OpenMx Documentation

Release 0.1

OpenMx Development Team

October 07, 2009

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

1.1 Tutorial

1.1.1 Prerequisites

Congratulations! You have decided to check out OpenMx, the open source version of the statistical modeling package Mx, rewritten in R. Before we get started, let's make sure you have the software installed and ready to go. You need:

- R
- OpenMx

1.1.2 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. 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=`). The matrix **A** will be stored as the object 'A'.

```
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 (1) the sum of the matrices **A** and **B**, (2) the element by element multiplication (Dot product) of **A** and **B**, (3) the transpose of matrix **A**, and the (4) outer and (5) inner products of the matrix **A**, using regular matrix multiplication, i.e.:

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

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

```
mxAlgebra(
        A + B,
        name='q1'
)
mxAlgebra(
        A * A,
        name='q2'
)
mxAlgebra(
        t(A),
        name='q3'
)
mxAlgebra(
        A %*% t(A),
        name='q4'
)
mxAlgebra(
        t(A) %*% A,
        name='q5'
)
```

For the algebras to be evaluated, they become arguments of the `mxModel` command, as do the defined matrices, each separated by comma's. The model, which is here given the name 'algebraExercises', is then executed by the `mxRun` command, as shown in the full code below:

```
require(OpenMx)

algebraExercises <- mxModel(
    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(A+B, name='q1'),
            mxAlgebra(A*A, name='q2'),
            mxAlgebra(t(A), name='q3'),
    mxAlgebra(A%*%t(A), name='q4'),
    mxAlgebra(t(A)%*%A, name='q5'))

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

As you notice, we added some lines at the end to generate the desired output. 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 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.

1.1.3 Optimization Script

When collecting data to test a specific hypothesis, one of the first things one typically does is checking the basic descriptive statistics, such as the means, variances and covariances/correlations. We could of course use basic functions in R, i.e., *meanCol(Data)* or *cov(Data)* to perform these operations. However, if we want to test specific hypotheses about the data, 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 the that 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 variables in `selVars` which is used in the `dimnames` statement later on. 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 the data using SEM, we estimate a saturated model with free means, free variances and a covariance. Let's start with specifying the mean vector. We use the `mxMatrix` command, provide the `type`, here `Full`, the number of rows and columns, respectively 1 and 2, the specification of free/fixed parameters, the starting values, the dimnames 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)`. The `dimnames` are a type of label that is required to recognize the expected mean vector and expected covariance matrix and match up the model with the data. For a mean vector, the first element is `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`. 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, 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 `c('X','Y')` (previously defined as `selVars`) as `dimnames` for the rows and columns, 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".

```
mxMatrix(
    type="Lower",
    nrow=2,
    ncol=2,
    free=TRUE,
    values=.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, generated using the `mxMatrix` and `mxAlgebra` commands as “expCov”, and the expected means vectors, generated using the `mxMatrix` command as “expMeans”.

```
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 object ‘bivCorModel’. This object becomes the argument of the `mxRun` command, which evaluates the model and provides output - if the model ran successfully - using the following command.

```
bivCorFit <- mxRun(bivCorModel)
```

We can request various parts of the output to inspect by referring to them by the name of the object resulting from the `mxRun` command, i.e. `bivCorFit`, followed by the name of the objects corresponding to the expected mean vector, i.e. `[['ExpMean']]`, and covariance matrix, i.e. `[['ExpCov']]`, in quotes and double square brackets, followed by `@values`. The command `mxEval` can also be used to extract relevant information, such as the likelihood, (`objective`) where the first argument of the command is the object of interest and the second the object obtaining the results.

```
EM <- bivCorFit[['expMean']]@values
EC <- bivCorFit[['expCov']]@values
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 parameter estimates and goodness-of-fit statistics can also 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 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,
                dimnames=list(selVars, selVars),
                name="Chol"
    )
```

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.

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

1.1.4 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 design is that the MZ and DZ twins shared 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 the data following 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.

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 0.65, 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(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
    MZ <- mvrnorm (1000, c(0,0), matrix(c(1,rMZ,rMZ,1),2,2))
    DZ <- mvrnorm (1000, c(0,0), matrix(c(1,rDZ,rDZ,1),2,2))

    selVars <- c('t1','t2')
    dimnames(DataMZ) <- list(NULL,selVars)
    dimnames(DataDZ) <- list(NULL,selVars)
    summary(DataMZ)
    summary(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. 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),
                dimnames=list(NULL,selVars),
                name="expMeanMZ"),
        mxMatrix(
                type="Lower",
                nrow=2,
                ncol=2,
                free=TRUE
                values=.5,
                dimnames=list(NULL, selVars),
                name="CholMZ"),
        mxAlgebra(
                CholMZ %*% t(CholMZ),
                name="expCovMZ",
                dimnames=list(selVars, selVars)),
        mxData(
                DataMZ,
                type="raw"),
        mxFIMLObjective(
                "expCovMZ",
                "expMeanMZ"))
```

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 go to 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, the argument names `dimnames=` and `name=` are included to refer to the right arguments. 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), dimnames=list(NULL, selVars), name="expMeanDZ"),
    mxMatrix("Lower", 2, 2, T, .5, dimnames=list(NULL, selVars), name="CholDZ"),
    mxAlgebra(CholDZ %*% t(CholDZ), name="expCovDZ", dimnames=list(selVars, selVars)),
    mxData(DataDZ, type="raw"),
    mxFIMLObjective("expCovDZ", "expMeanDZ")),
```

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), dimnames=list(NULL, selVars), name="expMeanMZ"),
                mxMatrix("Lower", 2, 2, T, .5, dimnames=list(NULL, selVars), name="CholMZ"),
                mxAlgebra(CholMZ %*% t(CholMZ), name="expCovMZ", dimnames=list(selVars, selVars)),
                mxData(DataMZ, type="raw"),
                mxFIMLObjective("expCovMZ", "expMeanMZ")),
        mxModel("DZ",
                mxMatrix("Full", 1, 2, T, c(0,0), dimnames=list(NULL, selVars), name="expMeanDZ"),
                mxMatrix("Lower", 2, 2, T, .5, dimnames=list(NULL, selVars), name="CholDZ"),
                mxAlgebra(CholDZ %*% t(CholDZ), name="expCovDZ", dimnames=list(selVars, selVars)),
                mxData(DataDZ, type="raw"),
                mxFIMLObjective("expCovDZ", "expMeanDZ")),
        mxAlgebra(MZ.objective + DZ.objective, name="twin"),
        mxAlgebraObjective("twin"))
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.

```
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,
    mxModel("MZ",
        mxMatrix("Full", 1, 2, T, 0, "mMZ", dimnames=list(NULL, selVars), name="expMeanMZ"),
    mxModel("DZ",
        mxMatrix("Full", 1, 2, T, 0, "mDZ", dimnames=list(NULL, selVars), name="expMeanDZ"))
twinSatFitSub1 <- mxModel(twinSatModelSub1)
```

If we want to test if we can equate both means and variances 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,
    mxModel("MZ",
        mxMatrix("Full", 1, 2, T, 0, "mean", dimnames=list(NULL, selVars), name="expMeanMZ"),
        mxMatrix("Lower", 2, 2, T, .5, labels= c("var","MZcov","var"),
            dimnames=list(NULL, selVars), name="CholMZ"),
    mxModel("DZ",
        mxMatrix("Full", 1, 2, T, 0, "mean", dimnames=list(NULL, selVars), name="expMeanDZ"),
        mxMatrix("Lower", 2, 2, T, .5, labels= c("var","DZcov","var"),
            dimnames=list(NULL, selVars), name="CholDZ"))
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
```

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",
        mxMatrix("Full", 1, 2, T, 20, "mean", dimnames=list(NULL, selVars), name="expMean"),
        # Matrix expMean for expected mean vector for MZ and DZ twins
        mxMatrix("Full", nrow=1, ncol=1, free=TRUE, values=.6, label="a", name="X"),
        mxMatrix("Full", nrow=1, ncol=1, free=TRUE, values=.6, label="c", name="Y"),
        mxMatrix("Full", nrow=1, ncol=1, free=TRUE, values=.6, label="e", name="Z"),
        # Matrices X, Y, and Z to store the a, c, and e path coefficients
        mxMatrix("Full", nrow=1, ncol=1, free=FALSE, values=.5, name="h"),
        mxAlgebra(X * t(X), name="A"),
        mxAlgebra(Y * t(Y), name="C"),
        mxAlgebra(Z * t(Z), name="E"),
        # Matrixes A, C, and E to compute A, C, and E variance components
        mxAlgebra(rbind(cbind(A+C+E    , A+C),
        cbind(A+C      , A+C+E)), dimnames = list(selVars, selVars), name="expCovMZ"),
        # Matrix expCOVMZ for expected covariance matrix for MZ twins
        mxAlgebra(rbind(cbind(A+C+E    , h%x%A+C),
        cbind(h%x%A+C , A+C+E)), dimnames = list(selVars, selVars), name="expCovDZ"),
        # Matrix expCOVMZ for expected covariance matrix for DZ twins
        mxModel("MZ",
                mxData(DataMZ, type="raw"),
                mxFIMLObjective("twinACE.expCovMZ", "twinACE.expMean")),
        mxModel("DZ",
                mxData(DataDZ, type="raw"),
                mxFIMLObjective("twinACE.expCovDZ", "twinACE.expMean")),
        mxAlgebra(MZ.objective + DZ.objective, name="twin"),
        mxAlgebraObjective("twin"))
twinACEFit <- mxRun(twinACEModel)
```

Relevant output can be generate 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(expCovMZ, twinACEFit)
        DZc <- mxEval(expCovDZ, twinACEFit)
        M   <- mxEval(expMean, twinACEFit)
#Retrieve the A, C, and E variance components
        A <- mxEval(A, twinACEFit)
        C <- mxEval(C, twinACEFit)
        E <- mxEval(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 <- mxModel(twinACEModel,
    mxMatrix("Full", nrow=1, ncol=1, free=F, values=0, label="c", name="Y"))
twinAEFit <- mxRun(twinAEModel)
```

We discuss twin analysis examples in more detail in the example code.

1.2 Two Model Styles - Two Data Styles

In this first detailed example, we introduce the different styles available to specify models and data. There are two main approaches to specifying models: (i) paths specification and (ii) matrix specification. We will go through all the examples in both approaches, so you can choose which fits your style better, or check them both out to get a sense of their advantage/disadvantages. The 'path specification' model style translates path diagrams into OpenMx code; the 'matrix specification' model style relies on matrices and matrix algebra to produce OpenMx code. For each of the two approaches, the data may come in (a) summary format, i.e. covariance matrices and possibly means, or (b) raw data format. We will illustrate both, as arguments of functions may differ. Thus, we will here describe the same example four different ways:

- i.a Path Specification - Covariance Matrices

- i.b Path Specification - Raw Data
- ii.a Matrix Specification - Covariance Matrices
- ii.b Matrix Specification - Raw Data

Our first example is fitting a simple model to one variable to estimate its mean and variance. This is also referred to as fitting a saturated model. We start with a univariate example, and also work through a bivariate example which differs in minor ways from the univariate one, as it forms the basis for later examples.

1.2.1 Univariate Saturated Model

The four versions of univariate example are available in the following files:

- UnivariateSaturated_PathCov.R
- UnivariateSaturated_PathRaw.R
- UnivariateSaturated_MatrixCov.R
- UnivariateSaturated_MatrixRaw.R
- UnivariateSaturated.R

The last file includes all four example in one. The bivariate examples are available in the following files:

- BivariateSaturated_PathCov.R
- BivariateSaturated_PathRaw.R
- BivariateSaturated_MatrixCov.R
- BivariateSaturated_MatrixRaw.R
- BivariateSaturated_MatrixCovCholesky.R
- BivariateSaturated_MatrixRawCholesky.R
- BivariateSaturated.R

Note that we have additional version of the matrix-style examples which use a Cholesky decomposition to estimate the expected covariance matrices, which is preferred to directly estimation the symmetric 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.

```
#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 the normal distribution function. 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.

1.2.2 Covariance Matrices and Path-style Input

Model Specification

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. 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 have a matching 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 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 distinguished with one-head arrows (if arrows=1) or 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 `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.

```
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, `matrix????`, 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 ?]

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

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. [How do we specify other$Saturated in mxEval?]

```
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 use 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.

```
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=matrix(var(testData),1,1),
    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 was as the expected variance before.

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

1.2.3 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. #With RAM path specification, and raw data input, OpenMx has a default model for the means, in which every observed variable has a free parameter for its mean [NB this should change in future versions to require means model]. 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 modified in any respect. So to change the above 'univSatModel1' can be effected this way:

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

This model can be run as usual with an `mxRun` command: .. code-block:: r

univRawFit1 <- mxRun(univSatModel1)

Note The output generated from this model now includes the expected mean, the expected covariance matrix and -2 times the log-likelihood of the data.

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

1.2.4 Covariance Matrices and Matrix-style Input

We now specify essentially the same models with matrices. Starting with the model fitted to the summary covariance matrix, we need a specify one matrix for the expected covariance matrix. We use the `mxMatrix` command for this. The first argument is its type, which is symmetric for a covariance matrix. The second and third arguments are the number of rows (`nrow`) and columns (`ncol`). The `free` and `values` command work in the same way as in the path specification. If only one element is given, it is applied to all the elements in 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 code to specify the model includes four commands, (i) `mxModel`, (ii) `mxMatrix`, (iii) `mxData` and (iv) `mxMLObjective. The ``mxData` is the same for paths and matrices specifications. A different objective function is used, namely the `mxMLObjective` command which takes one argument, the expression/name of the expected covariance matrix, which we specified in the `mxMatrix` command.

```
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(
        "expCov", dimnames=selVars)
    )
univSatFit3 <- mxRun(univSatModel3)
```

A means vector can also be added here as part of the input summary statistics (as the fourth argument of the `mxData` command). In that case, a second `mxMatrix` command is used to specify the expected mean vector, which is of type 'Full', has 1 row and 1 column, is assigned 'free' with start value 0, dimnames for the column, and the name "expMean". The second change is an additional argument to the `mxMLObjective` function for the expected mean, here "expMean".

```
....
mxMatrix(
    type="Full",
    nrow=1,
    ncol=1,
    free=T,
    values=0,
    name="expMean"
),
mxData(
    observed=var(testData),
    type="cov",
    numObs=1000,
    means=mean(testData)
),
mxMLObjective(
    "expCov",
    "expMean",
    dimnames=selVars
)
```

1.2.5 Raw Data and Matrix-style Input

Finally, if we want to use the matrix specification with raw data, we again specify two matrices using the `mxMatrix` command, one for the expected covariance matrix and one for the expected mean vector, in the same way as before. The `mxData` command directly read the raw data from a matrix or data.frame and the `mxFIMLObjective` command is used to evaluate the likelihood of the data using FIML. This function also takes two arguments, one for the expected covariance matrix and one for the expected mean.

```
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(
        "expCov",
        "expMean",
        dimnames=selVars)
    )
```

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

1.2.6 Bivariate Saturated Model

Rarely will we analyze a single variable. As soon as a second variable is added, not only can be then estimate two means and two variances, but also a covariance between the two variables.

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 .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)
```

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.

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 one. The second `mxPath` command specifies a two-headed arrow from ‘X’ to ‘Y’, 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 for fitting to raw data just require 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 .5 for the covariance

```
mxMatrix(
    type="Symm",
    nrow=2,
    ncol=2,
    free=T,
    values=c(1,.5,1),
    dimnames=list(selVars,selVars),
    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),
    dimnames=list(NULL,selVars)
    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's need 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 but now create a lower triangular matrix (by using a 2x2 full matrix and fixing the element above the diagonal to zero; note that the matrix type="lower" will be implemented later). We then use an `mxAlgebra` command to multiply this matrix, named 'Chol' with its transpose (R function t()). As this resulting matrix represents the expected covariance matrix, dimnames are required such that the matrix elements can be properly matched to the data.

```
mxMatrix(
    type="Full",
    nrow=2,
    ncol=2,
    free=c(T,T,F,T),
    values=c(1,.2,0,1),
    name="Chol"
)
mxAlgebra(
    Chol %*% t(Chol),
    name="expCov",
    dimnames=list(selVars,selVars)
)
```

CHAPTER

TWO

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 and the OpenMx library for R. 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.

2.1 Pass By Value (READ THIS)

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

2.2 Matrix Model Specification

```
require(OpenMx)

data(demoOneFactor)

factorModel <- mxModel(name = "One Factor")

matrixA <-  mxMatrix("Full", 5, 1, values=0.2, free=T, name="A")
matrixL <-  mxMatrix("Symm", 1, 1, values=1, free=F, name="L")
matrixU <-  mxMatrix("Diag", 5, 5, values=1, free=T, name="U")

algebraR <- mxAlgebra(A %*% L %*% t(A) + U, name="R")

objective <- mxMLObjective("R", dimnames = names(demoOneFactor))
data <- mxData(cov(demoOneFactor), type="cov", numObs=500)

factorModel <- mxModel(factorModel, matrixA, matrixL, matrixU,
    algebraR, objective, data)

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

Above is an example that creates 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. Let's break down what is happening in each section of this example.

2.2.1 Preamble

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

2.2.2 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.

2.2.3 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, 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 `model` stores a value that is a MxModel object with the name "One Factor".

2.2.4 Matrix Creation

The next three lines create three MxMatrix objects. The first argument declares the type of the matrix, the second argument declares the number of rows in the matrix, and the third argument declares the number of columns. The 'values' argument specifies the starting values in the matrix. The 'free' argument specifies whether a cell is a free or fixed parameter, and the 'name' argument specifies the name of the matrix. 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".

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

2.2.5 Algebra Creation

Lines 11-12 construct an expression for the expected covariance algebra. The first argument 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 (`?mxAlgebra`). The algebra expression refers to entities according to their names.

2.2.6 Objective Function Creation

Line 14 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 correspondance can be determined between the expected covariance matrix and the observed covariance matrix.

2.2.7 Data Source Creation

Line 15 constructs 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.

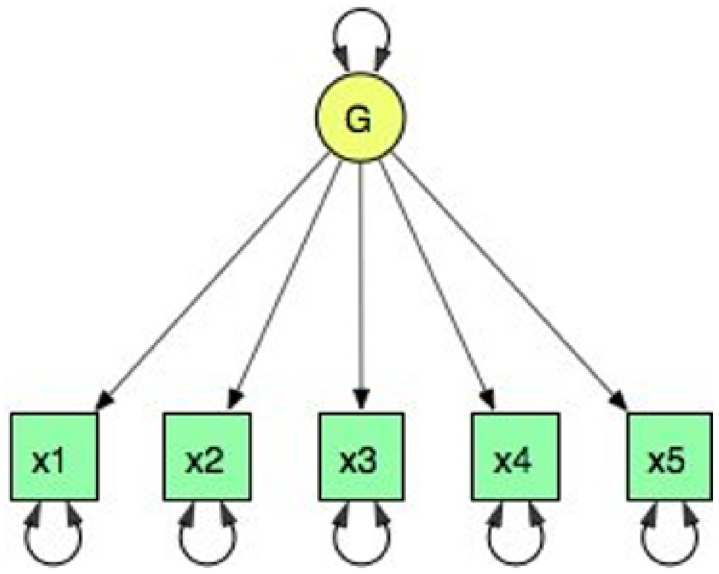
2.2.8 Model Population

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). In our example, we are populating the model with three matrices, an algebra, an objective function, and a data source. Lines 5, 17, and 18 could have been combined with the following call: `factorModel <- mxModel(matrixA, matrixL, matrixU, algebraR, objective, data, name = "One Factor")`.

2.2.9 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 cells of the matrices of the model assigned to their final values. The summary function is a convenient method for displaying the highlights of a model after it has been executed.

2.3 Path Model Specification



```
require(OpenMx)

data(demoOneFactor)

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

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

summary(mxRun(factorModel))
```

We will now re-create the model from the previous section, but this time we will use a RAM-style specification technique. Let's break down what is happening in each section of this example.

2.3.1 Preamble

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

2.3.2 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.

2.3.3 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 and a vector for latent variables. In this case the manifest variables are `c("x1", "x2", "x3", "x4", "x5")` and the latent variable is `c("G")`.

2.3.4 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. '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.

CHAPTER

THREE

EXAMPLES, PATH SPECIFICATION

3.1 Regression, Path Specification

Our next 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 are available; one with raw data, and one where the data is supplied as a covariance matrix and vector of means. These examples are available in the following files:

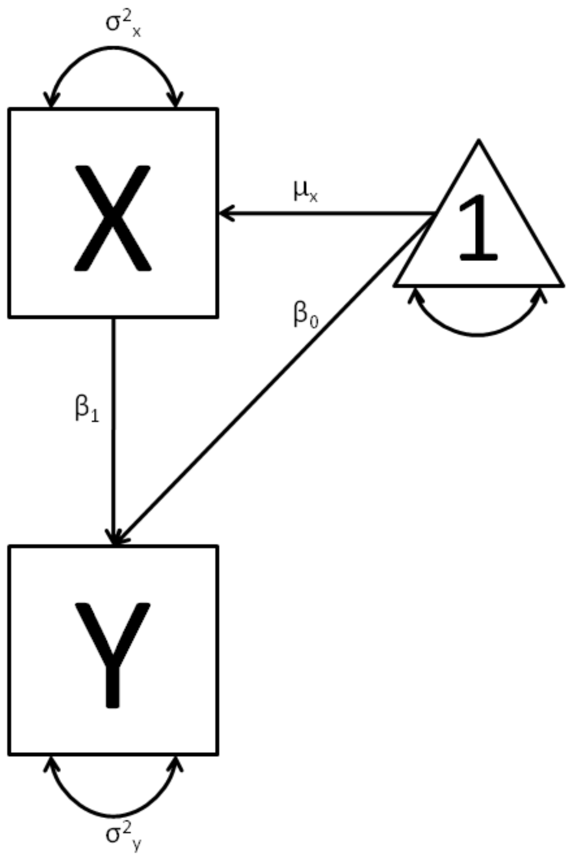
- SimpleRegression_PathCov.R
- SimpleRegression_PathRaw.R
- MultipleRegression_PathCov.R
- MultipleRegression_PathRaw.R
- MultivariateRegression_PathCov.R
- MultivariateRegression_PathRaw.R

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

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 variance) 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 put 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.

```
myRegDataRaw <- read.table("myRegData.txt",header=TRUE)
```

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

```
> names(myRegDataRaw)
[1] "w" "x" "y" "z"
```

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'll 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 also 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'll 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'll then run.

```
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, 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=SimpleRegMeans
)
```

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.

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.

```
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"`.

```
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.

```
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, 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)
```

```
uniRegFit@output
```

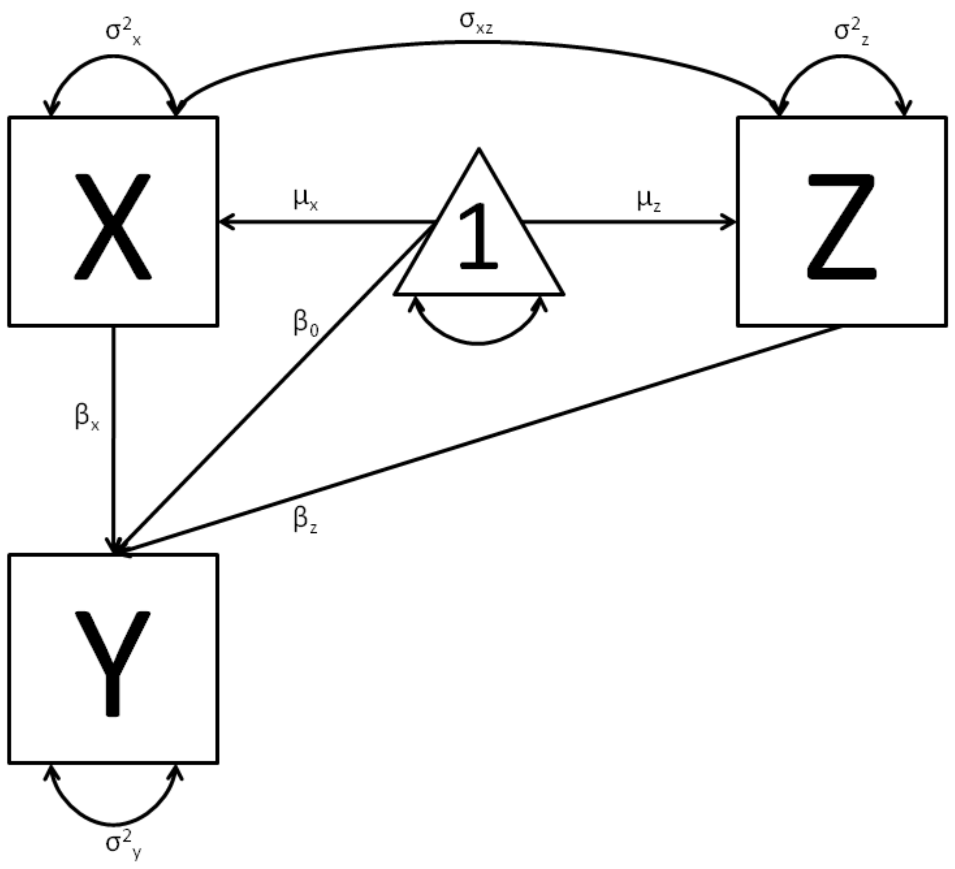
The `output` slot contains a great deal of information, including parameter estimates and information about the matrix operations underlying our model. A more parsimonious report on the results of our model can be viewed using the `summary` function, as shown here.

```
summary(uniRegFit)
```

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, 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="y",
        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 variaces. 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`.

```
mxPath(
    from="x",
    to="y",
    arrows=2,
    free=TRUE,
    values=0.5,
    labels="covxz"
),
```

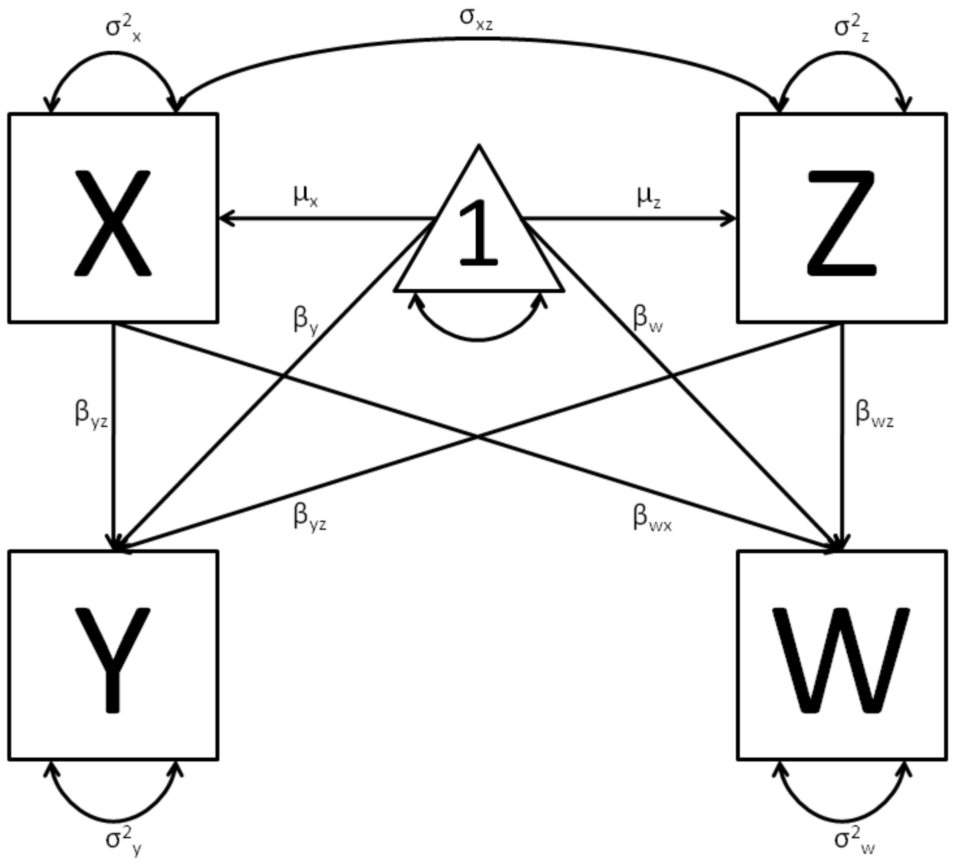
The third and fourth `mxPath` functions mirror the second and third `mxPath` functions from our simple regression, defining the regressions of y on both x and z as well as the means and intercepts of our model.

The model is run and output is viewed just as before, using the `mxRun` function, `@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".

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



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.

```
myRegDataRaw<-read.table("myRegData.txt",header=TRUE)

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),
        labels=c("residualw", "varx", "residualy", "varz")
    ),
    # covariance of x and z
    mxPath(
        from="x",
        to="y",
        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),
        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 link for matrix specification of these models.

3.2 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:

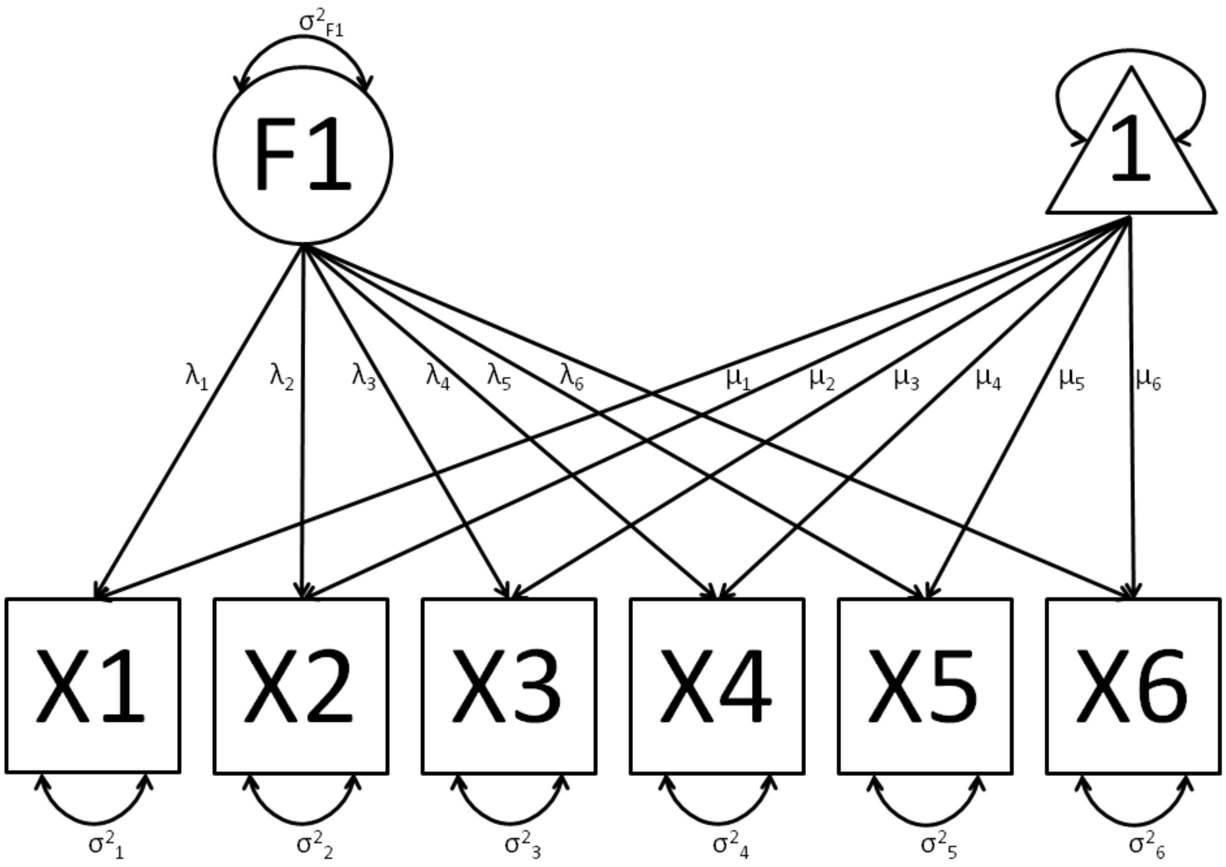
- OneFactorModel_PathCov.R
- OneFactorModel_PathRaw.R
- TwoFactorModel_PathCov.R
- TwoFactorModel_PathRaw.R

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

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 put 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.

```
myFADataRaw <- read.table("myFAData.txt",header=TRUE)

> names(myFADataRaw)
[1] "x1" "x2" "x3" "x4" "x5" "x6" "y1" "y2" "y3"

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", "x

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'll then run.

```
require(OpenMx)

oneFactorModel<-mxModel("Common Factor Model - Path",
    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 for the model and a `type="RAM"` argument. The name for the model may be omitted, or may be specified an 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 raw data specification 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",
    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 a name for the latent variable, 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 specifying 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`.

```
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.

```
mxPath(from="F1",
    arrows=2,
    free=TRUE,
    values=1,
    labels ="varF1"
)
```

Next come the factor loadings. These are specified as assymetric 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`.

```
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.

```
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)
)
```

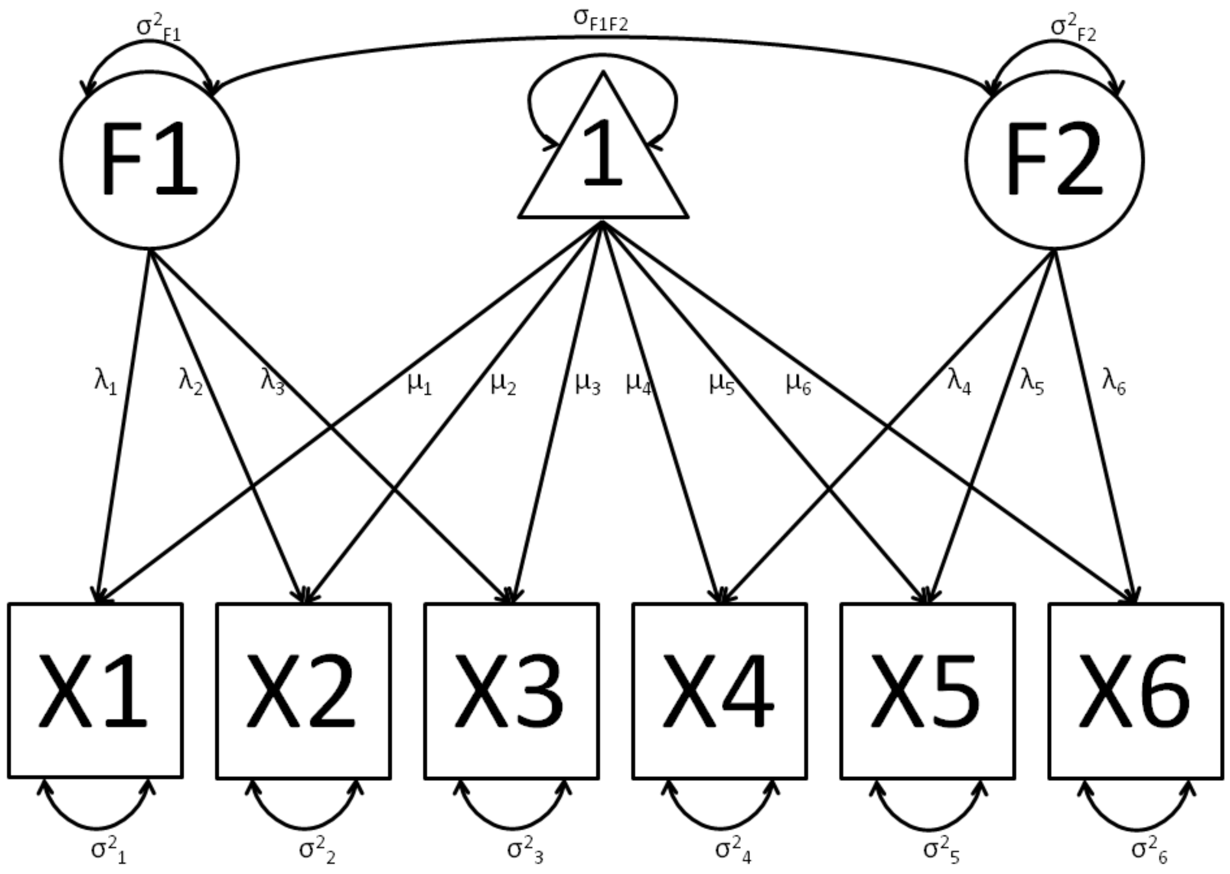
The model can now be run using the `mxRun` function, and the output of the model can be accessed from the `output` slot of the resulting model. A summary of the output can be reached using `summary()`.

```
oneFactorFit <- mxRun(oneFactorModel)

oneFactorFit@output

summary(oneFactorFit)
```

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 $x_1' - : math : `x_3$ and $y_1' - : math : `y_3$ 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'll 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'll 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", "y

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",
    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'll 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 F2 (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.

```
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 link for matrix specification of these models.

3.3 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:

- LGC_PathCov.R
- LGC_PathRaw.R

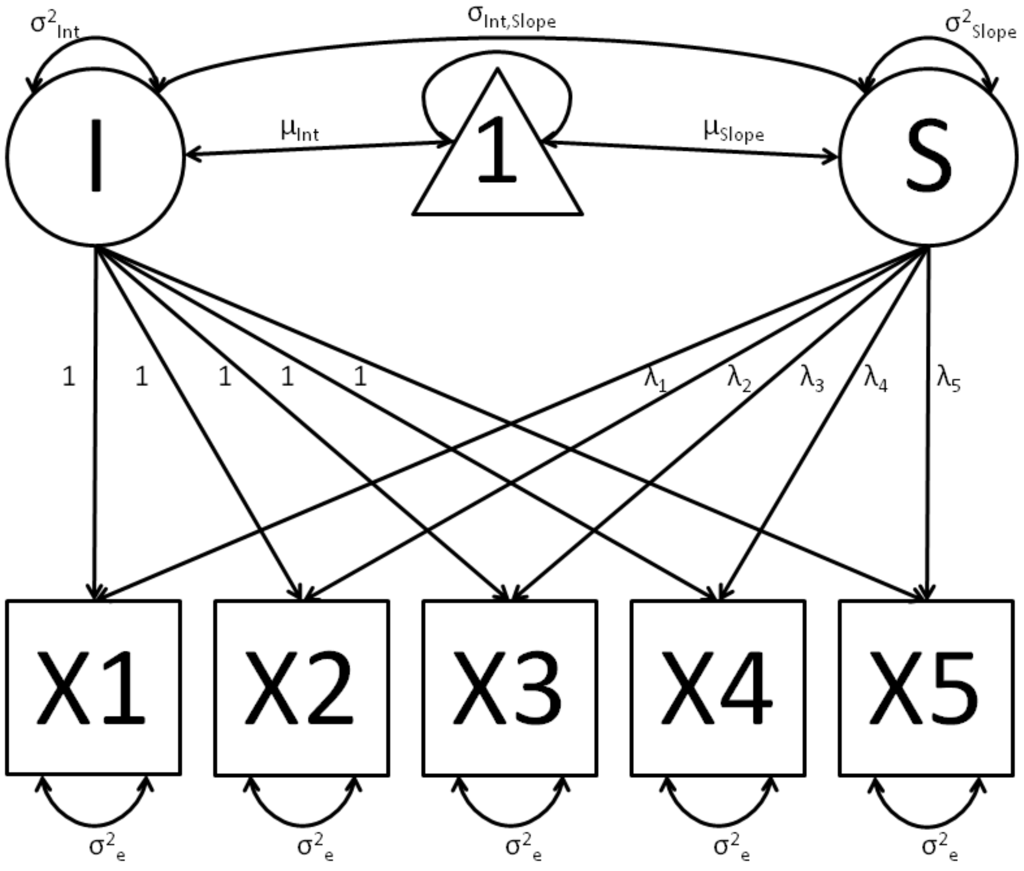
A parallel version of this example, using matrix-centric specification of models rather than paths, can be found here link.

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:

. math:: **nowrap**

begin{eqnarray*} x_{ij} = Intercept_{i} + lambda_{j} * Slope_{i} + epsilon_{i} end{eqnarray*}



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'll 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.

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

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"))
      )
```

myLongitudinalDataMean <- 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'll 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 entites 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 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)

growthCurveFit@output

summary(growthCurveFit)

These models may also be specified using matrices instead of paths. See link for matrix specification of these models.

3.4 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 to estimated 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:

- BivariateHeterogeneity_PathRaw.R

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

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.

Data

For this example we simulated two datasets ('xy1' and 'xy2') each with zero means and unit variances, one with a correlation of .5, and the other with a correlation of .4 with 1000 subjects each. 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

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. `mxModel` commands are unlimited in the number of arguments.

```
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 free parameters for the variances and the covariance using two-headed arrows to generate the expected covariance matrix. Single-headed arrows from 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. For more details, see example 1.

```
#Fit Heterogeneity Model
bivHetModel <- mxModel("bivHet",
    mxModel("group1",
        manifestVars= selVars,
        mxPath(
            from=c("X", "Y"),
            arrows=2,
            free=T,
            values=1,
            lbound=.01,
            labels=c("vX1","vY1")
        ),
        mxPath(
            from="X",
            to="Y",
            arrows=2,
            free=T,
            values=.2,
            lbound=.01,
            labels="cXY1"
        ),
        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,
        mxPath(
            from=c("X", "Y"),
            arrows=2,
            free=T,
            values=1,
            lbound=.01,
            labels=c("vX2","vY2")
        ),
        mxPath(
            from="X",
            to="Y",
            arrows=2,
            free=T,
            values=.2,
            lbound=.01,
            labels="cXY2"
        ),
        mxPath(
            from="one",
            to=c("X", "Y"),
            arrows=1,
            free=T,
            values=0,
            labels=c("mX2", "mY2")
        ),
        mxData(
            observed=xy2,
            type="raw",
        ),
        type="RAM"
        ),              ), ....
```

As a result, 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"  "zero"
    Y "cXY1" "vY1"

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

in group2
    $S
      X       Y
    X "vX2"  "zero"
    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. 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.

```
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 specificiation, respectively in the matrices M and S for the two groups. OpenMx also puts the values for the expected means and covariances in 'means' and 'covariance' objects. We also print 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.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)
```

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.

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 Homnogeneity 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 S) for the expected means 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"   "zero"
    Y "cXY1" "vY1"

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

in group2
    $S
      X       Y
      X "vX1"   "zero"
      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.

```
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 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
```

3.5 Genetic Epidemiology, Path 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:

- UnivariateTwinAnalysis_PathRaw.R

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

3.5.1 ACE Model: a Twin Analysis

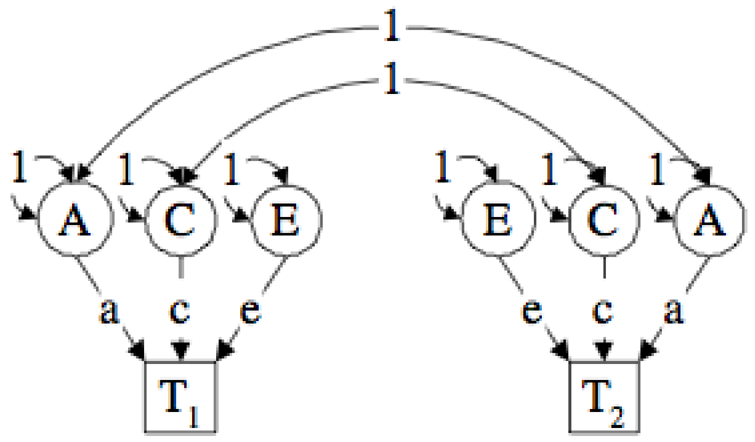
Data

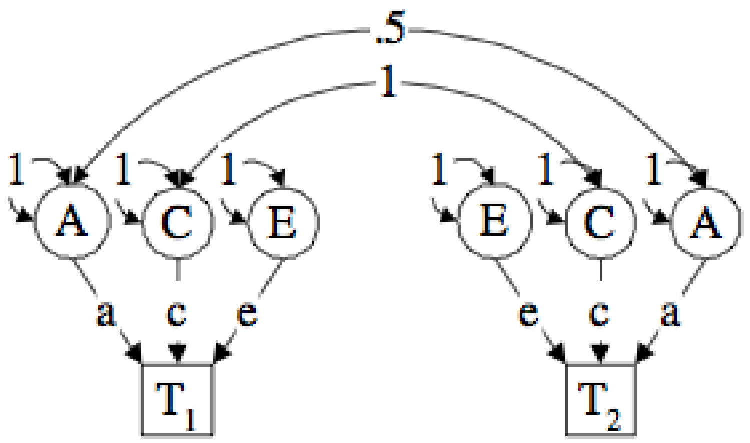
Let us assume you have collected data on a large sample of twin pairs for your phenotype of interest. For illustration purposes, we use Australian data on body mass index (BMI) which are saved in a text file 'myTwinData.txt'. We use R to read the data into a data.frame and to create two subsets of the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

```
require(OpenMx)

#Prepare Data
twinData <- read.table("myTwinData.txt", header=T, na.strings=".")
twinVars <- c('fam','age','zyg','part','wt1','wt2','ht1','ht2','htwt1','htwt2','bmi1','bmi2')
summary(twinData)
selVars <- c('bmi1','bmi2')
aceVars <- c("A1","C1","E1","A2","C2","E2")
mzfData <- as.matrix(subset(twinData, zyg==1, c(bmi1,bmi2)))
dzfData <- as.matrix(subset(twinData, zyg==3, c(bmi1,bmi2)))
```

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) ***bmi1*** and ***bmi2*** at the bottom 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. DZ twins share on average half their genes, 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 twin 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. The unique environmental factors are by definition uncorrelated between twins.





Let's go through each of the paths specification step by step. They will all form arguments of the `mxModel`, specified as follows. Given the diagrams for the MZ and the DZ group look rather similar, we start by specifying all the common elements which will then be shared with the two submodels for each of the twin types. Thus we call the first model 'share'.

```
#Fit ACE Model with RawData and Path-style Input
share <- mxModel("share",
    type="RAM",
```

Models specifying paths are translated into 'RAM' specifications for optimization, indicated by using the `type='RAM'`. 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. We start the path diagram specification by providing the names for the manifest variables in `manifestVars` and the latent varibles in `latentVars`. We use here the 'selVars' and 'aceVars' objects that we created before when preparing the data.

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

We start by specifying paths for the variances and means of the latent variables. This includes double-headed arrows from each latent variable back to itself, fixed at one, and single-headed arrows from the triangle (with a fixed value of one) to each of the latent variables, fixed at zero. 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. 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 .6 and appropriate labels. As the common environmental factors are by definition the same for both twins, we fix the correlation between **C1** and **C2** to one.

```
mxPath(
    from=aceVars,
    arrows=2,
    free=FALSE,
    values=1
),
mxPath(
    from="one",
    to=aceVars,
    arrows=1,
    free=FALSE,
    values=0
),
mxPath(
    from="one",
    to=selVars,
    arrows=1, free=TRUE,
    values=20,
    labels= c("mean","mean")
),
mxPath(
    from=c("A1","C1","E1"),
    to="bmi1",
    arrows=1,
    free=TRUE,
    values=.6,
    label=c("a","c","e")
),
mxPath(
    from=c("A2","C2","E2"),
    to="bmi2",
    arrows=1,
    free=TRUE,
    values=.6,
    label=c("a","c","e")
),
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 submodels which will be combined in 'twinACEModel'. So we have two `mxModel` statement within the "twinACE" model statement. Each of the two models are based on the previously specified "share" model by including it as its first argument. Then we add the path for the correlation between **A1** and **A2** which is fixed to one for the MZ group. 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, with the type specified explicitly. Given we use the path specification, the objective function uses RAM, thus `type='RAM'`. We also give it the model a name to refer back to it later when we need to add the objective functions. 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 are read in.

```
mzModel <- mxModel(share,
    mxPath(from="A1", to="A2", arrows=2, free=FALSE, values=1),
    mxData(mzfData, type="raw"),
    type="RAM", name="MZ")

dzModel <- mxModel(share,
    mxPath(from="A1", to="A2", arrows=2, free=FALSE, values=.5),
    mxData(dzfData, type="raw"),
    type="RAM", name="DZ")
```

Finally, both models need to be evaluated simultaneously. We first generate the sum of the objective functions for the two groups, using `mxAlgebra`, and then use that as argument of the `mxAlgebraObjective` command.

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

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(MZ.covariance, twinACEFit)
DZc <- mxEval(DZ.covariance, twinACEFit)
M <- mxEval(MZ.means, twinACEFit)
A <- mxEval(a*a, twinACEFit)
C <- mxEval(c*c, twinACEFit)
E <- mxEval(e*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 two `mxPath` commands. First, we call up the previous 'full' model and save it as a new model 'twinAEModel'. 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 can run this model in the same way as before and generate similar summaries of the results.

```
#Run AE model
mzModel <- mxModel(mzModel,
    mxPath(from=c("A1","C1","E1"), to="bmi1", arrows=1, free=c(T,F,T), values=c(.6,0,.6), label=c("a
    mxPath(from=c("A2","C2","E2"), to="bmi2", arrows=1, free=c(T,F,T), values=c(.6,0,.6), label=c("a

dzModel <- mxModel(dzModel,
    mxPath(from=c("A1","C1","E1"), to="bmi1", arrows=1, free=c(T,F,T), values=c(.6,0,.6), label=c("a
    mxPath(from=c("A2","C2","E2"), to="bmi2", arrows=1, free=c(T,F,T), values=c(.6,0,.6), label=c("a
twinAEModel <- mxModel(twinACEModel, mzModel, dzModel, name = "twinAE")

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) 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
```

3.6 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.

3.6.1 Mean Differences

The scripts are presented here

- DefinitionMeans_PathRaw.R
- DefinitionMeans_PathRaw.mx

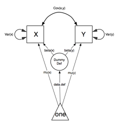
Statistical Model

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 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=n, c(1,2), Sigma)
group2<-mvrnorm(n=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)
y<-rbind(group1,group2)
dimnames(y)[2]<-list(c("x","y"))
def<-rep(c(1,0),each=n)
selvars<-c("x","y")
```

The objects y 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

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.

```
require(OpenMx)
defmeansmodel<-mxModel("Definition Means via Paths",
    type="RAM",
```

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 `"DefinitionMeans"`.

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. 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(y,def),
    type="raw"),
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 keword "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. 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.

```
mxPath(from=c("x","y"),
    arrows=2,
    free=TRUE,
    values=c(1,.1,1),
    labels=c("Varx","Vary")
), # variances
mxPath(from="x", to="y",
    arrows=2,
    free=TRUE,
    values=c(.1),
    labels=c("Covxy")
), # covariances
mxPath(from="one",
    to=c("x","y","DefDummy"),
    arrows=1,
    free=c(TRUE,TRUE,FALSE),
    values=c(1,1,1),
    labels =c("meanx","meany","data.def")
), # means
mxPath(from="DefDummy",
    to=c("x","y"),
    arrows=1,
    free=c(TRUE,TRUE),
    values=c(1,1),
    labels =c("beta_1","beta_2")
)) # moderator paths
```

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 `defmeansresult` contains matrices and algebras; here we are interested in the matrices, which can be seen with the `defmeansresult@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 we compute some summary statistics from the 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 we extract the parameter estimates and matrix algebra results from the model
Sigma<-defmeansresult@matrices$S@values[1:2,1:2]
Mu<-defmeansresult@matrices$M@values[1:2]
beta<-defmeansresult@matrices$A@values[1:2,3]

# Third, we 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)
```

CHAPTER

FOUR

EXAMPLES, MATRIX SPECIFICATION

4.1 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 with raw data, and one where the data is supplied as a covariance matrix and vector of means. These examples are available in the following files:

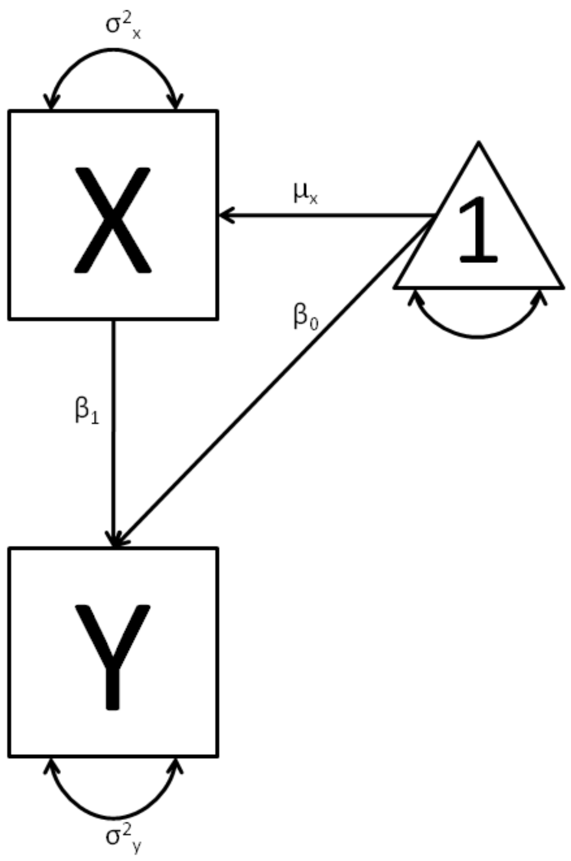
- SimpleRegression_MatrixCov.R
- SimpleRegression_MatrixRaw.R
- MultipleRegression_MatrixCov.R
- MultipleRegression_MatrixRaw.R
- MultivariateRegression_MatrixCov.R
- MultivariateRegression_MatrixRaw.R

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 link.

4.1.1 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 variance) 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 put 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.

```
myRegDataRaw <- read.table("myRegData.txt",header=TRUE)
```

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

```
> names(myRegDataRaw)
[1] "w" "x" "y" "z"
```

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

```
uniRegModel <- mxModel("Simple Regression - Matrix Specification",
    mxData(
      observed=SimpleRegRaw,
      type="raw"
    ),
    mxMatrix(
        type="Full",
        nrow=2,
        ncol=2,
        free=c(F, F,
               F, F),
        values=c(0, 0,
                 1, 0),
        labels=c(NA,     NA,
                "beta1", NA),
        byrow=TRUE,
        name="A"
    ),
    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"
    ),
    mxMatrix(
        type="Iden",
        nrow=2,
        ncol=2,
        name="F"
    ),
    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 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 `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=SimpleRegMeans
)
```

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"`.

```
mxMatrix(
    type="Full",
    nrow=2,
    ncol=2,
    free=c(F, F,
           F, 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. 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"`.

```
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. The `dimnames` provided at this matrix should match the names of the data, either the column names for raw data or the `dimnames` of covariance data. There are no free parameters in any F matrix.

```
mxMatrix(
    type="Iden",
    nrow=2,
    ncol=2,
    dimnames=list(c("x","y"),c("x","y")),
    name="F"
)
```

The fourth and final `mxMatrix` function specifies the M matrix. This matrix is used to specify the means and intercepts of our model. Exogenous or independent variables receive means, while endogenous or dependent variables get intercepts, or means conditional on regression on other variables. This matrix contains only one row. This matrix consists of two free parameters; the mean of x (labeled "meanx") and the intercept of y (labeled "beta0"). This matrix gives starting values of 1 for both parameters, and is named "M".

```
mxMatrix(
    type="Full",
    nrow=1,
    ncol=2,
    free=c(T, T),
    values=c(0, 0),
    labels=c("meanx", "beta0"),
    dimnames=list(NULL, c("x","y")),
    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)

uniRegFit@output
```

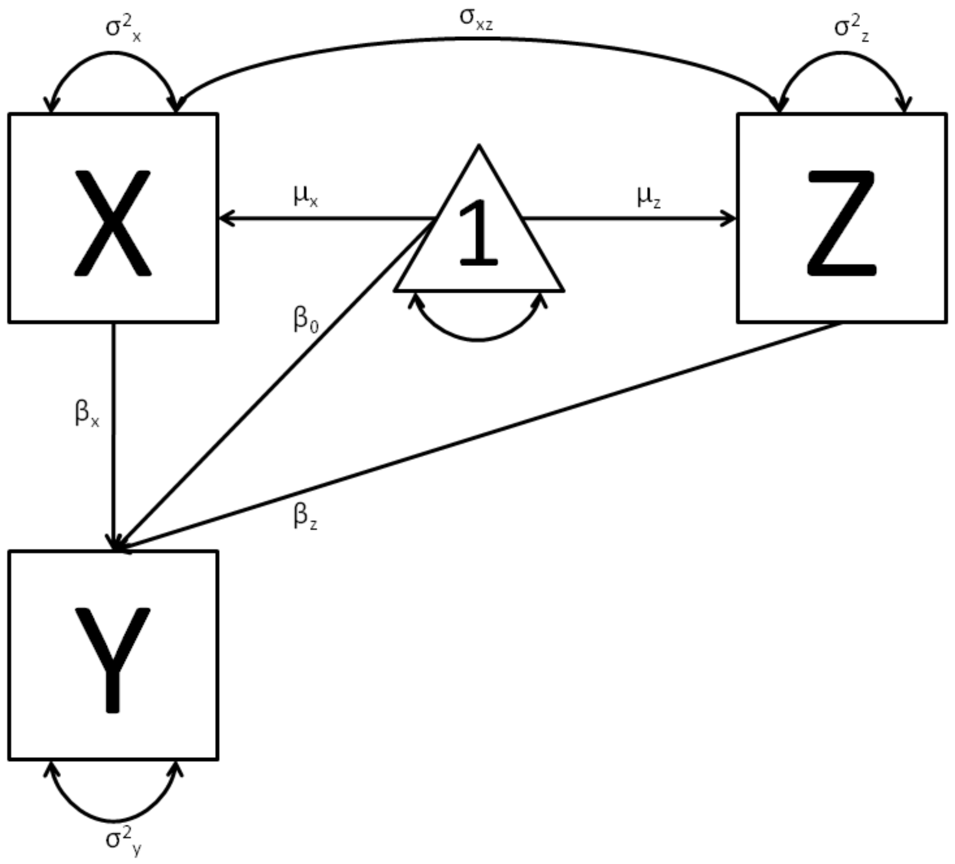
The `output` slot contains a great deal of information, including parameter estimates and information about the matrix operations underlying our model. A more parsimonious report on the results of our model can be viewed using the `summary` function, as shown here.

```
summary(uniRegFit)
```

4.1.2 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"),
    mxMatrix("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"),
    mxMatrix("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"),
    mxMatrix("Iden",
        nrow=3,
        ncol=3,
        name="F",
        dimnames = list(c("x","y","z"), c("x","y","z"))),
    mxMatrix("Full",
        nrow=1,
        ncol=3,
        values=c(0,0,0),
        free=c(T,T,T),
        labels=c("meanx","beta0","meanz"),
        dimnames = list(NULL, c("x","y","z")),
        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 accomodate 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.

```
mxMatrix("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")
```

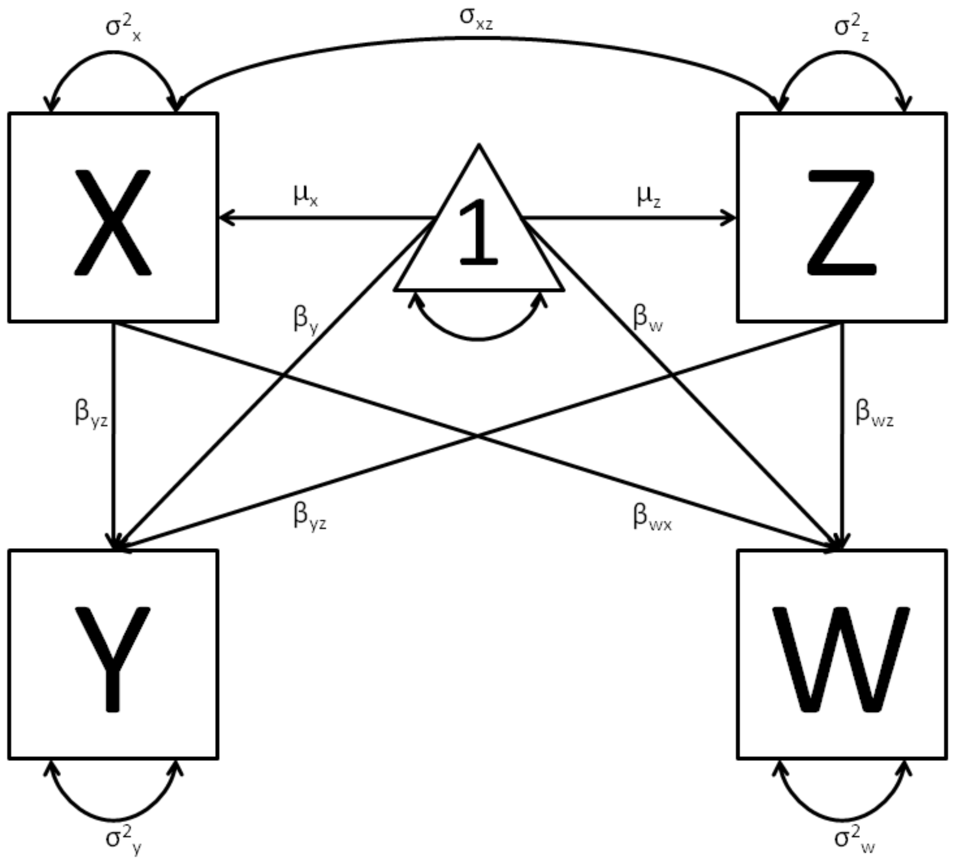
We've made a similar changes in the other matrices. The S matrix includes not only a variance term for the z variable, but also a covariance between the two independent variables. The F matrix still does not contain free parameters, but has expanded in size and made parallel changes in the `dimnames`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

4.1.3 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".

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



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.

```
myRegDataRaw<-read.table("myRegData.txt",header=TRUE)

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"),
    mxMatrix("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"),
    mxMatrix("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, 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"),
    mxMatrix("Iden",  nrow=4, ncol=4,
        dimnames=list(
            c("w","x","y","z"),
            c("w","x","y","z")),
        name="F"),
    mxMatrix("Full", nrow=1, ncol=4,
        values=c(0,0,0,0),
        free=c(T,T,T,T),
        labels=c("betaw","meanx","betay","meanz"),
        dimnames=list(
                  NULL,c("w","x","y","z")),
        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 link for matrix specification of these models.

4.2 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:

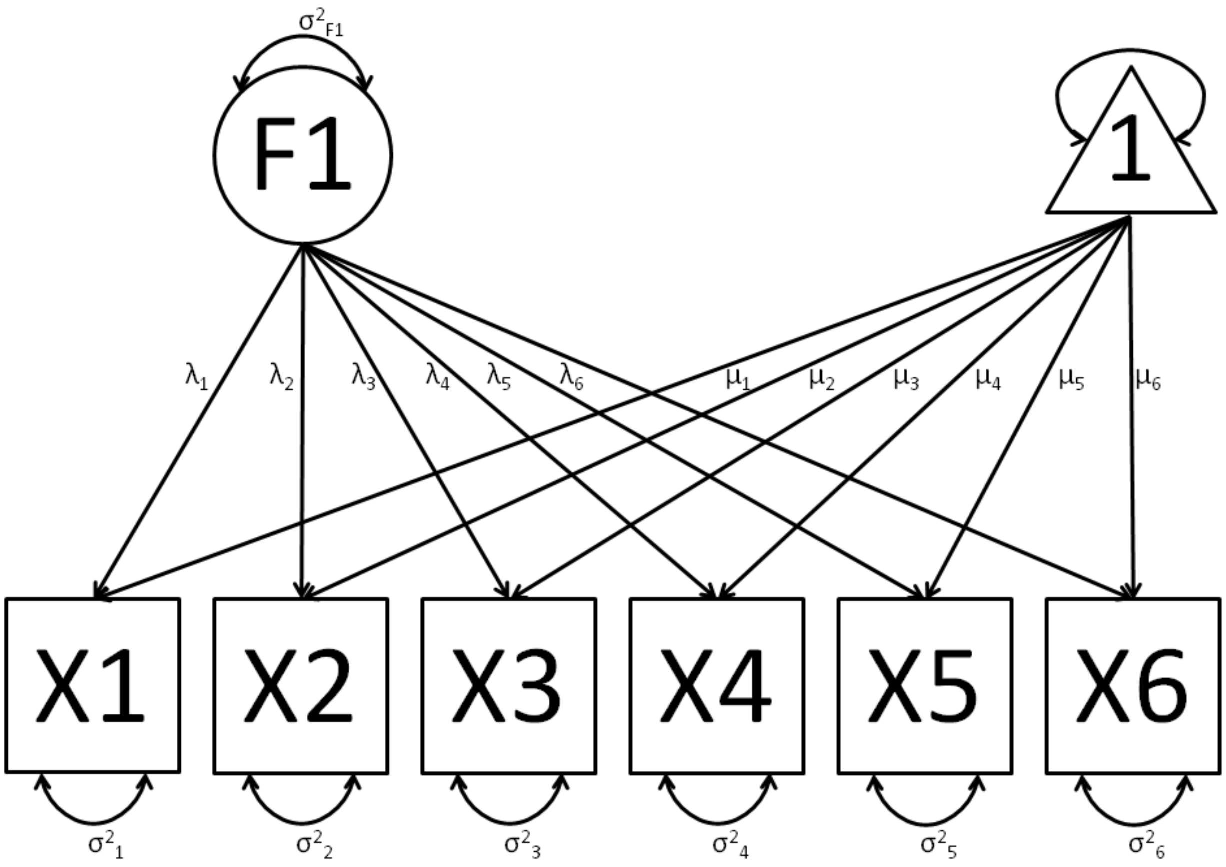
- OneFactorModel_MatrixCov.R
- OneFactorModel_MatrixRaw.R
- TwoFactorModel_MatrixCov.R
- TwoFactorModel_MatrixRaw.R

A parallel version of this example, using path-centric specification of models rather than matrices, can be found here link.

4.2.1 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 put 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.

```
myFADataRaw <- read.table("myFAData.txt",header=TRUE)

> names(myFADataRaw)
[1] "x1" "x2" "x3" "x4" "x5" "x6" "y1" "y2" "y3"

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", "x

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'll then run.

```
oneFactorModel<-mxModel("Common Factor Model - Matrix Specification",
      mxData(myFADataRaw, type="raw"),
      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"),
      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"),
      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,
          dimnames=list(
              c("x1","x2","x3","x4","x5","x6"),
              c("x1","x2","x3","x4","x5","x6","F1")),
          name="F"),
      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),
          dimnames=list(
              NULL,
                              c("x1","x2","x3","x4","x5","x6","F1")),
          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 assymetric 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'll 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"`.

```
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"`.

```
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 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.

```
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,
    dimnames=list(
              c("x1","x2","x3","x4","x5","x6"),
              c("x1","x2","x3","x4","x5","x6","F1")),
    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"`.

```
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),
    dimnames=list(
                      NULL,
              c("x1","x2","x3","x4","x5","x6","F1")),
    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.

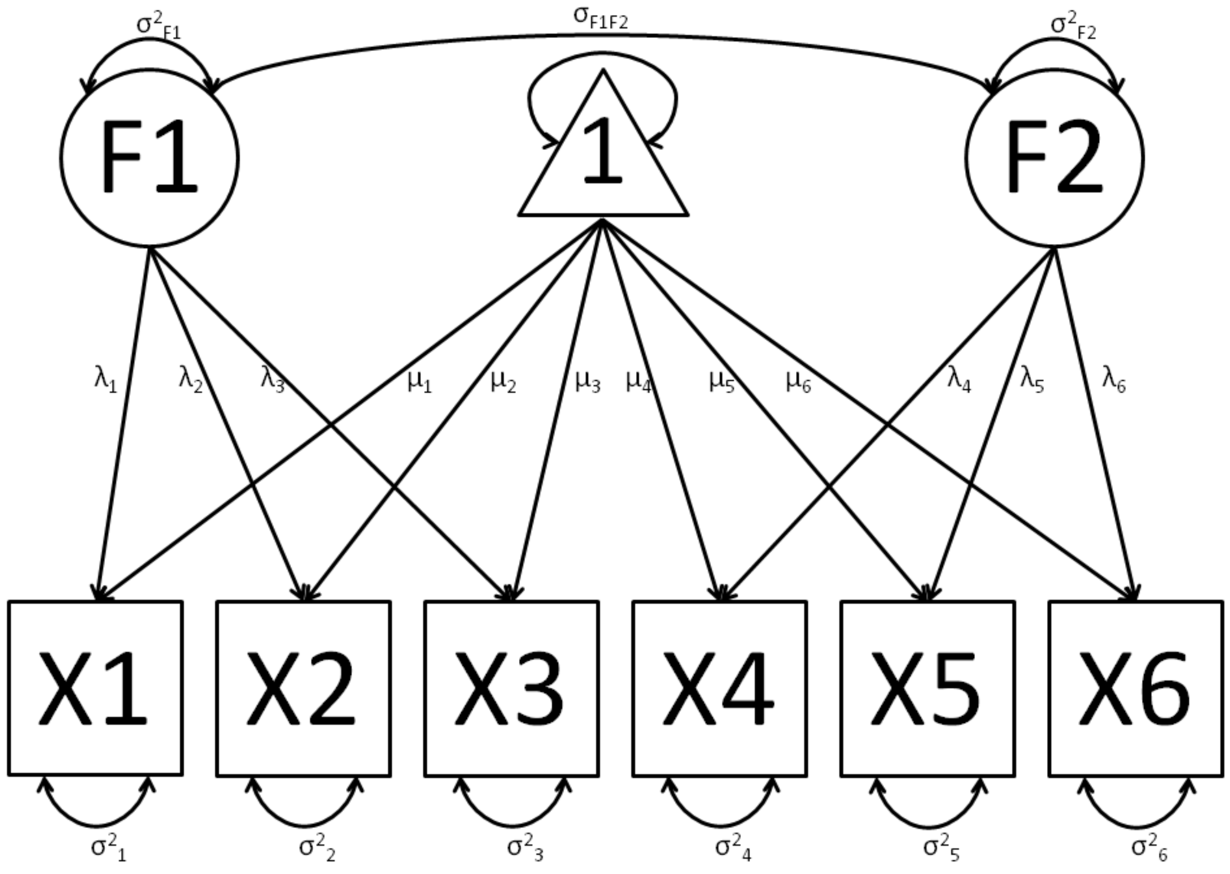
The model can now be run using the `mxRun` function, and the output of the model can be accessed from the `output` slot of the resulting model. A summary of the output can be reached using `summary()`.

```
oneFactorFit <- mxRun(oneFactorModel)

oneFactorFit@output

summary(oneFactorFit)
```

4.2.2 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'll 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'll 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", "y

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",
    type="RAM",
    mxData(
        observed=twoFactorRaw,
        type="raw",
        ),
    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"),
    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"),
    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"),
    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.

```
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.

```
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 link for path specification of these models.

4.3 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:

- LGC_MatrixCov.R
- LGC_MatrixRaw.R

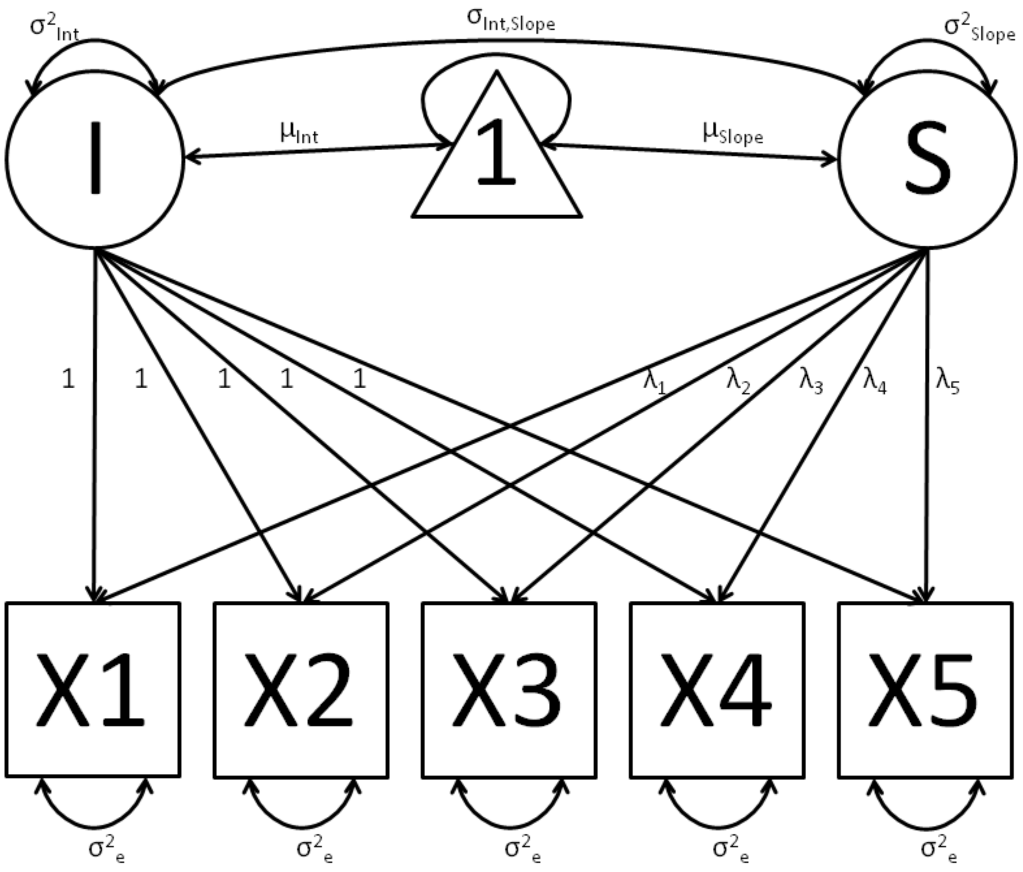
A parallel version of this example, using path-centric specification of models rather than matrices, can be found here link.

4.3.1 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:

. math:: **nowrap**

begin{eqnarray*} x_{ij} = Intercept_{i} + lambda_{j} * Slope_{i} + epsilon_{i} end{eqnarray*}



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'll 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.

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

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"))
      )
```

myLongitudinalDataMean <- 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'll then run.

```
require(OpenMx)

growthCurveModel <- mxModel("Linear Growth Curve Model, Matrix Specification",
    mxData(myLongitudinalDataRaw,
        type="raw"),
    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, "vari", "cov",
                 NA, NA, NA, NA, NA, "cov", "vars"),
        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,
        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,"meani","means"),
        name="M"),
    mxRAMObjective("A","S","F","M")
    )
```

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 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 sixth column of the A matrix with fixed values in this series: [0, 1, 2, 3, 4].

```
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. 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.

```
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.

```
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"`

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 link for path specification of these models.

4.4 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 to estimated 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:

- BivariateHeterogeneity_MatrixRaw.R

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

4.4.1 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.

Data

For this example we simulated two datasets ('xy1' and 'xy2') each with zero means and unit variances, one with a correlation of .5, and the other with a correlation of .4 with 1000 subjects each. 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

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. `mxModel` commands are unlimited in the number of arguments.

```
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, 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="Full",
            nrow=2,
            ncol=2,
            free=c(T,T,F,T),
            values=c(1,.2,0,1),
            labels=c("vX1", "cXY1", "zero", "vY1"),
            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(
            "EC1",
            "EM1"
            selVars)
        ),
    mxModel("group2",
        mxMatrix(
            type="Full",
            nrow=2,
            ncol=2,
            free=c(T,T,F,T),
            values=c(1,.2,0,1),
            labels=c("vX2", "cXY2", "zero", "vY2"),
            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(
            "EC2",
            "EM2",
            selVars)
        ), ....
```

As a result, 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`

```
$Chol1
  X      Y
X "vX1"  "zero"
Y "cXY1" "vY1"

$EM1
     X     Y
[1,] "mX1" "mY1"

$Chol2
  X      Y
X "vX2"  "zero"
Y "cXY2" "vY2"

$EM2
     X     Y
[1,] "mX2" "mY2"
```

Model Fitting

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.

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

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)
```

4.4.2 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 Homnogeneity 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 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 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.

```
$Chol1
  X      Y
X "vX1"  "zero"
Y "cXY1" "vY1"

$EM1
     X     Y
```

```
[1,] "mX1" "mY1"

$Chol2
  X      Y
  X "vX1"  "zero"
  Y "cXY1" "vY1"

$EM2
     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
```

4.5 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:

- UnivariateTwinAnalysis_MatrixRaw.R

A parallel version of this example, using path specification of models rather than matrices, can be found here link.

4.5.1 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 (mzfData) and DZ females (dzfData) respectively with the code below.

```
require(OpenMx)

#Prepare Data
twinData <- read.table("myTwinData.txt", header=T, na.strings=".")
twinVars <- c('fam','age','zyg','part','wt1','wt2','ht1','ht2','htwt1','htwt2','bmi1','bmi2')
summary(twinData)
selVars <- c('bmi1','bmi2')
mzfData <- as.matrix(subset(twinData, zyg==1, c(bmi1,bmi2)))
dzfData <- as.matrix(subset(twinData, zyg==3, c(bmi1,bmi2)))
```

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 **X** represents the additive genetic path 'a', the **Y** matrix is used for the shared environmental path 'c' and the matrix **Z** 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 **X** matrix by its transpose **t(A)**. 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. Let's go through each of the matrices step by step. 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",
```

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 two elements of the matrix 'expMeanMZ' and the two elements of the matrix 'expMeanDZ', each of which are 'Full' 1x2 matrices with free parameters and start values of 20. Note again that `dimnames` are required for matrices or algebras that generate the expected mean vectors and expected covariance matrices.

```
mxMatrix(
    type="Full",
    nrow=1,
    ncol=2,
    free=T,
    values=c(20,20),
    labels= c("mean","mean"),
    name="expMeanMZ"
),
mxMatrix(
    type="Full",
    nrow=1,
    ncol=2,
    free=T,
    values=c(20,20),
    labels= c("mean","mean"),
    name="expMeanDZ"
),
```

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', respectively, **X**, **Y** and **Z** are all 'Full' 1x1 matrices assigned the 'free' status and given a .6 starting value. We also specify the matrix **h** to have a fixed value of 0.5, necessary for the expectation of DZ twins.

```
mxMatrix(
    type="Full",
    nrow=1,
    ncol=1,
    free=TRUE,
    values=.6,
    label="a",
    name="X"
),
mxMatrix(
    type="Full",
    nrow=1,
    ncol=1,
    free=TRUE,
    values=.6,
    label="c",
    name="Y"
),
mxMatrix(
    type="Full",
    nrow=1,
    ncol=1,
    free=TRUE,
    values=.6,
    label="e",
    name="Z"
),
mxMatrix(
    type="Full",
    nrow=1,
    ncol=1,
    free=FALSE,
    values=.5,
    name="h"
),
```

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).

```
mxAlgebra(
    expression=X * t(X),
    name="A"
),
mxAlgebra(
    expression=Y * t(Y),
    name="C"
),
mxAlgebra(
    expression=Z * t(Z),
    name="E"
),
```

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 and the DZ covariance. In R, concatenation of matrices is accomplished with the 'rbind' and 'cbind' functions. Thus to represent the matrices in expression ? in R, we use the following code.

$$covMZ = \left[\begin{array}{l} a^2+c^2+e^2, a^2+c^2 \\ a^2+c^2, a^2+c^2+e^2 \end{array} \right] \quad covDZ = \left[\begin{array}{l} a^2+c^2+e^2, .5a^2+c^2 \\ .5a^2+c^2, a^2+c^2+e^2 \end{array} \right]$$

```
mxAlgebra(
    expression=rbind (cbind(A + C + E, A + C),
                      cbind(A + C    , A + C + E)),
    name="expCovMZ"
),
mxAlgebra(
    expression=rbind (cbind(A + C + E  , h %x% A + C),
                      cbind(h %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.

```
mxModel("MZ",
    mxData(
        observed=mzfData,
        type="raw"
    ),
    mxFIMLObjective(
        covariances="twinACE.expCovMZ",
        means="twinACE.expMeanMZ",
        dimnames=selVars
    )
),
mxModel("DZ",
    mxData(
        observed=dzfData,
        type="raw"
    ),
    mxFIMLObjective(
        covariances="twinACE.expCovDZ",
        means="twinACE.expMeanDZ",
        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`, and then use that as argument of the `mxAlgebraObjective` command.

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

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(expCovMZ, twinACEFit)
DZc <- mxEval(expCovDZ, twinACEFit)
M <- mxEval(expMeanMZ, twinACEFit)
A <- mxEval(A, twinACEFit)
C <- mxEval(C, twinACEFit)
E <- mxEval(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)
```

4.5.2 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 `mxmMatrix` command. First, we call up the previous 'full' model and save it as a new model 'twinAEModel'. Next we re-specify the matrix **Y** 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 <- mxModel(twinACEModel,
    mxMatrix(
        type="Full",
        nrow=1,
        ncol=1,
        free=F,
        values=0,
        label="c",
        name="Y"
    )
    )
twinAEFit <- mxRun(twinAEModel)

MZc <- mxEval(expCovMZ, twinAEFit)
DZc <- mxEval(expCovDZ, twinAEFit)
A <- mxEval(A, twinAEFit)
C <- mxEval(C, twinAEFit)
E <- mxEval(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
```

4.6 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.

4.6.1 Mean Differences

The scripts are presented here

- DefinitionMeans_MatrixRaw.R

- DefinitionMeans_MatrixRaw.mx

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 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) which we will call Σ or simply "S" in the R script.

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)

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=n, c(1,2), Sigma)
group2<-mvrnorm(n=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)
y<-rbind(group1,group2)
dimnames(y)[2]<-list(c("x","y"))
def<-rep(c(1,0),each=n)
selvars<-c("x","y")
```

The objects *y* 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("DefinitionMeans",
    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 `"DefinitionMeans"`.

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.

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(y,def)),
    type="raw"
),
```

Model specification is carried out using `mxMatrix` functions to create matrices for the model. In the present case, we need four matrices. First is the predicted covariance matrix, *Sigma*. Next, we use three matrices to specify the model for the means. First is *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.

```
mxMatrix(
    type="Symm",
    nrow=2,
    ncol=2,
    free=TRUE,
    values=c(1, 0, 1),
    name="Sigma"
),
mxMatrix(
    type="Full",
    nrow = 1,
    ncol = 2,
    free=TRUE,
    name = "M"
),
mxMatrix(
    type="Full",
    nrow=1,
    ncol=2,
    free=TRUE,
    values=c(0, 0),
    name="beta"
),
mxMatrix(
    type="Full",
    nrow=1,
    ncol=2,
    free=FALSE,
    labels=c("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. 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 the 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 we compute some summary statistics from the 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 we extract the parameter estimates and matrix algebra results from the model
Sigma<-run@matrices$Sigma@values
Mu<-run@algebras$Mu@result
M<-run@matrices$M@values
beta<-run@matrices$beta@values

# Third, we 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(Mu),.001)
```

CHAPTER

FIVE

CHANGES IN OPENMX

5.1 trunk

- in summary(), renamed "parameter estimate" column to "Estimate" and renamed "error estimate" to "Std.Error"
- tools/mxAlgebraParser.py will convert Mx 1.0 algebra expressions (Python PLY library is required)
- added 'dimnames' argument to mxFIMLObjective() and mxMLObjective()

5.2 Release 0.1.5-851

- 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

5.3 Release 0.1.4-827

- 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

5.4 Release 0.1.3-776

- 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).

5.5 Release 0.1.2-708

- **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.

5.6 Release 0.1 (August 3, 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

- *Index*
- *Module Index*
- *Search Page*