData,
        type="raw"
    ),
    mxFIMLObjective(
        covariance="expCov",
        means="expMean",
        dimnames=selVars
    )
)
```

Note that the output generated for the paths and matrices specification are completely equivalent.

1.3.6 Bivariate Saturated Model

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 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.

Data

The data used for the example were generated using the multivariate normal function (`mvrnorm` in the R package MASS). We have simulated data on two variables named 'X' and 'Y' with means of zero, variances of one and a covariance of 0.5 using the following R code, and saved is as `testData`. Note that we can now use the R function `cov` to generate the observed covariance matrix.

```
#Simulate Data
require(MASS)
set.seed(200)
rs=.5
xy <- mvrnorm (1000, c(0,0), matrix(c(1,rs,rs,1),2,2))
testData <- xy
selVars <- c('X','Y')
dimnames(testData) <- list(NULL, selVars)
summary(testData)
cov(testData)
```

Model Specification

The `mxPath` commands look as follows. The first one specifies two-headed arrows from **X** and **Y** to themselves. 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.

```
mxPath(
    from=c("X", "Y"),
    arrows=2,
    free=T,
    values=1,
    lbound=.01,
    labels=c("varX","varY")
)
mxPath(
    from="X",
    to="Y",
    arrows=2,
    free=T,
    values=.2,
    lbound=.01,
```

```
    labels="covXY"
)
```

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

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

Changes required for fitting to raw data are to the `mxData` command to read in the data directly with `type=raw`.

Using matrices instead of paths, our `mxMatrix` command for the expected covariance matrix now specifies a **2x2** 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

```
mxMatrix(
    type="Symm",
    nrow=2,
    ncol=2,
    free=T,
    values=c(1,0.5,1),
    name="expCov"
)
```

The optional expected means command specifies a **1x2** row vector with two free parameters, each given a 0 start value.

```
mxMatrix(
    type="Full",
    nrow=1,
    ncol=2,
    free=T,
    values=c(0,0),
    name="expMean"
)
```

Combining these two `mxMatrix` commands with the raw data, specified in the `mxData` command and the `mxFIMLObjective` command with the appropriate arguments is all that is needed to fit a saturated bivariate model. 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 specify `type="Lower"`. We then use an `mxAlgebra` command to multiply this matrix, named `Chol` with its transpose (R function *t()*).

```
mxMatrix(
    type="Lower",
    nrow=2,
    ncol=2,
    free=T,
    values=.5,
```

```
    name="Chol"
)
mxAlgebra(
    Chol %*% t(Chol),
    name="expCov",
)
```

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.

1.4 OpenMx Style Guide

The goal of the OpenMx Style Guide is to make our OpenMx scripts easier to read, share, and verify.

1.4.1 The '$' 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:

```
model <- 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',
        mxAlgebraObjective('topmodel1.A')
    )
)
```

Accessing 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$A                     model$submodel2
model$B                     model$submodel2.objective
model$submodel1             model$undersub1
model$submodel1.C           model$undersub1.data
```

The named entities of the model are displayed in one of three modes. In the first mode, all of the submodels contained within the parent model are accessed by using their unique model name (`submodel1`, `submodel2`, and `undersub1`). In the second mode, all of the named entities contained within the parent model are displayed by their names (`A` and `B`). In the third mode, all of the named entities contained by the submodels are displayed in the `modelname.entityname` format (`submodel1.C`, `submodel2.objective`, and `undersub1.data`). The

three models will become even more important in the next section, *Modifying Elements*, so make sure you are comfortable with them before moving on.

Modifying 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 models for accessing elements.

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

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

In the second mode, the named entities of the parent model are modified using their names:

```
# 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:

```
# 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.

1.4.2 mxEval()

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.

```
 model <- mxModel('topmodel',
     mxMatrix('Full', 1, 1, values=1, free=TRUE, labels='p1', name='A'),
     mxModel('submodel1',
         mxMatrix('Full', 1, 1, values=2, free=FALSE, labels='p2', name='B')
     ),
     mxModel('submodel2',
         mxMatrix('Full', 1, 1, values=3, name = 'B')
     )
 )

modelOut <- mxRun(model)
```

```
mxEval(A + submodel1.B + submodel2.B + p1 + p2, model)        # initial values
mxEval(A + submodel1.B + submodel2.B + p1 + p2, modelOut)     # final values
```

To reinforce an earlier point, it is not necessary to restrict the expression to only valid MxAlgebra expressions. In the following example, we use the `harmonic.mean` function from the psych package.

```
library(psych)
nVars <- 3
mxEval(nVars * harmonic.mean(c(A, submodel1.B, submodel2.B)), model)
```

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 execution.

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/repoview/1/trunk/demo/SimpleRegression_PathCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/SimpleRegression_PathRaw.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/MultipleRegression_PathCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/MultipleRegression_PathRaw.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/MultivariateRegression_PathCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/MultivariateRegression_PathRaw.R

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

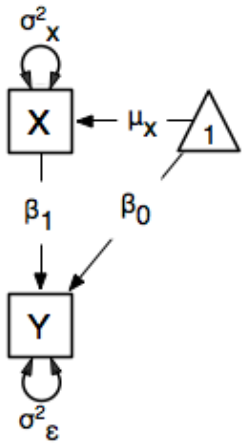
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/SimpleRegression_MatrixCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/SimpleRegression_MatrixRaw.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/MultipleRegression_MatrixCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/MultipleRegression_MatrixRaw.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/MultivariateRegression_MatrixCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/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)

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)

uniRegModel <- mxModel("Simple Regression Path Specification",
    type="RAM",
    mxData(
        observed=SimpleDataRaw,
        type="raw"
    ),
    manifestVars=c("x", "y"),
    # variance paths
    mxPath(
        from=c("x", "y"),
        arrows=2,
        free=TRUE,
        values = c(1, 1),
        labels=c("varx", "residual")
    ),
    # regression weights
    mxPath(
        from="x",
        to="y",
        arrows=1,
        free=TRUE,
        values=1,
        labels="beta1"
    ),
    # means and intercepts
    mxPath(
        from="one",
        to=c("x", "y"),
        arrows=1,
        free=TRUE,
        values=c(1, 1),
        labels=c("meanx", "beta0")
    )
) # close model
```

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.

```
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:

```
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
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
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
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)
```

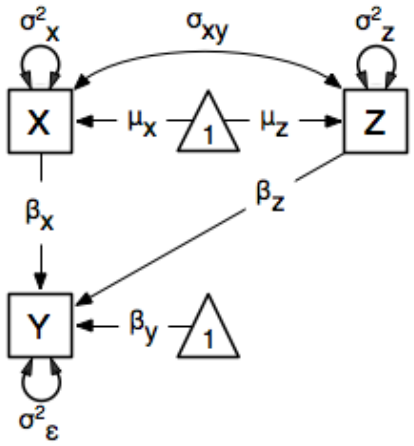
The `output` slot contains a great deal of information, including parameter estimates and information about the matrix operations underlying our model. A more parsimonious report on the results of our model can be viewed using the `summary` function, as shown here.

```
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)

multiRegModel <- mxModel("Multiple Regression Path Specification",
    type="RAM",
    mxData(
        observed=MultipleDataRaw,
        type="raw"
    ),
    manifestVars=c("x", "y", "z"),
    # variance paths
    mxPath(
        from=c("x", "y", "z"),
        arrows=2,
        free=TRUE,
        values = c(1, 1, 1),
        labels=c("varx", "residual", "varz")
    ),
    # covariance of x and z
    mxPath(
        from="x",
        to="z",
        arrows=2,
        free=TRUE,
        values=0.5,
        labels="covxz"
    ),
```

```
    # regression weights
    mxPath(
        from=c("x","z"),
        to="y",
        arrows=1,
        free=TRUE,
        values=1,
        labels=c("betax","betaz")
    ),
    # means and intercepts
    mxPath(
        from="one",
        to=c("x", "y", "z"),
        arrows=1,
        free=TRUE,
        values=c(1, 1),
        labels=c("meanx", "beta0", "meanz")
    )
) # close model

multiRegFit <- mxRun(multiRegModel)

multiRegFit@output
summary(multiRegFit)
```

The first bit of our code should look very familiar. `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. The `type="RAM"` argument is identical. The `mxData` function references our multiple regression data, which contains one more variable than our simple regression data. 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
mxPath(
    from="x",
    to="z",
    arrows=2,
    free=TRUE,
    values=0.5,
    labels="covxz"
)
```

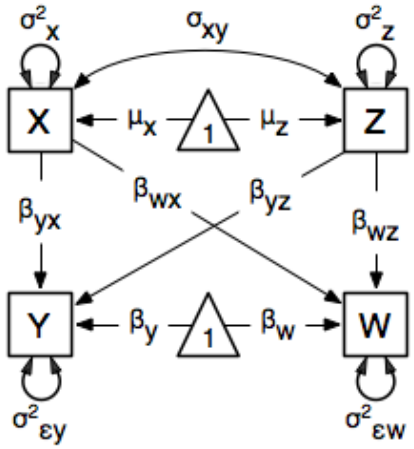
The third and fourth `mxPath` functions mirror the second and third `mxPath` functions from our simple regression, defining the regressions of y on both x and z as well as the means and intercepts of our model.

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.

```
multivariateRegModel <- mxModel("MultiVariate Regression Path Specification",
    type="RAM",
    mxData(
        observed=myRegDataRaw,
```

```
        type="raw"
    ),
    manifestVars=c("w", "x", "y", "z"),
    # variance paths
    mxPath(
        from=c("w", "x", "y", "z"),
        arrows=2,
        free=TRUE,
        values = c(1, 1, 1, 1),
        labels=c("residualw", "varx", "residualy", "varz")
    ),
    # covariance of x and z
    mxPath(
        from="x",
        to="z",
        arrows=2,
        free=TRUE,
        values=0.5,
        labels="covxz"
    ),
    # regression weights for y
    mxPath(
        from=c("x","z"),
        to="y",
        arrows=1,
        free=TRUE,
        values=1,
        labels=c("betayx","betayz")
    ),
    # regression weights for w
    mxPath(
        from=c("x","z"),
        to="w",
        arrows=1,
        free=TRUE,
        values=1,
        labels=c("betawx","betawz")
    ),
    # means and intercepts
    mxPath(
        from="one",
        to=c("w", "x", "y", "z"),
        arrows=1,
        free=TRUE,
        values=c(1, 1, 1 , 1),
        labels=c("betaw", "meanx", "betay", "meanz")
    )
) # close model

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/repoview/1/trunk/demo/OneFactorModel_PathCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/OneFactorModel_PathRaw.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/TwoFactorModel_PathCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/TwoFactorModel_PathRaw.R

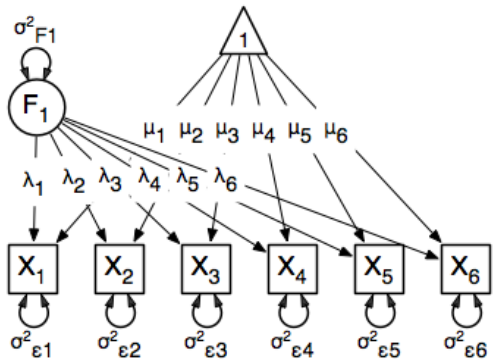
Parallel versions of this example, using matrix specification of models rather than paths, can be found here:

- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/OneFactorModel_MatrixCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/OneFactorModel_MatrixRaw.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/TwoFactorModel_MatrixCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/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 (6 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)

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)

oneFactorModel <- mxModel("Common Factor Model Path Specification",
    type="RAM",
    mxData(
        observed=oneFactorRaw,
        type="raw"
    ),
    manifestVars=c("x1","x2","x3","x4","x5","x6"),
    latentVars="F1",
    # residual variances
    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
    mxPath(
        from="F1",
        arrows=2,
        free=TRUE,
        values=1,
        labels ="varF1"
    ),
    # factor loadings
    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
    mxPath(
        from="one",
        to=c("x1","x2","x3","x4","x5","x6","F1"),
        arrows=1,
        free=c(TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,FALSE),
        values=c(1,1,1,1,1,1,0),
        labels =c("meanx1","meanx2","meanx3","meanx4","meanx5","meanx6",NA)
    )
) # close model
```

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:

```
mxData(
    observed=oneFactorRaw,
    type="raw"
)
```

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

```
oneFactorModel<-mxModel("Common Factor Model Path Specification",
    type="RAM",
    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
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
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
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
mxPath(
    from="one",
    to=c("x1","x2","x3","x4","x5","x6","F1"),
    arrows=1,
    free=c(TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,FALSE),
    values=c(1,1,1,1,1,1,0),
    labels =c("meanx1","meanx2","meanx3","meanx4","meanx5","meanx6",NA)
))
```

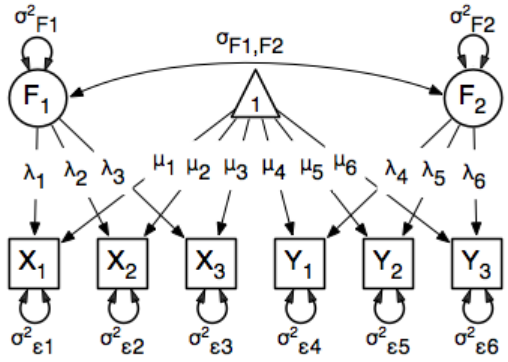
The model can now be run using the `mxRun` function, and the output of the model can be accessed from the `output` slot of the resulting model. A summary of the output can be reached using `summary()`.

```
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 (6 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.

```
twofactorModel<-mxModel("Two Factor Model Path Specification",
    type="RAM",
    mxData(
        observed=twoFactorRaw,
        type="raw"
    ),
    manifestVars=c("x1","x2","x3","y1","y2","y3"),
    latentVars=c("F1","F2"),
    # residual variances
    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
    mxPath(
        from=c("F1","F2"),
        arrows=2,
        all=TRUE,
        free=TRUE,
        values=c(1, .5,.5, 1),
        labels=c("varF1","cov","cov","varF2")
    ),
    # factor loadings for x variables
    mxPath(
        from="F1",
        to=c("x1","x2","x3"),
        arrows=1,
        free=c(FALSE,TRUE,TRUE),
        values=c(1,1,1),
        labels=c("l1","l2","l3")
    ),
    #factor loadings for y variables
    mxPath(
        from="F2",
        to=c("y1","y2","y3"),
        arrows=1,
        free=c(FALSE,TRUE,TRUE),
        values=c(1,1,1),
        labels=c("l4","l5","l6")
    ),
    #means
    mxPath(
        from="one",
        to=c("x1","x2","x3","y1","y2","y3","F1","F2"),
        arrows=1,
        free=c(TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,FALSE,FALSE),
        values=c(1,1,1,1,1,1,0,0),
        labels=c("meanx1","meanx2","meanx3","meany1","meany2","meany3",NA,NA)
    )
)
```

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
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
mxPath(
    from="one",
    to=c("x1","x2","x3","y1","y2","y3","F1","F2"),
    arrows=1,
    free=c(TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,FALSE,FALSE),
    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 `all` argument to `TRUE`, which creates all possible 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 possible paths from and to our two latent variables. This results in four paths: from F1 to F1 (the variance of F1), from F1 to F2 (the covariance of the latent variables), from F2 to F1 (again, the covariance), and from F2 to F2 (the variance of F2). As the covariance is both the second and third path on this list, the second and third elements of both the `values` argument (.5) and the `labels` argument (`"cov"`) are the same.

```
# latent variances and covariance
mxPath(
     from=c("F1","F2"),
    arrows=2,
    all=TRUE,
    free=TRUE,
    values=c(1, .5,.5, 1),
    labels=c("varF1","cov","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
mxPath(
    from="F1",
    to=c("x1","x2","x3"),
    arrows=1,
    free=c(FALSE,TRUE,TRUE),
    values=c(1,1,1),
    labels=c("l1","l2","l3")
)
#factor loadings for y variables
```

```
mxPath(
    from="F2",
    to=c("y1","y2","y3"),
    arrows=1,
    free=c(FALSE,TRUE,TRUE),
    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 raw and covariance data. These examples can be found in the following files:

- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/LatentGrowthCurveModel_PathCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/LatentGrowthCurveModel_PathRaw.R

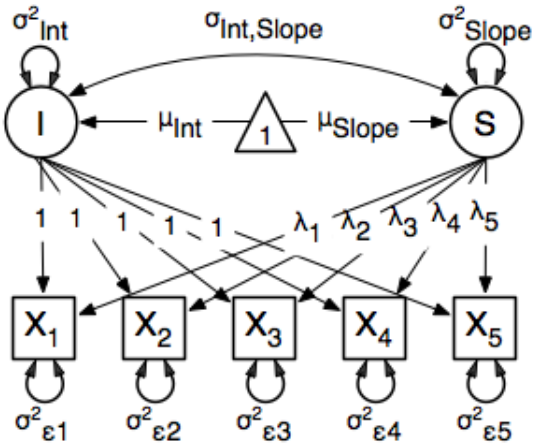
Parallel versions of this example, using matrix specification of models rather than paths, can be found here:

- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/LatentGrowthCurveModel_MatrixCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/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 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 `"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)
```

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)

growthCurveModel <- mxModel("Linear Growth Curve Model Path Specification",
    type="RAM",
    mxData(
        myLongitudinalData,
        type="raw"
    ),
    manifestVars=c("x1","x2","x3","x4","x5"),
    latentVars=c("intercept","slope"),
    # residual variances
    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
    mxPath(
        from=c("intercept","slope"),
        arrows=2,
        all=TRUE,
        free=TRUE,
        values=c(1, 1, 1, 1),
        labels=c("vari", "cov", "cov", "vars")
    ),
    # intercept loadings
    mxPath(
        from="intercept",
        to=c("x1","x2","x3","x4","x5"),
        arrows=1,
        free=FALSE,
        values=c(1, 1, 1, 1, 1)
    ),
    # slope loadings
    mxPath(
        from="slope",
        to=c("x1","x2","x3","x4","x5"),
        arrows=1,
        free=FALSE,
        values=c(0, 1, 2, 3, 4)
    ),
    # manifest means
    mxPath(
        from="one",
        to=c("x1", "x2", "x3", "x4", "x5"),
        arrows=1,
        free=FALSE,
```

```
        values=c(0, 0, 0, 0, 0)
    ),
    # latent means
    mxPath(
        from="one",
        to=c("intercept", "slope"),
        arrows=1,
        free=TRUE,
        values=c(1, 1),
        labels=c("meani", "means")
    )
) # close model
```

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.

```
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
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 `all` argument to `TRUE`, which creates all possible 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 possible paths from and to our two latent variables. This results in four paths: from intercept to intercept (the variance of the interecpts), from intercept to slope (the covariance of the latent variables), from slope to intercept (again, the covariance), and from slope to slope (the variance of the slopes). As the covariance is both the second and third path on this list, the second and third elements of both the `values` argument (.5) and the `labels` argument (`"cov"`) are the same.

```
# latent variances and covariance
mxPath(
    from=c("intercept","slope"),
```

```
    arrows=2,
    all=TRUE,
    free=TRUE,
    values=c(1, 1, 1, 1),
    labels=c("vari", "cov", "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
mxPath(
    from="intercept",
    to=c("x1","x2","x3","x4","x5"),
    arrows=1,
    free=FALSE,
    values=c(1, 1, 1, 1, 1)
)
# slope loadings
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
mxPath(
    from="one",
    to=c("x1", "x2", "x3", "x4", "x5"),
    arrows=1,
    free=FALSE,
    values=c(0, 0, 0, 0, 0)
)
# latent means
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/repoview/1/trunk/demo/BivariateHeterogeneity_PathRaw.R

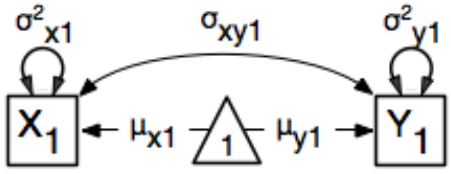
A parallel version of this example, using matrix specification of models rather than paths, can be found here:

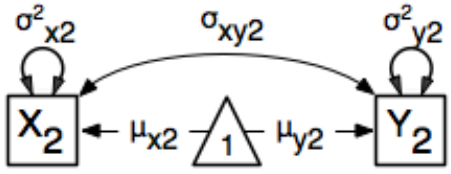
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/BivariateHeterogeneity_MatrixRaw.R

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)
rs=.5
xy1 <- mvrnorm (1000, c(0,0), matrix(c(1,rs,rs,1),2,2))
set.seed(200)
#group 2
rs=.4
xy2 <- mvrnorm (1000, c(0,0), matrix(c(1,rs,rs,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 look 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"),
    mxAlgebra(group1.objective + group2.objective, name="h12"),
    mxAlgebraObjective("h12")
)
```

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.

```
#Fit Heterogeneity Model
bivHetModel <- mxModel("bivHet",
    mxModel("group1",
        manifestVars= selVars,
        # variances
        mxPath(
            from=c("X", "Y"),
            arrows=2,
            free=T,
            values=1,
            lbound=.01,
            labels=c("vX1","vY1")
        ),
```

```
        # covariance
        mxPath(
            from="X",
            to="Y",
            arrows=2,
            free=T,
            values=.2,
            lbound=.01,
            labels="cXY1"
        ),
        # means
            mxPath(
            from="one",
            to=c("X", "Y"),
            arrows=1,
            free=T,
            values=0,
            labels=c("mX1", "mY1")
        ),
        mxData(
            observed=xy1,
            type="raw",
        ),
        type="RAM"
    ),
    mxModel("group2",
        manifestVars= selVars,
        # variances
        mxPath(
            from=c("X", "Y"),
            arrows=2,
            free=T,
            values=1,
            lbound=.01,
            labels=c("vX2","vY2")
        ),
        # covariance
        mxPath(
            from="X",
            to="Y",
            arrows=2,
            free=T,
            values=.2,
            lbound=.01,
            labels="cXY2"
        ),
        # means
        mxPath(
            from="one",
            to=c("X", "Y"),
            arrows=1,
            free=T,
            values=0,
            labels=c("mX2", "mY2")
        ),
        mxData(
            observed=xy2,
            type="raw",
```

```
        ),
        type="RAM"
    ),
```

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"

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

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

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

To evaluate both models together, we use an `mxAlgebra` command that adds up the values of the objective functions of the two groups, and assigns a `name`. The objective function to be used here is the `mxAlgebraObjective` which uses as its argument the sum of the function values of the two groups, referred to by the name of the previously defined `mxAlgebra` object `h12`.

```
    mxAlgebra(
        group1.objective + group2.objective,
        name="h12"
    ),
    mxAlgebraObjective("h12")
)
```

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. 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.means` to refer to the objects in the correct model.

```
EM1Het <- mxEval(group1.means, bivHetFit)
EM2Het <- mxEval(group2.means, bivHetFit)
EC1Het <- mxEval(group1.covariance, bivHetFit)
EC2Het <- mxEval(group2.covariance, bivHetFit)
LLHet <- mxEval(objective, bivHetFit)
```

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 <- mxEval(group1.means, bivHomFit)
    EM2Hom <- mxEval(group2.means, bivHomFit)
    EC1Hom <- mxEval(group1.covariance, bivHomFit)
    EC2Hom <- mxEval(group2.covariance, bivHomFit)
    LLHom <- mxEval(objective, bivHomFit)
```

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 is probably 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:

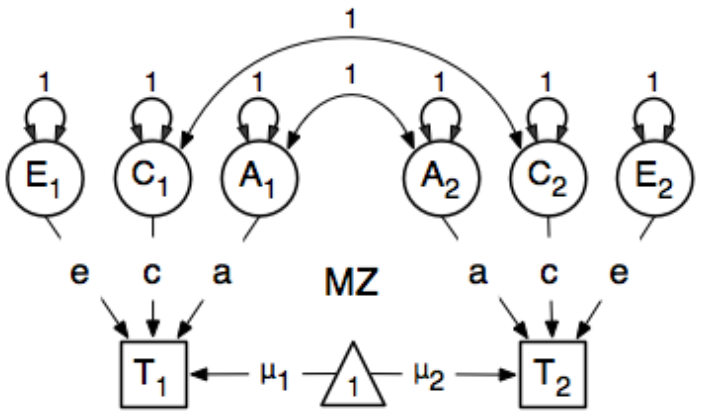
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/UnivariateTwinAnalysis_PathRaw.R

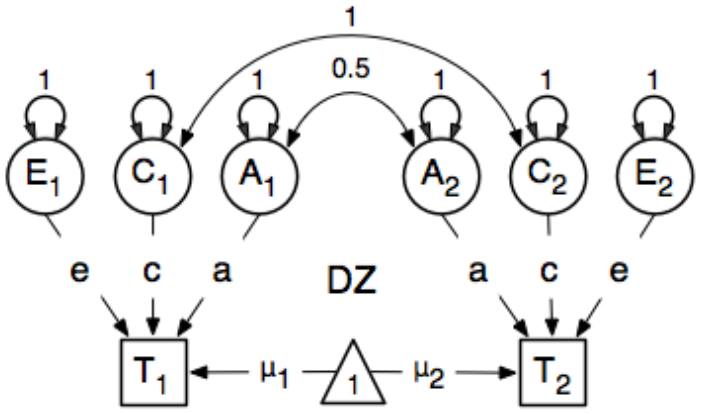
A parallel version of this example, using matrix specification of models rather than paths, can be found here:

- http://openmx.psyc.virginia.edu/repoview/1/trunk/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 '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. We also define the objects `selVars` for the variables selected for analysis, and `aceVars` for the latent variables to simplify the OpenMx code.

```
require(OpenMx)

#Prepare Data
data(myTwinData)
twinVars <- c(  'fam','age','zyg','part',
                'wt1','wt2','ht1','ht2','htwt1','htwt2','bmi1','bmi2')
summary(myTwinData)
selVars <- c('bmi1','bmi2')
aceVars <- c("A1","C1","E1","A2","C2","E2")
mzData <- as.matrix(subset(myTwinData, zyg==1, c(bmi1,bmi2)))
dzData <- as.matrix(subset(myTwinData, zyg==3, c(bmi1,bmi2)))
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. As 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), 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 in yet another model, called `ACEModel` which will then be shared with the two submodels for each of the twin types, defined in separate `mxModel` commands. The latter two `MxModel` objects (`mzModel` and `dzModel`) are arguments of the overall `twinACE` model, and will be saved together in the R object `twinACEModel` and thus be run together.

```
#Fit ACE Model with RawData and Path-Style Input
ACEModel <- mxModel("ACE",
    type="RAM",
    manifestVars=selVars,
    latentVars=aceVars,
    # variances of latent variables
    mxPath(
        from=aceVars,
        arrows=2,
        free=FALSE,
        values=1
    ),
    # means of latent variables
    mxPath(
        from="one",
        to=aceVars,
        arrows=1,
        free=FALSE,
        values=0
    ),
    # means of observed variables
    mxPath(
        from="one",
        to=selVars,
        arrows=1,
        free=TRUE,
        values=20,
        labels="mean",
    ),
    # path coefficients for twin 1
    mxPath(
```

```
        from=c("A1","C1","E1"),
        to="bmi1",
        arrows=1,
        free=TRUE,
        values=.6,
        label=c("a","c","e")
    ),
    # path coefficients for twin 2
    mxPath(
        from=c("A2","C2","E2"),
        to="bmi2",
        arrows=1,
        free=TRUE,
        values=.6,
        label=c("a","c","e")
    ),
    # covariance between C1 & C2
    mxPath(
        from="C1",
        to="C2",
        arrows=2,
        free=FALSE,
        values=1
    )
)
mzModel <- mxModel(ACEModel, name="MZ",
    # covariance between A1 & A2
    mxPath(
        from="A1",
        to="A2",
        arrows=2,
        free=FALSE,
        values=1
    ),
    mxData(
        observed=mzData,
        type="raw"
    )
)
dzModel <- mxModel(ACEModel, name="DZ",
# covariance between A1 & A2
    mxPath(
        from="A1",
        to="A2",
        arrows=2,
        free=FALSE,
        values=.5
    ),
    mxData(
        observed=dzData,
        type="raw"
    )
)
twinACEModel <- mxModel("twinACE", mzModel, dzModel,
    mxAlgebra(
        expression=MZ.objective + DZ.objective,
        name="minus2loglikelihood"
    ),
```

```
    mxAlgebraObjective("minus2loglikelihood")
)
```

Now we will discuss the script line by line. For further details on RAM, see ref. Note that we left the comma's at the end of the lines which are necessary when all the arguments are combined prior to running the model. 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"`. 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.

```
mxModel("ACE",
        type="RAM",
        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
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
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
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 0.6 and appropriate labels.

```
# path coefficients for twin 1
mxPath(
    from=c("A1","C1","E1"),
    to="bmi1",
    arrows=1,
    free=TRUE,
    values=0.6,
    label=c("a","c","e")
),
# path coefficients for twin 2
mxPath(
    from=c("A2","C2","E2"),
    to="bmi2",
    arrows=1,
    free=TRUE,
    values=0.6,
    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
mxPath(
    from="C1",
    to="C2",
    arrows=2,
    free=FALSE,
    values=1
))
```

We add the paths that are specific to the MZ group or the DZ group into the respective models, `mzModel` and `dzModel`, which are combined in `twinACEModel`. So we have two `mxModel` statements following the `ACEModel` model statement. Each of the two models have access to all the paths already defined given `ACEModel` is the first argument of `mzModel` and `dzModel`. In the MZ model we add the path for the correlation between **A1** and **A2** which is fixed to one. That concludes the specification of the model for the MZ's, thus we move to the `mxData` command that calls up the data.frame with the MZ raw data, `mzData`, with the type specified explicitly. We also give the model a name, `MZ`, to refer back to it later when we need to add the objective functions.

```
mzModel <- mxModel(ACEModel, name="MZ",
    # covariance between A1 & A2 in MZ's
    mxPath(
        from="A1",
        to="A2",
        arrows=2,
        free=FALSE,
        values=1
    ),
    mxData(
        observed=mzData,
        type="raw"
    )
)
```

The `mxModel` command for the DZ group is very similar, except that the the correlation between **A1** and **A2** is fixed to 0.5 and the DZ data, `dzData` are read in, and the model is named `DZ`. Note that OpenMx can handle constants in

algebra.

```
dzModel <- mxModel(ACEModel, name="DZ",
    # covariance between A1 & A2 in DZ's
    mxPath(
        from="A1",
        to="A2",
        arrows=2,
        free=FALSE,
        values=.5
    ),
    mxData(
        observed=dzData,
        type="raw"
    )
)
```

Finally, both models need to be evaluated simultaneously. We specify a new `mxModel` which has the `mzModel` and `dzModel` as its arguments. We then generate the sum of the objective functions for the two groups, using `mxAlgebra`, and use the result (`minus2loglikelihood`) as argument of the `mxAlgebraObjective` command.

```
twinACEModel <- mxModel("twinACE", mzModel, dzModel,
    mxAlgebra(
        expression=MZ.objective + DZ.objective,
        name="minus2loglikelihood"
    ),
    mxAlgebraObjective("minus2loglikelihood")
)
```

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.

```
#Run ACE model
twinACEFit <- mxRun(twinACEModel)
```

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/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 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.

```
MZc <- mxEval(MZ.covariance, twinACEFit)     # expected covariance matrix for MZ's
DZc <- mxEval(DZ.covariance, twinACEFit)     # expected covariance matrix for DZ's
M <- mxEval(MZ.means, twinACEFit)     # expected mean
A <- mxEval(a*a, twinACEFit)     # additive genetic variance, a^2
C <- mxEval(c*c, twinACEFit)     # shared environmental variance, c^2
E <- mxEval(e*e, twinACEFit)     # unique environmental variance, e^2
V <- (A+C+E)     # total variance
a2 <- A/V        # standardized A
```

```
c2 <- C/V         # standardized C
e2 <- E/V         # standardized E
ACEest <- rbind(cbind(A,C,E),cbind(a2,c2,e2))     # table of estimates
LL_ACE <- mxEval(objective, twinACEFit)         # likelihood of ACE model
```

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 two `mxPath` commands. First, we define a new model 'AEModel' with 'ACEModel' as its first argument. `ACEModel` included the common parts of the model, necessary for both MZ and DZ group. Next 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 respecify the mzModel and the dzModel, so that they are now built with the changed paths from the common `AEModel`. We can run this model in the same way as before, by combining the objective functions of the two groups and generate similar summaries of the results.

```
#Run AE model
AEModel <- mxModel(ACEModel, name="twinAE",
    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")
    ),
    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")
    )
)
mzModel <- mxModel(AEModel, name="MZ",
    mxPath(
        from="A1",
        to="A2",
        arrows=2,
        free=FALSE,
        values=1
    ),
    mxData(
        observed=mzData,
        type="raw"
    )
)
dzModel <- mxModel(AEModel, name="DZ",
    mxPath(
        from="A1",
        to="A2",
        arrows=2,
        free=FALSE,
```

```
        values=.5
    ),
    mxData(
        observed=dzData,
        type="raw"
    )
)
twinAEModel <- mxModel("twinAE", mzModel, dzModel,
    mxAlgebra(
        expression=MZ.objective + DZ.objective,
        name="twin"
    ),
    mxAlgebraObjective("twin")
)

twinAEFit <- mxRun(twinAEModel)


MZc <- mxEval(MZ.covariance, twinAEFit)
DZc <- mxEval(DZ.covariance, twinAEFit)
M <- mxEval(MZ.means, twinAEFit)
A <- mxEval(a*a, twinAEFit)
C <- mxEval(c*c, twinAEFit)
E <- mxEval(e*e, twinAEFit)
V <- (A+C+E)
a2 <- A/V
c2 <- C/V
e2 <- E/V
AEest <- rbind(cbind(A, C, E),cbind(a2, c2, e2))
LL_AE <- mxEval(objective, twinAEFit)
```

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, and print relevant output.

```
LRT_ACE_AE <- LL_AE - LL_ACE

#Print relevant output
ACEest
AEest
LRT_ACE_AE
```

Note that the way to specify submodels using path specification is not straightforward and requires repeating code. The OpenMx team is currently working on better alternatives. These models may also be specified using matrices instead of paths, which allow for easier submodel specification. 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/repoview/1/trunk/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/repoview/1/trunk/demo/DefinitionMeans_MatrixRaw.R

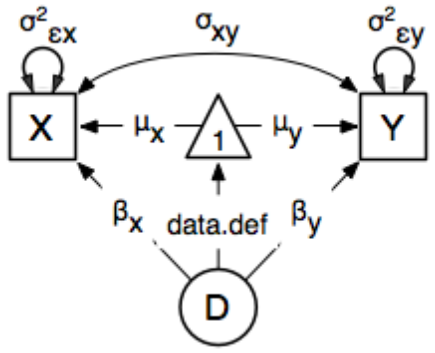
Statistical Model

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 and the residual sources of variance, ϵ_{xi} and ϵ_{yi} covary to the extent ρ. 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)
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 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)
dimnames(xy)[2] <- list(c("x","y"))
def <- rep(c(1,0),each=N)
selVars <- c("x","y")
```

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, and an objective function) are declared within the `mxModel` function. This type of code structure is recommended for OpenMx scripts generally.

```
defMeansModel <- mxModel("Definition Means Path Specification",
    type="RAM",
    manifestVars=selVars,
    latentVars  ="DefDummy",
    # variances
    mxPath(
        from=c("x","y"),
        arrows=2,
        free= TRUE,
        values=1,
        labels=c("Varx","Vary")
    ),
    # covariances
    mxPath(
        from="x",
        to="y",
        arrows=2,
        free= TRUE,
        values=.1,
        labels=c("Covxy")
    ),
    # means
    mxPath(
        from="one",
        to=c("x","y"),
        arrows=1,
        free= TRUE,
        values=1,
        labels=c("meanx","meany")
    ),
    # definition value
    mxPath(
        from="one",
```

```
        to="DefDummy",
        arrows=1,
        free= FALSE,
        values=1,
        labels="data.def"
    ),
    # beta weights
    mxPath(
        from="DefDummy",
        to=c("x","y"),
        arrows=1,
        free= TRUE,
        values=1,
        labels=c("beta_1","beta_2")
    ),
    mxData(
        observed=data.frame(xy,def),
        type="raw"
    )
)
```

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"`.

```
require(OpenMx)

defMeansModel<-mxModel("Definition Means Path Specification",
    type="RAM",
```

The second line 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.

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

Model specification is carried out using two lists of variables, `manifestVars` and `latentVars`. Then `mxPath` functions are used to specify paths between them. 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. The 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 from that of those with `data.def` values of zero.

```
# variances
mxPath(
    from=c("x","y"),
    arrows=2,
    free= TRUE,
    values=1,
    labels=c("Varx","Vary")
),
# covariances
```

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

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.

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

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 `defmeansFi@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 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/repoview/1/trunk/demo/GrowthMixtureModel_Path.R

A parallel example using matrix specification can be found here:

** http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/GrowthMixtureModel_Matrix.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/repoview/1/trunk/demo/LatentGrowthCurveModel_PathRaw.R

** http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/LatentGrowthCurveModel_MatrixRaw.R

2.7.1 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.

```
class1 <- mxModel("Class1",
    type="RAM",
    manifestVars=c("x1","x2","x3","x4","x5"),
    latentVars=c("intercept","slope"),
        # residual variances
    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
    mxPath(
        from=c("intercept","slope"),
        arrows=2,
        all=TRUE,
        free=TRUE,
        values=c(1, .4, .4, 1),
        labels=c("vari1", "cov1", "cov1", "vars1")
    ),
    # intercept loadings
    mxPath(
        from="intercept",
        to=c("x1","x2","x3","x4","x5"),
```

```
        arrows=1,
        free=FALSE,
        values=c(1, 1, 1, 1, 1)
    ),
    # slope loadings
    mxPath(
        from="slope",
        to=c("x1","x2","x3","x4","x5"),
        arrows=1,
        free=FALSE,
        values=c(0, 1, 2, 3, 4)
    ),
    # manifest means
    mxPath(from="one",
        to=c("x1", "x2", "x3", "x4", "x5"),
        arrows=1,
        free=FALSE,
        values=c(0, 0, 0, 0, 0)
    ),
    # latent means
    mxPath(from="one",
        to=c("intercept", "slope"),
        arrows=1,
        free=TRUE,
        values=c(0, -1),
        labels=c("meani1", "means1")
    )
) # close model
```

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.

```
class2 <- mxModel(class1,
        # latent variances and covariance
    mxPath(
        from=c("intercept","slope"),
        arrows=2,
        all=TRUE,
        free=TRUE,
        values=c(1, .5, .5, 1),
        labels=c("vari2", "cov2", "cov2", "vars2")
    ),
    # latent means
    mxPath(from="one",
        to=c("intercept", "slope"),
        arrows=1,
        free=TRUE,
        values=c(5, 1),
        labels=c("meani2", "means2")
    ),
        name="Class2"
) # close model
```

We must make one other change to our class-specific models before creating the parent model that will contain them. The objective 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 objective 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 `mxRAMObjective` or `mxFIMLObjective` or with the syntax below.

```
class1@objective@vector <- TRUE
class2@objective@vector <- TRUE
```

While the class-specific models can be specified using either path or matrix specification, the class proportion parameter must be specified using a matrix, though it can be specified a number of different ways. The code below demonstrates one method of specifying class proportion parameters as probabilities.

The matrix in the object `classP` contains two elements representing the proportion of the sample in each of the two classes, while the object `classA` contains an `MxAlgebra` that scales this proportion as a probability. Placing bounds on the class probabilities matrix constrains each of the probabilities to be between zero and one, while the algebra defines the probability of being in class 2 to be 1 minus the probability of being in class 1. This ensures that the sum of the class probabilities is 1. Notice that the second element of the class probability matrix is constrained to be equal to the result of the `mxAlgebra` statement. The brackets in the `mxMatrix` function are required; the second element in the "classProbs" object is actually constrained to be equal to the first row and first column of the `MxAlgebra` object "pclass2", which evaluates to a 1 x 1 matrix.

```
classP <- mxMatrix("Full", 2, 1, free=c(TRUE, FALSE),
          values=.2, lbound=0.001, ubound=0.999,
          labels = c("pclass1", "pclass2[1,1]"), name="classProbs")

classA <- mxAlgebra(1-pclass1, name="pclass2")
```

The above code creates one free parameter for class probability ("pclass1") and one fixed parameter, which is the result of an algebra ("pclass2"). There are at least two other ways to specify this class proportion parameter, each with benefits and drawbacks. One could create two free parameters named "pclass1" and "pclass2" and constrain them using the `mxConstraint` function. This approach is relatively straightforward, but comes at the expense of standard errors. Alternatively, one could omit the algebra and fix "pclass2" to a specific value. This would make model specification easier, but the resulting "pclass1" parameter would not be scaled as a probability.

Finally, we can specify the mixture model. We must first specify the model's -2 log likelihood function defined as:

$$-2LL = -2 * \sum \log(p_1 l_{1i} + p_2 l_{2i})$$

This is specified using an `mxAlgebra` function, and used as the argument to the `mxAlgebraObjective` function. Then the objective 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.

```
algObj <- mxAlgebra(-2*sum(
          log(pclass1%x%Class1.objective + pclass2%x%Class2.objective)),
          name="mixtureObj")

obj <- mxAlgebraObjective("mixtureObj")

gmm <- mxModel("Growth Mixture Model",
        mxData(
        observed=myGrowthMixtureData,
        type="raw"
    ),
    class1, class2,
```

```
    classP, classA,
    algObj, obj
        )

gmmFit <- mxRun(gmm)

summary(gmmFit)
```

Multiple Runs

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.

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    vari2     cov2    vars2   m
#           0.2      1.0      1.0      0.4      1.0      0.0     -1.0      1.0      0.5      1.0
#       means2
#           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. A `for` loop repeatedly generates starting values (from a set of uniform distributions using `runif`), runs the model with those starting values and places the starting values, final estimates and fit statistics in the `input`, `output` and `fit` matrices, respectively.

```
trials <- 20

omxGetParameters(gmm)

parNames <- names(omxGetParameters(gmm))

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, 4)
dimnames(fit) <- list(c(1: trials), c("Minus2LL", "Status", "Iterations", "pclass1"))
```

```
for (i in 1: trials){
        cp <- runif(1, 0.1, 0.9) # class probability
        v  <- runif(5, 0.1, 5.0) # variance terms
        cv <- runif(2,-0.9, 0.9) # covariances (as correlations)
        m  <- runif(4,-5.0, 5.0) # means
        cv <- cv*c(sqrt(v[2]*v[3]), sqrt(v[4]*v[5])) #rescale covariances

        temp1 <- omxSetParameters(gmm,
                labels=parNames,
                values=c(
                        cp, # class probability
                        v[1],
                        v[2], cv[1], v[3], m[1], m[2],
                        v[4], cv[2], v[5], m[3], m[4]
                        )
                        )

                        temp1@name <- paste("Starting Values Set", i)

                        temp2 <- mxRun(temp1, unsafe=TRUE, suppressWarnings=TRUE)

                        input[i,] <- omxGetParameters(temp1)
                        output[i,] <- omxGetParameters(temp2)
                        fit[i,] <- c(
                        temp2@output$Minus2LogLikelihood,
                        temp2@output$status[[1]],
                        temp2@output$iterations,
                        temp2@output$estimate[1]
                        )
                }
```

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
#       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.

Multicore Estimation

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/repoview/1/trunk/demo/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.

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/repoview/1/trunk/demo/SimpleRegression_MatrixCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/SimpleRegression_MatrixRaw.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/MultipleRegression_MatrixCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/MultipleRegression_MatrixRaw.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/MultivariateRegression_MatrixCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/MultivariateRegression_MatrixRaw.R

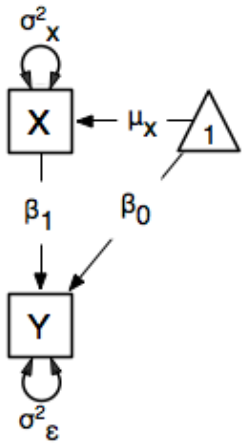
This example will focus on the RAM approach to building structural models. A parallel version of this example, using path-centric rather than matrix specification, is available here:

- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/SimpleRegression_PathCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/SimpleRegression_PathRaw.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/MultipleRegression_PathCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/MultipleRegression_PathRaw.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/MultivariateRegression_PathCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/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)

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.

```
require(OpenMx)

uniRegModel <- mxModel("Simple Regression Matrix Specification",
    mxData(
        observed=SimpleDataRaw,
        type="raw"
    ),
    # asymmetric paths
    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"
    ),
    # symmetric paths
    mxMatrix(
        type="Symm",
        nrow=2,
        ncol=2,
        values=c(1, 0,
                 0, 1),
        free=c(T, F,
               F, T),
        labels=c("varx", NA,
                  NA,    "residual"),
        byrow=TRUE,
        name="S"
    ),
    # filter matrix
    mxMatrix(
        type="Iden",
        nrow=2,
```

```
        ncol=2,
        name="F",
        dimnames=list(c("x","y"),c("x","y"))
    ),
    # means
    mxMatrix(
        type="Full",
        nrow=1,
        ncol=2,
        free=c(T, T),
        values=c(0, 0),
        labels=c("meanx", "beta0"),
        name="M"),
    mxRAMObjective("A", "S", "F", "M")
)
```

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.

```
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:

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

The next four functions specify the four matricies 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 objective 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 assymetric 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
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. 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
mxMatrix(
    type="Symm",
    nrow=2,
    ncol=2,
    values=c(1, 0,
             0, 1),
    free=c(T, F,
           F, T),
    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
mxMatrix(
    type="Iden",
    nrow=2,
    ncol=2,
    name="F",
    dimnames=list(c("x","y"),c("x","y"))
)
```

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 1 for both parameters, and is named `"M"`.

```
# means
mxMatrix(
    type="Full",
    nrow=1,
    ncol=2,
    free=c(T, T),
    values=c(0, 0),
    labels=c("meanx", "beta0"),
    name="M"
)
```

The final part of this model is the objective function. This defines both how the specified matrices combine to create the expected covariance matrix of the data, as well as the fit function to be 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. While users may define their own expected covariance matrices using other objective functions in OpenMx, the `mxRAMObjective` function yields maximum likelihood estimates of structural equation models 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 `mxRAMObjective` function takes four arguments, which are the names of the `A`, `S`, `F` and `M` matrices in your model.

```
mxRAMObjective("A", "S", "F", "M")
```

The model now includes an observed covariance matrix (i.e., data) and the matrices and objective function required 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")

uniRegModel <- mxModel("Simple Regression Matrix Specification",
    mxData(
        observed=SimpleDataRaw,
        type="raw"
    ),
    mxMatrix(
        type="Full",
        nrow=1,
        ncol=1,
        free=FALSE,
        labels=c("data.x"),
        name="X"
    ),
    mxMatrix(
        type="Full",
        nrow=1,
        ncol=1,
        free=T,
        values=0,
        labels="beta0",
        name="Intercept"
    ),
    mxMatrix(
        type="Full",
        nrow=1,
        ncol=1,
        free=T,
        values=1,
        labels="beta1",
        name="regCoef"
    ),
    mxMatrix(
        type="Diag",
        nrow=1,
        ncol=1,
        values=1,
        free=T,
        labels="residual",
        name="resVar"
    ),
    mxAlgebra(
        expression= Intercept + regCoef %*% X,
        name="expMean",
    ),

    mxAlgebra(
        expression= resVar,
        name="expCov"
    ),
    mxFIMLObjective(
        covariance="expCov",
        means="expMean",
        dimnames=selVars
    )
)
```

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.

```
uniRegModel <- mxModel("Simple Regression Matrix Specification",
    mxData(
        observed=SimpleDataRaw,
        type="raw"
    ),
    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`.

```
mxMatrix(
    type="Full",
    nrow=1,
    ncol=1,
    free=T,
    values=0,
    labels="beta0",
    name="Intercept"
),
mxMatrix(
    type="Full",
    nrow=1,
    ncol=1,
    free=T,
    values=1,
    labels="beta1",
    name="regCoef"
),
mxMatrix(
    type="Diag",
    nrow=1,
    ncol=1,
    values=1,
    free=T,
    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.

```
mxAlgebra(
     expression= Intercept + regCoef %*% X,
     name="expMean",
),
mxAlgebra(
     expression= resVar,
     name="expCov"
),
```

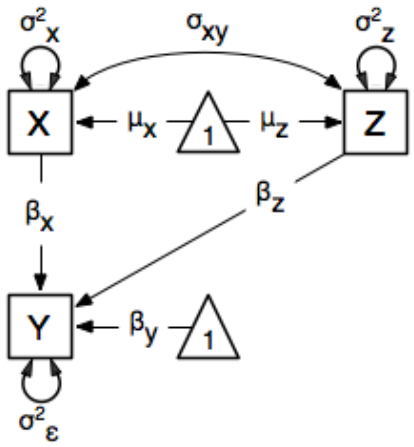
Finally, we call up the results of the algebras as the arguments for the objective function. The dimnames map the data to the model. Note that `selVars` now includes only the *y* variable.

```
mxFIMLObjective(
    covariance="expCov",
    means="expMean",
    dimnames=selVars
))
```

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^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, containing the same **A**, **S**, **F** and **M** matrices. With the addition of a third variables, the **A**, **S** and **F** matrices become **3x3**, while the **M** matrix becomes a **1x3** matrix.

```
multiRegModel<-mxModel("Multiple Regression Matrix Specification",
    mxData(
        MultipleDataRaw,
        type="raw"
    ),
    # asymmetric paths
    mxMatrix(
        type="Full",
        nrow=3,
        ncol=3,
        values=c(0,0,0,
                 1,0,1,
                 0,0,0),
        free=c(F, F, F,
               T, F, T,
               F, F, F),
        labels=c(NA,      NA, NA,
                 "betax", NA,"betaz",
                 NA,      NA, NA),
        byrow=TRUE,
        name = "A"
    ),
    # symmetric paths
    mxMatrix(
        type="Symm",
        nrow=3,
        ncol=3,
        values=c(1, 0, .5,
                 0, 1, 0,
                .5, 0, 1),
        free=c(T, F, T,
               F, T, F,
               T, F, T),
        labels=c("varx",  NA,          "covxz",
                  NA,    "residual",   NA,
                 "covxz", NA,          "varz"),
        byrow=TRUE,
        name="S"
    ),
    # filter matrix
    mxMatrix(
        type="Iden",
        nrow=3,
        ncol=3,
        name="F",
        dimnames=list(c("x","y","z"),c("x","y","z"))
    ),
    # means
    mxMatrix(
        type="Full",
        nrow=1,
        ncol=3,
        values=c(0,0,0),
        free=c(T,T,T),
```

```
        labels=c("meanx","beta0","meanz"),
        name="M"
    ),
    mxRAMObjective("A","S","F","M")
)
```

The `mxData` function now takes a different data object (`MultipleDataRaw` replaces `SingleDataRaw`, adding an additional variable), but is otherwise unchanged. The `mxRAMObjective` does 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
mxMatrix(
    type="Full",
    nrow=3,
    ncol=3,
    values=c(0,0,0,
             1,0,1,
             0,0,0),
    free=c(F, F, F,
           T, F, T,
           F, F, F),
    labels=c(NA,      NA, NA,
            "betax", NA,"betaz",
             NA,      NA, NA),
    byrow=TRUE,
    name = "A"
)
```

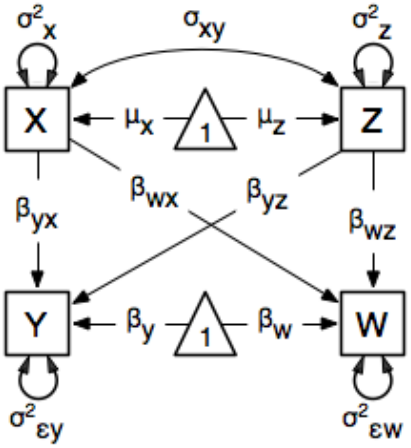
We've made a similar changes in the other matrices. The **S** matrix includes not only a variance term for the z variable, but also a covariance between the two independent variables. The **F** matrix still does not contain free parameters, but has expanded in size. The **M** matrix includes an additional 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 `mxRAMObjective` 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.

```
multivariateRegModel<-mxModel("Multiple Regression Matrix Specification",
    mxData(
        myRegDataRaw,
        type="raw"
    ),
    # asymmetric paths
    mxMatrix(
        type="Full",
        nrow=4,
        ncol=4,
        values=c(0,1,0,1,
                 0,0,0,0,
                 0,1,0,1,
                 0,0,0,0),
        free=c(F, T, F, T,
```

```
            F, F, F, F,
            F, T, F, T,
            F, F, F, F),
        labels=c(NA, "betawx", NA, "betawz",
                 NA,  NA,      NA,  NA,
                 NA, "betayx", NA, "betayz",
                 NA,  NA,      NA,  NA),
        byrow=TRUE,
        name="A"
    ),
    # symmetric paths
    mxMatrix(
        type="Symm",
        nrow=4,
        ncol=4,
        values=c(1,  0, 0,  0,
                 0,  1, 0, .5,
                 0,  0, 1,  0,
                 0, .5, 0,  1),
        free=c(T, F, F, F,
               F, T, F, T,
               F, F, T, F,
               F, T, F, T),
        labels=c("residualw",  NA,      NA,          NA,
                  NA,         "varx",   NA,         "covxz",
                  NA,          NA,     "residualy", NA,
                  NA,         "covxz", NA,         "varz"),
        byrow=TRUE,
        name="S"
    ),
    # filter matrix
    mxMatrix(
        type="Iden",
        nrow=4,
        ncol=4,
        name="F",
        dimnames=list(c("w", "x", "y", "z"),c("w", "x", "y", "z"))
    ),
    # means
    mxMatrix(
        type="Full",
        nrow=1,
        ncol=4,
        values=c(0,0,0,0),
        free=c(T,T,T,T),
        labels=c("betaw","meanx","betay","meanz"),
        name="M"
    ),
    mxRAMObjective("A","S","F","M")
)
```

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/repoview/1/trunk/demo/OneFactorModel_MatrixCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/OneFactorModel_MatrixRaw.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/TwoFactorModel_MatrixCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/TwoFactorModel_MatrixRaw.R

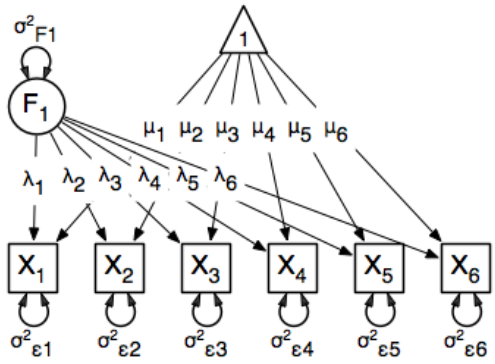
Parallel versions of this example, using path-centric specification of models rather than paths, can be found here:

- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/OneFactorModel_PathCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/OneFactorModel_PathRaw.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/TwoFactorModel_PathCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/TwoFactorModel_PathRaws.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 (6 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)

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, and an objective function) are included in their functions. 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"

oneFactorModel <- mxModel("Common Factor Model Matrix Specification",
    mxData(
        myFADataRaw,
        type="raw"
    ),
    # asymmetric paths
    mxMatrix(
        type="Full",
        nrow=7,
        ncol=7,
        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),
```

```
        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),
        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"
    ),
    # symmetric paths
    mxMatrix(
        type="Symm",
        nrow=7,
        ncol=7,
        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),
        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),
        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"
    ),
    # filter matrix
    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",
        dimnames=list(manifestVars, c(manifestVars, latentVars))
    ),
    # means
    mxMatrix(
        type="Full",
        nrow=1,
        ncol=7,
        values=c(1,1,1,1,1,1,0),
        free=c(T,T,T,T,T,T,F),
        labels=c("meanx1","meanx2","meanx3","meanx4","meanx5","meanx6",NA),
        name="M"
    ),
    mxRAMObjective("A","S","F","M")
)
```

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.

```
mxData(
    observed=oneFactorRaw,
    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:

```
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
mxMatrix(
    type="Full",
    nrow=7,
    ncol=7,
    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),
    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),
    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
mxMatrix(
    type="Symm",
    nrow=7,
    ncol=7,
    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),
    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),
    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
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
mxMatrix(
    type="Full",
    nrow=1,
    ncol=7,
    values=c(1,1,1,1,1,1,0),
    free=c(T,T,T,T,T,T,F),
    labels=c("meanx1","meanx2","meanx3","meanx4","meanx5","meanx6",NA),
    name="M"
)
```

The final part of this model is the objective function. This defines both how the specified matrices combine to create the expected covariance matrix of the data, as well as the fit function to be 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. While users may define their own expected covariance matrices

using other objective functions in OpenMx, the `mxRAMObjective` function yields maximum likelihood estimates of structural equation models 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 `mxRAMObjective` function takes four arguments, which are the names of the **A**, **S**, **F** and **M** matrices in your model.

```
mxRAMObjective("A", "S", "F", "M")
```

The model now includes an observed covariance matrix (i.e., data) and the matrices and objective function required 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.

```
oneFactorModel <- mxModel("Common Factor Model Matrix Specification",
    mxData( observed=myFADataRaw, type="raw" ),
    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" ),
    mxMatrix( type="Symm", nrow=1, ncol=1, values=1, free=T,
        labels="varF1",
        name="facVariances" ),
    mxMatrix( type="Diag", nrow=6, ncol=6, free=T, values=1,
        labels=c("e1","e2","e3","e4","e5","e6"),
        name="resVariances" ),
    mxMatrix( type="Full", nrow=1, ncol=6, values=1, free=T,
        labels=c("meanx1","meanx2","meanx3","meanx4","meanx5","meanx6"),
        name="varMeans" ),
    mxMatrix( type="Full", nrow=1, ncol=1, values=0, free=F,
        name="facMeans" ),
    mxAlgebra( expression= facLoadings %&% facVariances + resVariances,
        name="expCov" ),
    mxAlgebra(expression= varMeans + t(facLoadings %*% facMeans),
        name="expMean" ),
    mxFIMLObjective( covariance="expCov", means="expMean", dimnames=manifestVars)
)
oneFactorFit<-mxRun(oneFactorModel)
```

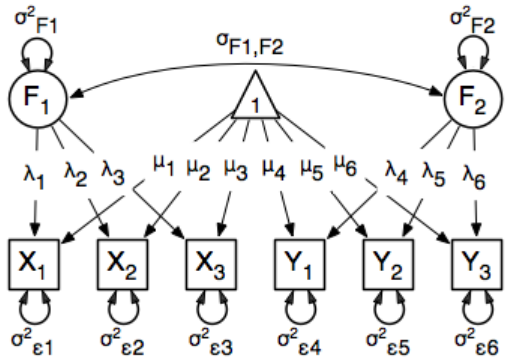
The first `mxMatrix` statement declares a `Full` **6x1** matrix of factor loadings to be estimated, called "facLoadings". We fix the first factor loading to 1 for identification. Even though we specify just one start value of 1 which is recycled for each of the elements in the matrix, it becomes the 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 statement to work out the expected mean 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 `mxFIMLObjective` now refer to these algebras for the expected covariance and expected means. The `dimnames` are used to map them onto the observed variables.

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 (6 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`).d

```
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 `mxRAMObjective` has not changed. The code for our two factor model looks like this:

```
twoFactorModel <- mxModel("Two Factor Model Matrix Specification",
    type="RAM",
    mxData(
        observed=twoFactorRaw,
        type="raw",
    ),
    # asymmetric paths
    mxMatrix(
```

```
    type="Full",
    nrow=8,
    ncol=8,
    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),
    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),
    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"
),
# symmetric paths
mxMatrix(
    type="Symm",
    nrow=8,
    ncol=8,
    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),
    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),
    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"
    ),
    # filter matrix
    mxMatrix(
        type="Full",
        nrow=6,
        ncol=8,
        free=F,
        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=T,
        name="F"
    ),
    # means
    mxMatrix(
        type="Full",
        nrow=1,
        ncol=8,
        values=c(1,1,1,1,1,1,0,0),
        free=c(T,T,T,T,T,T,F,F),
        labels=c("meanx1","meanx2","meanx3",
                 "meanx4","meanx5","meanx6",
                  NA,NA),
        name="M"
    ),
    mxRAMObjective("A","S","F","M")
)
```

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
mxMatrix(
    type="Full",
    nrow=8,
    ncol=8,
    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),
    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),
    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
mxMatrix(
    type="Symm",
    nrow=8,
    ncol=8,
    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),
    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),
    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/repoview/1/trunk/demo/LatentGrowthCurveModel_MatrixCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/LatentGrowthCurveModel_MatrixRaw.R

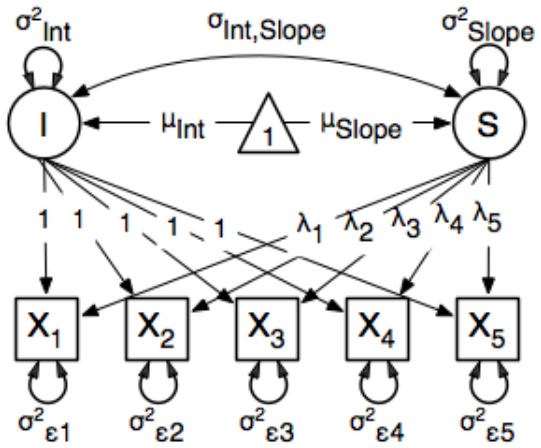
Parallel versions of this example, using path-centric specification of models rather than matrices, can be found here:

- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/LatentGrowthCurveModel_PathCov.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/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)
```

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, and an objective function) are included in their functions. This code uses the `mxModel` function to create an `MxModel` object, which we will then run.

```
require(OpenMx)

growthCurveModel <- mxModel("Linear Growth Curve Model Matrix Specification",
    mxData(
        myLongitudinalData,
        type="raw"
    ),
    # asymmetric paths
    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"
    ),
    # symmetric paths
    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,  1,0.5,
                 0,0,0,0,0,0.5,  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"
    ),
    # filter matrix
    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",
                    dimnames=list(NULL, c("x1","x2","x3","x4","x5","","")) )
    ),
    # means
        mxMatrix(
        type="Full",
        nrow=1,
        ncol=7,
        values=c(0,0,0,0,0,1,1),
        free=c(F,F,F,F,F,T,T),
        labels=c(NA,NA,NA,NA,NA,"meani","means"),
        name="M"
    ),
    mxRAMObjective("A","S","F","M")
)
```

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.

```
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
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 sevenths rows and columns of this matrix as well.

```
# symmetric paths
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,  1,0.5,
            0,0,0,0,0,0.5,  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
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
mxMatrix(
    type="Full",
    nrow=1,
    ncol=7,
    values=c(0,0,0,0,0,1,1),
    free=c(F,F,F,F,F,T,T),
    labels=c(NA,NA,NA,NA,NA,"meani","means"),
    name="M"
)
```

The last piece of our model is the `mxRAMObjective` function, which defines both how the specified matrices combine to create the expected covariance matrix of the data, as well as the fit function to be minimized. 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 `mxRAMObjective` function takes four arguments, which are the names of the **A**, **S**, **F** and **M** matrices in your model.

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/repoview/1/trunk/demo/BivariateHeterogeneity_MatrixRaw.R

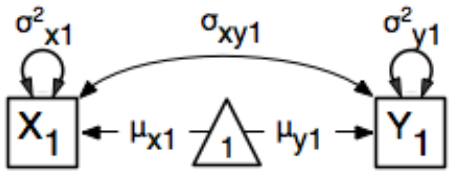
A parallel version of this example, using paths specification of models rather than matrices, can be found here:

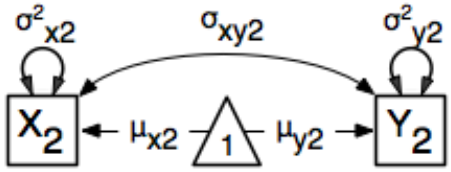
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/BivariateHeterogeneity_PathRaw.R

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)
rs=0.5
xy1 <- mvrnorm (1000, c(0,0), matrix(c(1,rs,rs,1),2,2))
set.seed(200)
#group 2
rs=0.4
xy2 <- mvrnorm (1000, c(0,0), matrix(c(1,rs,rs,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 look 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"),
     mxAlgebra(group1.objective + group2.objective, name="minus2loglikelihood"),
     mxAlgebraObjective("minus2loglikelihood")
)
```

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.

```
#Fit Heterogeneity Model
bivHetModel <- mxModel("bivHet",
    mxModel("group1",
        mxMatrix(
            type="Lower",
            nrow=2,
            ncol=2,
            free=T,
            values=.5,
            labels=c("Ch11", "Ch21", "Ch31"),
            name="Chol1"
        ),
        mxAlgebra(
            Chol1 %*% t(Chol1),
            name="EC1"
        ),
        mxMatrix(
            type="Full",
            nrow=1,
            ncol=2,
            free=T,
            values=c(0,0),
            labels=c("mX1", "mY1"),
            name="EM1"
        ),
        mxData(
            xy1,
            type="raw"
        ),
        mxFIMLObjective(
            covariance="EC1",
            means="EM1",
            dimnames=selVars
        )
    ),
    mxModel("group2",
        mxMatrix(
            type="Lower",
            nrow=2,
            ncol=2,
            free=T,
            values=.5,
            labels=c("Ch12", "Ch22", "Ch32"),
            name="Chol2"
```

```
        ),
        mxAlgebra(
            Chol2 %*% t(Chol2),
            name="EC2"
        ),
        mxMatrix(
            type="Full",
            nrow=1,
            ncol=2,
            free=T,
            values=c(0,0),
            labels=c("mX2", "mY2"),
            name="EM2"
        ),
        mxData(
            xy2,
            type="raw"
        ),
        mxFIMLObjective(
            covariance="EC2",
            means="EM2",
            dimnames=selVars
        )
    ),
```

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  "Ch11"     NA
    Y  "Ch21"  "Ch22"

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

in group2
    $S
            X      Y
    X  "Ch12"     NA
    Y  "Ch22" "Ch32"

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

To evaluate both models together, we use an `mxAlgebra` command that adds up the values of the objective functions of the two groups. The objective function to be used here is the `mxAlgebraObjective` which uses as its argument the sum of the function values of the two groups, referred to by the name of the previously defined `mxAlgebra` object `h12`.

```
    mxAlgebra(
        group1.objective + group2.objective,
        name="minus2loglikelihood"
```

```
    ),
    mxAlgebraObjective("minus2loglikelihood")
)
```

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.

```
EM1Het <- mxEval(group1.EM1, bivHetFit)
EM2Het <- mxEval(group2.EM2, bivHetFit)
EC1Het <- mxEval(group1.EC1, bivHetFit)
EC2Het <- mxEval(group2.EC2, bivHetFit)
LLHet <- mxEval(objective, bivHetFit)
```

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 group1 to the labels of group2, 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$group2.EM2@labels <- bivHomModel$group1.EM1@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)
    EM1Hom <- mxEval(group1.EM1, bivHomFit)
    EM2Hom <- mxEval(group2.EM2, bivHomFit)
    EC1Hom <- mxEval(group1.EC1, bivHomFit)
    EC2Hom <- mxEval(group2.EC2, bivHomFit)
    LLHom <- mxEval(objective, bivHomFit)
```

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/repoview/1/trunk/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/repoview/1/trunk/demo/UnivariateTwinAnalysis_PathRaw.R

3.5.1 ACE Model: a Twin Analysis

Data

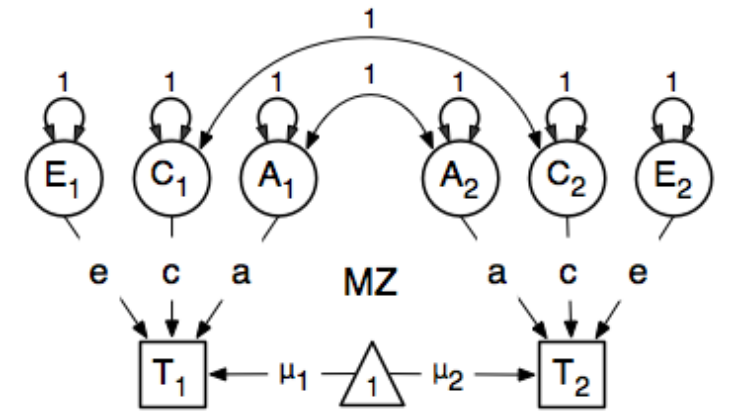
Let us assume you have collected data on a large sample of twin pairs for your phenotype of interest. For illustration purposes, we use Australian data on body mass index (BMI) which are saved in a text file 'myTwinData.txt'. We use R to read the data into a data.frame and to create two subsets of the data for MZ females (mzData) and DZ females (dzData) respectively with the code below.

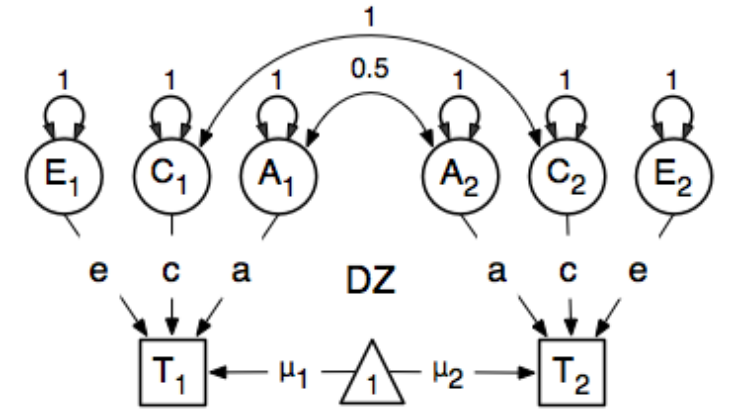
```
require(OpenMx)

#Prepare Data
data(myTwinData)
twinVars <- c(  'fam','age','zyg','part',
                'wt1','wt2','ht1','ht2','htwt1','htwt2','bmi1','bmi2')
summary(myTwinData)
selVars <- c('bmi1','bmi2')
mzData <- as.matrix(subset(myTwinData, zyg==1, c(bmi1,bmi2)))
dzData <- as.matrix(subset(myTwinData, zyg==3, c(bmi1,bmi2)))
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 observed 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.

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 have become 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 in yet another model, called `ACE` which will then be evaluated together with the two submodels for each of the twin types, defined in separate `mxModel` commands, as they are all three arguments of the overall `twinACE` model, and will be saved together in the R object `twinACEModel` and thus be run together.

```
require(OpenMx)

twinACEModel <- mxModel("twinACE",
    mxModel("ACE",
    # Matrices a, c, and e to store a, c, and e path coefficients
        mxMatrix(
            type="Lower",
            nrow=1,
            ncol=1,
            free=TRUE,
            values=0.6,
            labels="a11",
            name="a"
        ),
        mxMatrix(
            type="Lower",
            nrow=1,
```

```
        ncol=1,
        free=TRUE,
        values=0.6,
        labels="c11",
        name="c"
    ),
    mxMatrix(
        type="Lower",
        nrow=1,
        ncol=1,
        free=TRUE,
        values=0.6,
        labels="e11",
        name="e"
    ),
# Matrices A, C, and E compute variance components
    mxAlgebra(
        expression=a %*% t(a),
        name="A"
    ),
    mxAlgebra(
        expression=c %*% t(c),
        name="C"
    ),
    mxAlgebra(
        expression=e %*% t(e),
        name="E"
    ),
# Matrix & Algebra for expected means vector
    mxMatrix(
        type="Full",
        nrow=1,
        ncol=1,
        free=TRUE,
        values=20,
        label="mean",
        name="Mean"
    ),
    mxAlgebra(
        expression= cbind(Mean,Mean),
        name="expMean"
    ),
# Algebra for expected variance/covariance matrix in MZ
    mxAlgebra(
        expression=rbind (cbind(A + C + E , A + C),
                          cbind(A + C     , A + C + E)),
        name="expCovMZ"
    ),
# Algebra for expected variance/covariance matrix in DZ
    mxAlgebra(
        expression=rbind (cbind(A + C + E     , 0.5 %x% A + C),
                          cbind(0.5 %x% A + C , A + C + E)),
        name="expCovDZ"
    )
),
mxModel("MZ",
    mxData(
        observed=mzData,
```

```
            type="raw"
        ),
        mxFIMLObjective(
            covariance="ACE.expCovMZ",
            means="ACE.expMean",
            dimnames=selVars
        )
    ),
    mxModel("DZ",
        mxData(
            observed=dzData,
            type="raw"
        ),
        mxFIMLObjective(
            covariance="ACE.expCovDZ",
            means="ACE.expMean",
            dimnames=selVars
        )
    ),
    mxAlgebra(
        expression=MZ.objective + DZ.objective,
        name="minus2loglikelihood"
    ),
    mxAlgebraObjective("minus2loglikelihood")
)

twinACEFit <- mxRun(twinACEModel)
```

They will all form arguments of the `mxModel`, specified as follows. Note that we left the comma's at the end of the lines which are necessary when all the arguments are combined prior to running the model. Each line can be pasted into R, and then evaluated together once the whole model is specified.

```
#Fit ACE Model with RawData and Matrix-style Input
twinACEModel <- mxModel("twinACE",
    mxModel("ACE",
```

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` 1x1 matrices assigned the `free` status and given a 0.6 starting value.

```
# Matrices a, c, and e to store a, c, and e path coefficients
# additive genetic path
mxMatrix(
    type="Full",
    nrow=1,
    ncol=1,
    free=TRUE,
    values=0.6,
    label="a11",
    name="a"
),
# shared environmental path
mxMatrix(
    type="Full",
    nrow=1,
    ncol=1,
    free=TRUE,
```

```
    values=0.6,
    label="c11",
    name="c"
),
# specific environmental path
mxMatrix(
    type="Full",
    nrow=1,
    ncol=1,
    free=TRUE,
    values=0.6,
    label="e11",
    name="e"
),
```

While the labels in these 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).

```
# Matrices A, C, and E compute variance components
# additive genetic variance
mxAlgebra(
    expression=a * t(a),
    name="A"
),
# shared environmental variance
mxAlgebra(
    expression=c * t(c),
    name="C"
),
# specific environmental variance
mxAlgebra(
    expression=e * t(e),
    name="E"
),
```

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` by concatenating the `Full` **1x1** matrix `Mean` with one free element, labeled `mean` and given a start value of `20`. The `expMean` matrix is then used in both the MZ and DZ model so that all four elements representing means are equated.

```
# Matrix & Algebra for expected means vector
    mxMatrix(
        type="Full",
        nrow=1,
        ncol=1,
        free=TRUE,
        values=20,
        label="mean",
        name="Mean"
    ),
    mxAlgebra(
        expression= cbind(Mean,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 variance/covariance matrix in MZ
    mxAlgebra(
        expression=rbind (cbind(A + C + E , A + C),
                          cbind(A + C     , A + C + E)),
        name="expCovMZ"
    ),
# Algebra for expected variance/covariance matrix in DZ
    mxAlgebra(
        expression=rbind (cbind(A + C + E     , 0.5 %x% A + C),
                          cbind(0.5 %x% A + C , A + C + E)),
        name="expCovDZ"
    )
),
```

As the expected covariance matrices are different for the two groups of twins, we specify two `mxModel` commands within the 'twinACE' mxModel command. They are given a name, and arguments for the data and the objective function to be used to optimize the model. We have set the model up for raw data, and thus will use the `mxFIMLObjective` function to evaluate it. For each model, the `mxData` command calls up the appropriate data, and provides a type, here `raw`, and the `mxFIMLObjective` command is given the names corresponding to the respective expected covariance matrices and mean vectors, specified above. Given the objects `expCovMZ`, `expCovDZ` and `expMean` were created in the mxModel named `twinACE` we need to use two-level names, starting with the model name separated from the object with a dot, i.e. `twinACE.expCovMZ`.

```
mxModel("MZ",
    mxData(
        observed=mzData,
        type="raw"
    ),
    mxFIMLObjective(
        covariance="ACE.expCovMZ",
        means="ACE.expMean",
        dimnames=selVars
    )
),
mxModel("DZ",
    mxData(
        observed=dzData,
        type="raw"
    ),
    mxFIMLObjective(
        covariance="ACE.expCovDZ",
        means="ACE.expMean",
        dimnames=selVars
    )
),
```

Finally, both models need to be evaluated simultaneously. We first generate the sum of the objective functions for the two groups, using `mxAlgebra`. We refer to the correct objective function (object named `objective`) by adding the name of the model to the two-level argument, i.e. `MZ.objective`. We then use that as argument of the `mxAlgebraObjective` command.

```
    mxAlgebra(
        expression=MZ.objective + DZ.objective,
        name="minus2loglikelihood"
    ),
    mxAlgebraObjective("minus2loglikelihood")
)
```

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.

```
#Run ACE model
twinACEFit <- mxRun(twinACEModel)
```

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.

```
MZc <- mxEval(ACE.expCovMZ, twinACEFit)
DZc <- mxEval(ACE.expCovDZ, twinACEFit)
M <- mxEval(ACE.expMean, twinACEFit)
A <- mxEval(ACE.A, twinACEFit)
C <- mxEval(ACE.C, twinACEFit)
E <- mxEval(ACE.E, twinACEFit)
V <- (A+C+E)
a2 <- A/V
c2 <- C/V
e2 <- E/V
ACEest <- rbind(cbind(A,C,E),cbind(a2,c2,e2))
LL_ACE <- mxEval(objective, twinACEFit)
```

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 and save it as a new model `twinAEModel`. Next we re-specify the matrix **c** to be fixed to zero. We can run this model in the same way as before and generate similar summaries of the results.

```
#Run AE model
    twinAEModel <- mxRename(twinACEModel, "twinAE")

# drop shared environmental path
twinAEModel$ACE.c <-
    mxMatrix(
        type="Full",
        nrow=1,
        ncol=1,
        free=F,
        values=0,
        label="c11"
    )

twinAEFit <- mxRun(twinAEModel)

MZc <- mxEval(ACE.expCovMZ, twinAEFit)
DZc <- mxEval(ACE.expCovDZ, twinAEFit)
A <- mxEval(ACE.A, twinAEFit)
C <- mxEval(ACE.C, twinAEFit)
E <- mxEval(ACE.E, twinAEFit)
V <- (A+C+E)
a2 <- A/V
c2 <- C/V
e2 <- E/V
AEest <- rbind(cbind(A,C,E),cbind(a2,c2,e2))
LL_AE <- mxEval(objective, twinAEFit)
```

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
ACEest
AEest
LRT_ACE_AE
```

Note that the OpenMx team is currently working on better alternatives for dropping parameters. 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 model can be specified in this fashion; some are 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/repoview/1/trunk/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/repoview/1/trunk/demo/DefinitionMeans_PathRaw.R

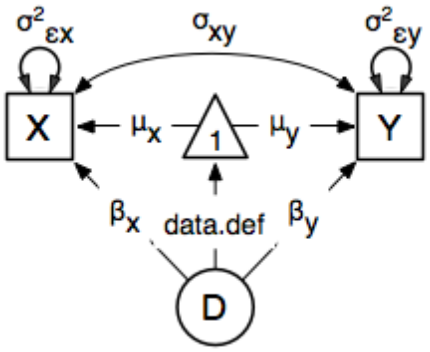
Statistical Model

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 ρ. 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)
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)
dimnames(xy)[2]<-list(c("x","y"))
def<-rep(c(1,0),each=N)
selVars<-c("x","y")
```

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.

```
defMeansModel <- mxModel("Definition Means Matrix Specification",
    mxFIMLObjective(
        covariance="Sigma",
        means="Mu",
        dimnames=selVars
    ),
```

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 Matrix Specification"`.

The second argument in this `mxModel` call is itself a function. It declares that the objective function to be optimized is full information maximum likelihood (FIML) under normal theory, which is tagged as `mxFIMLObjective`. 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.

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 `M` 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 `M` along with 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 `mxData` data frame is referred to as `data.def`. In the present case, the definition variable contains a 1 for group 1, and a zero otherwise.

```
# covariance matrix
mxMatrix(
    type="Symm",
    nrow=2,
    ncol=2,
    free=TRUE,
    values=c(1, 0, 1),
    name="Sigma"
),
# means
mxMatrix(
```

```
    type="Full",
    nrow=1,
    ncol=2,
    free=TRUE,
    name="M"
),
# regression coefficient
mxMatrix(
    type="Full",
    nrow=1,
    ncol=2,
    free=TRUE,
    values=c(0, 0),
    name="beta"
),
# definition variable
mxMatrix(
    type="Full",
    nrow=1,
    ncol=2,
    free=FALSE,
    labels="data.def",
    name="def"
),
```

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:

```
mxAlgebra(
     expression= M+beta*def,
     name="Mu"
 ),
```

The result of this algebra is named `Mu`, and this handle is referred to in the `mxFIMLObjective` function call.

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. Analyses using definition variables have to use raw data, so that the model can be specified on an individual data vector level.

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

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)
M <- mxEval(M, defMeansFit)
beta <- mxEval(beta, defMeansFit)

# Third, check to see if things are more or less equal
omxCheckCloseEnough(ObsCovs,Sigma,.01)
omxCheckCloseEnough(ObsMeansGroup1,as.vector(M+beta),.001)
omxCheckCloseEnough(ObsMeansGroup2,as.vector(M),.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 ABO Blood Groups, 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/repoview/1/trunk/demo/OneLocusLikelihood.R

and

- http://openmx.psyc.virginia.edu/repoview/1/trunk/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/repoview/1/trunk/models/passing/mx-scripts/one.mx
- http://openmx.psyc.virginia.edu/repoview/1/trunk/models/passing/mx-scripts/two.mx

3.7.1 Specification of the Single Locus Model

Data

The data for this example come from the German mathematician *Felix Bernstein* <*http://en.wikipedia.org/wiki/Felix_Bernstein*> (1924). Their observed frequencies are shown in this 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

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 $p, q' and : math : `r$, 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 $p, q' and : math : `r$. 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

OneLocusModel <- mxModel("OneLocus",
# Matrices for allele frequencies, p, q and r
    mxMatrix( type="Full", nrow=1, ncol=1, free=TRUE, values=c(.3333), name="P"),
    mxMatrix( type="Full", nrow=1, ncol=1, free=TRUE, values=c(.3333), name="Q"),
    mxMatrix( type="Full", nrow=1, ncol=1, free=TRUE, values=c(.3333), name="R"),
# Constraint for sum of allele frequencies to equal 1.0
    mxConstraint(P+Q+R == 1, name="SumAlleleFreqs"),
# Matrix of observed data
    mxMatrix( type="Full", nrow=4, ncol=1, values=c(211,104,39,148),
        name="ObservedFreqs"),
# Algebra for predicted proportions
    mxAlgebra( expression=rbind(P*(P+2*R), Q*(Q+2*R), 2*P*Q, R*R),
        name="ExpectedFreqs"),
# Algebra for -logLikelihood
```

```
    mxAlgebra( expression=-(sum(log(ExpectedFreqs) * ObservedFreqs)),
        name="NegativeLogLikelihood"),
# User-defined objective
    mxAlgebraObjective("NegativeLogLikelihood")
)

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)*t^2
B	aaBB, aaBb	261	q^2(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

TwoLocusModel <- mxModel("TwoLocus",
# Matrices for allele frequencies, p and s
    mxMatrix( type="Full", nrow=1, ncol=1, free=TRUE, values=c(.3333), name="P"),
    mxMatrix( type="Full", nrow=1, ncol=1, free=TRUE, values=c(.3333), name="S"),
# Matrix of observed data
    mxMatrix( type="Full", nrow=4, ncol=1, values=c(211,104,39,148),
        name="ObservedFreqs"),
# Algebra for predicted proportions
    mxAlgebra( expression=1-P, name="Q"),
    mxAlgebra( expression=1-S, name="T"),
```

```
    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 -logLikelihood
    mxAlgebra( expression=-(sum(log(ExpectedFreqs) * ObservedFreqs)),
        name="NegativeLogLikelihood"),
# User-defined objective
    mxAlgebraObjective("NegativeLogLikelihood")
)

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.

3.8 Factor Analysis Ordinal, Matrix Specification

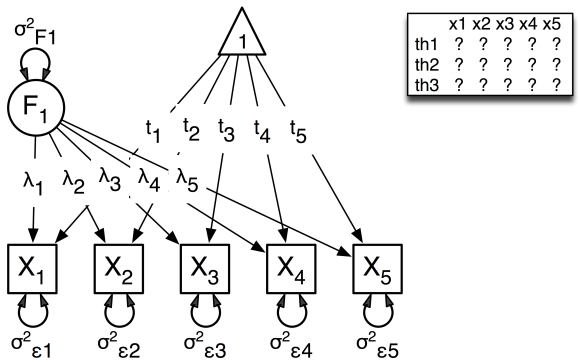
This example builds on the 'Factor Analysis, Matrix Specification' example, and extends it to the ordinal case. We will discuss a common factor model with several indicators. This example can be found in the following files:

- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/OneFactorModelOrdinal_MatrixRaw.R
- http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/OneFactorModelOrdinal01_MatrixRaw.R

3.8.1 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 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

fruitynames<-paste("banana",1:nVariables,sep="")
names(ordinalData)<-fruitynames
```

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, and an objective 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.

```
oneFactorThresholdModel <- mxModel("oneFactorThresholdModel",
    mxMatrix(
        type="Full",
        nrow=nVariables,
        ncol=nFactors,
        free=TRUE,
        values=0.2,
        lbound=-.99,
        ubound=.99,
        name="facLoadings"
    ),
    mxMatrix(
        type="Unit",
        nrow=nVariables,
        ncol=1,
        name="vectorofOnes"
    ),
    mxAlgebra(
        expression=vectorofOnes - (diag2vec(facLoadings %*% t(facLoadings))) ,
        name="resVariances"
    ),
    mxAlgebra(
        expression=facLoadings %*% t(facLoadings) + vec2diag(resVariances),
        name="expCovariances"
    ),
    mxMatrix(
        type="Zero",
        nrow=1,
        ncol=nVariables,
        name="expMeans"
    ),
    mxMatrix(
        type="Full",
        nrow=nThresholds,
        ncol=nVariables,
        free=TRUE,
        values=.2,
        lbound=rep( c(-Inf,rep(.01,(nThresholds-1))) , nVariables),
        dimnames=list(c(), fruitynames),
        name="thresholdDeviations"
    ),
    mxMatrix(
        type="Lower",
        nrow=nThresholds,
        ncol=nThresholds,
        free=FALSE,
        values=1,
        name="unitLower"
    ),
```

```
    mxAlgebra(
        expression=unitLower %*% thresholdDeviations,
        name="expThresholds"
    ),
    mxData(
        observed=ordinalData,
        type='raw'
    ),
    mxFIMLObjective(
        covariance="expCovariances",
        means="expMeans",
        dimnames=fruitynames,
        thresholds="expThresholds"
    )
)
```

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.

```
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
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** `mxMatrix` 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`.

```
mxMatrix(
    type="Unit",
    nrow=nVariables,
    ncol=1,
    name="vectorofOnes"
)
# residuals
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`.

```
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
mxMatrix(
    type="Zero",
    nrow=1,
    ncol=nVariables,
    name="expMeans"
)
```

The mean of the factor(s) is also fixed to 1, which is implied by not including a matrix for it. Alternatively, we could explicitly add a `Full` **1 x nFactors** `mxMatrix` 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`.

```
mxMatrix(
     type="Full",
     nrow=nThresholds,
     ncol=nVariables,
     free=TRUE,
     values=.2,
     lbound=rep( c(-Inf,rep(.01,(nThresholds-1))) , nVariables),
     dimnames=list(c(), fruitynames),
     name="thresholdDeviations"
 )
 mxMatrix(
     type="Lower",
     nrow=nThresholds,
     ncol=nThresholds,
     free=FALSE,
     values=1,
     name="unitLower"
 )
 # expected thresholds
 mxAlgebra(
     expression=unitLower %*% thresholdDeviations,
     name="expThresholds"
 )
```

The final part of this model is the objective function. The choice of fit 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.

```
mxFIMLObjective(
    covariance="expCovariances",
    means="expMeans",
    dimnames=fruitynames,
    thresholds="expThresholds"
)
```

The free parameters in the model can then be estimated using full information maximum likelihood (FIML) for covariances, means and thresholds. To do so, 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()`.

```
oneFactorFit <- mxRun(oneFactorThresholdModel)

oneFactorFit@output

summary(oneFactorFit)
```

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:

```
oneFactorThresholdModel01 <- mxModel("oneFactorThresholdModel01",
    mxMatrix(
        type="Full",
        nrow=nVariables,
        ncol=nFactors,
        free=TRUE,
```

```
        values=0.2,
        lbound=-.99,
        ubound=2,
        name="facLoadings"
    ),
    mxMatrix(
        type="Diag",
        nrow=nVariables,
        ncol=nVariables,
        free=TRUE,
        values=0.9,
        name="resVariances"
    ),
    mxAlgebra(
        expression=facLoadings %*% t(facLoadings) + resVariances,
        name="expCovariances"
    ),
    mxMatrix(
        type="Full",
        nrow=1,
        ncol=nVariables,
        free=TRUE,
        name="expMeans"
    ),
    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(), fruitynames),
        name="thresholdDeviations"
    ),
    mxMatrix(
        type="Lower",
        nrow=nThresholds,
        ncol=nThresholds,
        free=FALSE,
        values=1,
        name="unitLower"
    ),
    mxAlgebra(
        expression=unitLower %*% thresholdDeviations,
        name="expThresholds"
    ),
    mxMatrix(
        type="Unit",
        nrow=nThresholds,
        ncol=1,
        name="columnofOnes"
    ),
    mxAlgebra(
        expression=expMeans %x% columnofOnes,
        name="meansMatrix"
    ),
    mxAlgebra(
        expression=sqrt(t(diag2vec(expCovariances))) %x% columnofOnes,
```

```
        name="variancesMatrix"
    ),
    mxAlgebra(
        expression=(expThresholds - meansMatrix) / variancesMatrix,
        name="thresholdMatrix"
    ),
    mxMatrix(
        type="Iden",
        nrow=nVariables,
        ncol=nVariables,
        name="Identity"
    ),
    mxAlgebra(
        expression=solve(sqrt(Identity * expCovariances)) %*% facLoadings,
        name="standFacLoadings"
    ),
    mxData(
        observed=ordinalData,
        type='raw'
    ),
    mxFIMLObjective(
        covariance="expCovariances",
        means="expMeans",
        dimnames=fruitynames,
        thresholds="expThresholds"
    )
)
```

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.

```
mxMatrix(
    type="Full",
    nrow=nVariables,
    ncol=nFactors,
    free=TRUE,
    values=0.2,
    lbound=-.99,
    ubound=2,
    name="facLoadings"
)
mxMatrix(
    type="Diag",
    nrow=nVariables,
    ncol=nVariables,
    free=TRUE,
    values=0.9,
    name="resVariances"
)
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.

```
mxMatrix(
    type="Full",
    nrow=1,
    ncol=nVariables,
    free=TRUE,
    name="expMeans"
)
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(), fruitynames),
    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.

```
mxMatrix(
    type="Unit",
    nrow=nThresholds,
    ncol=1,
    name="columnofOnes"
)
mxAlgebra(
    expression=expMeans %x% columnofOnes,
    name="meansMatrix"
)
mxAlgebra(
    expression=sqrt(t(diag2vec(expCovariances))) %x% columnofOnes,
    name="variancesMatrix"
)
mxAlgebra(
    expression=(expThresholds - meansMatrix) / variancesMatrix,
    name="thresholdMatrix"
)
mxMatrix(
    type="Iden",
    nrow=nVariables,
```

```
    ncol=nVariables,
    name="Identity"
)
mxAlgebra(
    expression=solve(sqrt(Identity * expCovariances)) %*% facLoadings,
    name="facLoadingsMatrix"
)
```

3.9 Growth Mixture Modeling, Matrix Specification

This example will demonstrate how to specify a growth mixture model using matrix 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/repoview/1/trunk/demo/GrowthMixtureModel_Matrix.R

A parallel example using path specification can be found here:

** http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/GrowthMixtureModel_Path.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/repoview/1/trunk/demo/LatentGrowthCurveModel_PathRaw.R

** http://openmx.psyc.virginia.edu/repoview/1/trunk/demo/LatentGrowthCurveModel_MatrixRaw.R

3.9.1 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 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 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. Note that the `vector` argument in the `mxRAMObjective` has been set to "TRUE", which will be discussed in more detail shortly.

```
class1 <- mxModel("Class1",
    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"
    ),
    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,  1,0.5,
                 0,0,0,0,0,0.5,  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, "vari1", "cov1",
                 NA, NA, NA, NA, NA, "cov1", "vars1"),
        byrow= TRUE,
        name="S"
    ),
```

```
    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,
        dimnames=list(NULL, c(names(myGrowthMixtureData), "intercept", "slope")),
        name="F"
    ),
    mxMatrix(
        type="Full",
        nrow=1,
        ncol=7,
        values=c(0,0,0,0,0,1,1),
        free=c(F,F,F,F,F,T,T),
        labels=c(NA,NA,NA,NA,NA,"meani1","means1"),
        name="M"
    ),
    mxRAMObjective("A","S","F","M", vector=TRUE)
) # close model
```

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.

```
class2 <- mxModel(class1,
        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,  1,0.5,
                 0,0,0,0,0,0.5,  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, "vari2", "cov2",
                NA, NA, NA, NA, NA, "cov2", "vars2"),
        byrow= TRUE,
        name="S"
    ),
    mxMatrix(
        type="Full",
        nrow=1,
        ncol=7,
        values=c(0,0,0,0,0,1,1),
        free=c(F,F,F,F,F,T,T),
        labels=c(NA,NA,NA,NA,NA,"meani2","means2"),
        name="M"
    ),
        name="Class2"
) # close model
```

The `vector=TRUE` argument in the above code merits further discussion. The objective 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 objective 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 `mxRAMObjective` or `mxFIMLObjective`, as was done above, or with the syntax below.

```
class1@objective@vector <- TRUE
class2@objective@vector <- TRUE
```

While the class-specific models can be specified using either path or matrix specification, the class proportion parameter must be specified using a matrix, though it can be specified a number of different ways. The code below demonstrates one method of specifying class proportion parameters as probabilities.

The matrix in the object `classP` contains two elements representing the proportion of the sample in each of the two classes, while the object `classA` contains an `MxAlgebra` that scales this proportion as a probability. Placing bounds on the class probabilities matrix constrains each of the probabilities to be between zero and one, while the algebra defines the probability of being in class 2 to be 1 minus the probability of being in class 1. This ensures that the sum of the class probabilities is 1. Notice that the second element of the class probability matrix is constrained to be equal to the result of the `mxAlgebra` statement. The brackets in the `mxMatrix` function are required; the second element in the "classProbs" object is actually constrained to be equal to the first row and first column of the `MxAlgebra` object "pclass2", which evaluates to a 1 x 1 matrix.

```
classP <- mxMatrix("Full", 2, 1, free=c(TRUE, FALSE),
          values=.2, lbound=0.001, ubound=0.999,
          labels = c("pclass1", "pclass2[1,1]"), name="classProbs")

classA <- mxAlgebra(1-pclass1, name="pclass2")
```

The above code creates one free parameter for class probability ("pclass1") and one fixed parameter, which is the result of an algebra ("pclass2"). There are at least two other ways to specify this class proportion parameter, each with benefits and drawbacks. One could create two free parameters named "pclass1" and "pclass2" and constrain them using the `mxConstraint` function. This approach is relatively straightforward, but comes at the expense of standard errors. Alternatively, one could omit the algebra and fix "pclass2" to a specific value. This would make model specification easier, but the resulting "pclass1" parameter would not be scaled as a probability.

Finally, we can specify the mixture model. We must first specify the model's -2 log likelihood function defined as:

$$-2LL = -2 * \sum \log(p_1 l_{1i} + p_2 l_{2i})$$

This is specified using an mxAlgebra function, and used as the argument to the mxAlgebraObjective function. Then the objective 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.

```
algObj <- mxAlgebra(-2*sum(
          log(pclass1%x%Class1.objective + pclass2%x%Class2.objective)),
          name="mixtureObj")

obj <- mxAlgebraObjective("mixtureObj")

gmm <- mxModel("Growth Mixture Model",
         mxData(
         observed=myGrowthMixtureData,
         type="raw"
    ),
    class1, class2,
    classP, classA,
    algObj, obj
        )

gmmFit <- mxRun(gmm)

summary(gmmFit)
```

Multiple Runs

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.

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    vari2     cov2    vars2   m
#           0.2      1.0      1.0      0.4      1.0      0.0     -1.0      1.0      0.5      1.0
#       means2
#           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. A for loop repeatedly generates starting values (from a set of uniform distributions using runif), runs the model with those starting values and places the starting values, final estimates and fit statistics in the input, output and fit matrices, respectively.

```
trials <- 20

omxGetParameters(gmm)

parNames <- names(omxGetParameters(gmm))

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, 4)
dimnames(fit) <- list(c(1: trials), c("Minus2LL", "Status", "Iterations", "pclass1"))

for (i in 1: trials){
        cp <- runif(1, 0.1, 0.9) # class probability
        v  <- runif(5, 0.1, 5.0) # variance terms
        cv <- runif(2,-0.9, 0.9) # covariances (as correlations)
        m  <- runif(4,-5.0, 5.0) # means
        cv <- cv*c(sqrt(v[2]*v[3]), sqrt(v[4]*v[5])) #rescale covariances

        temp1 <- omxSetParameters(gmm,
                labels=parNames,
                values=c(
                        cp, # class probability
                        v[1],
                        v[2], cv[1], v[3], m[1], m[2],
                        v[4], cv[2], v[5], m[3], m[4]
                        )
                        )

                        temp1@name <- paste("Starting Values Set", i)

                        temp2 <- mxRun(temp1, unsafe=TRUE, suppressWarnings=TRUE)

                        input[i,] <- omxGetParameters(temp1)
                        output[i,] <- omxGetParameters(temp2)
                        fit[i,] <- c(
                        temp2@output$Minus2LogLikelihood,
                        temp2@output$status[[1]],
                        temp2@output$iterations,
                        temp2@output$estimate[1]
                        )
                }
```

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
#      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.

Multicore Estimation

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/repoview/1/trunk/demo/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.

CHAPTER

FOUR

ADVANCED CONCEPTS

4.1 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 path of 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"),
    mxMLObjective(covariance="R", dimnames=manifestVars),
    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, an optional prefix to the checkpoint filename, the decision to save based on minutes or optimizer iterations, and the checkpoint interval in either 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 when the checkpoint file was created:

```
factorFit <- mxRun(factorModel, checkpoint = TRUE)
factorRestore <- mxRestore(factorModel, chkpt.directory = directory)
```

The checkpoint directory 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.

4.2 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()
```

4.2.1 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) <- omxExtractNames(submodels)
topModel@submodels <- submodels
```

4.2.2 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='')
        model@data@observed <- randomCov(nObs, nVar, chl, dn)
        model@name <- 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(name="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),
               mxMLObjective(covariance='preCov'),
               independent = TRUE)

topModel <- mxModel(name = 'container')

submodels <- lapply(1:nReps, createNewModel, 'stErrSim', template)

names(submodels) <- omxExtractNames(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)
```

CHAPTER

FIVE

CHANGES IN OPENMX

5.1 Release 1.0.0-1448 (September 30, 2010)

- added missing entries to demo 00INDEX file

5.2 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

5.3 Release 0.9.1-1421 (September 12, 2010)

- fixed a bug in -2 LL calculation in RAM models with definition variables and raw data

5.4 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

5.5 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.

5.6 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

5.7 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

5.8 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

5.9 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'

5.10 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

5.11 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

5.12 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

5.13 Release 0.3.0-1217 (Apr 20, 2010)

5.13.1 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()

5.13.2 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

5.13.3 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

5.13.4 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)

5.14 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

5.15 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

5.16 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

5.17 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.

5.18 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

5.19 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.

5.20 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

5.21 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

5.22 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

5.23 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

5.24 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.

5.25 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

5.26 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

5.27 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 byrow=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).

5.28 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.

5.29 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

SIX

REFERENCE

- OpenMx R documentation

CHAPTER

SEVEN

INDICES AND TABLES

- *Search Page*