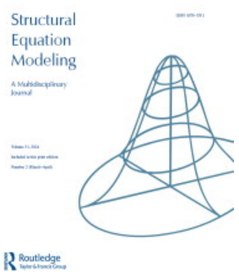
# Using SymPy (Symbolic Python) for Understanding Structural Equation Modeling

**Article** · March 2024

DOI: 10.1080/10705511.2024.2325122

Joel S Steele
University of North Dakota
**53** PUBLICATIONS **2,012** CITATIONS



Structural Equation Modeling: A Multidisciplinary Journal

ISSN: (Print) (Online) Journal homepage: www.tandfonline.com/journals/hsem20

# Using SymPy (Symbolic Python) for Understanding Structural Equation Modeling

Joel S. Steele & Kevin J. Grimm

**To cite this article:** Joel S. Steele & Kevin J. Grimm (27 Mar 2024): Using SymPy (Symbolic Python) for Understanding Structural Equation Modeling, Structural Equation Modeling: A Multidisciplinary Journal, DOI: 10.1080/10705511.2024.2325122

**To link to this article:** https://doi.org/10.1080/10705511.2024.2325122

STRUCTURAL EQUATION MODELING: A MULTIDISCIPLINARY JOURNAL
https://doi.org/10.1080/10705511.2024.2325122

Routledge
Taylor & Francis Group

TEACHER'S CORNER

OPEN ACCESS  Check for updates

# Using SymPy (Symbolic Python) for Understanding Structural Equation Modeling

Joel S. Steele[a] and Kevin J. Grimm[b]

[a]University of North Dakota; [b]Arizona State University

**ABSTRACT**
Structural Equation Modeling (SEM) continues to grow in popularity with numerous articles, books, courses, and workshops available to help researchers become proficient with SEM quickly. However, few resources are available to help users gain a deep understanding of the analytic steps involved in SEM, with even fewer providing reproducible syntax for those learning the technique. This work builds off of the original work by Ferron and Hess to provide computer syntax, written in python, for the specification, estimation, and numerical optimization steps necessary for SEM. The goal is to provide readers with many of the numerical and analytic details of SEM that may not be regularly taught in workshops and courses. This work extends the original demonstration by Ferron and Hess to incorporate the reticular action model notation for specification as well as the estimation of variable means. All of the code listed is provided in the appendix.

**KEYWORDS**
Open-source software; Python; RAM; SEM; symbolic math

## 1. Introduction

There is little debate that structural equation modeling (SEM) is a vital tool for understanding multivariate data. The flexibility and sophistication that is possible within SEM continues to support progression in research design and analysis through the concise specification of complex models (Asparouhov & Muthen, 2006), mediation analysis and assessment of direct and indirect relations (Goldsmith et al., 2018; Gunzler et al., 2013; Ledermann et al., 2011), as well as the identification of latent level constructs (Muthén, 2002) and their trajectories over time (McArdle & Bell, 2000; Preacher, 2010).

However, teaching Structural Equation Modeling presents unique challenges for both students and instructors. The question arises of whether the focus should be on the technical inner workings of the method, or is an applied approach most appropriate? In many situations a delicate balance is attempted to present both. Within the time constraints of a workshop or academic term this may suffice to get attendees or students started using SEM, yet many of the mathematical details are then left to be presented as esoteric, ancillary facts. It can be difficult for those interested in a deeper understanding of SEM to see exactly how the various steps are performed. Enter the wonderful paper by Ferron and Hess (2007), which presents a concrete example of the process of fitting a structural equation model. Their presentation is explicit, yet concise, and offers those curious about SEM a much needed resource for building a deep understanding of the technical and analytic steps involved in fitting these models.

There remains, however, an untapped opportunity in their presentation, in that readers are not able to work with the equations and values themselves, and so, cannot explore the methods or examine alternate specifications and models. As Chinese philosopher Confucius (551-479 BC) said, "I hear and I forget, I see and I remember, I do and I understand." Thus, in what follows, the free, Open-source scripting language Python (Van Rossum & Drake, 2009) and the symbolic mathematics package SymPy (Meurer et al., 2017), are used to fill this gap. The goal is to allow readers to "do" and "understand" SEM more deeply. While there are existing packages that can perform symbolic math in R (Andersen & Højsgaard, 2019, 2021) or that integrate SymPy with R for SEM (Cheung, 2020), this demonstration is intended to be more instructional regarding the analytic steps of SEM. Specifically, this presentation outlines the various steps in fitting SEMs along with the accompanying Python and Sympy code so that readers can test and replicate what is presented. The order of topics include (1) parameter and model specification, (2) the computation of model-based expectations, (3) selection of a loss function for parameter estimation, and (4) coding of the steps necessary for numerical optimization. To the extent possible, each portion includes the necessary code to produce the analytic and numerical output provided. In order to check the work along the way, numerical results are compared to estimates obtained from the *lavaan* package (Rosseel, 2012) available in R.

**CONTACT** Joel S. Steele joel.steele@und.edu University of North Dakota, Grand Forks, ND, USA.

Regrettably, there is far too much ground to cover in a single presentation. SEM, after all, is a mature and widely developed technique for modeling, with myriad options, use-cases, and caveats to consider. Therefore, this presentation is focused on providing tools and code to demonstrate how major numerical tasks in SEM are done. This means that many of the details regarding inference, hypothesis testing, and model evaluation are omitted. In spite of this, the hope is that this presentation remains useful.

## 2. Parameter and Model Specification

Much of statistical modeling boils down to defining what is *expected* and comparing it to what is *observed* and SEM is no different. The process of specifying a statistical model involves a combination of theoretical curiosity and mathematical translation. The result is often either a single equation or a set of equations that represent the hypothesized relations among variables. Within SEM multiple relations can be specified within the same model. The exact relations among various measures should ideally reflect theory, yet analytically, it is also important to be explicit about what can be assumed to be known, or to use the statistical term *fixed*, and what needs to be estimated, or *free*. Within the equations, *free* components may include the relations among measures, the central tendency of measures, or indices of variability and spread. Importantly, if these components are not *fixed* they represent *parameters* to be estimated and are assigned a unique label (greek letters are used throughout this presentation to distinguish *free* parameters). Importantly, all models result in a set of expectations that specify how variables relate and which combinations of *fixed* and *free* parameters are involved in computing specific values.

### 2.1. RAM Specification

In practice, much of the minutia of determining model-based expectations is hidden from users, in that the software handles this based on the relations specified among variables. There are a number of ways in which model based relations can be specified and how expectations can be computed including the popular **LiSRel** model (Jöreskog & Sörbom, 1982). For this presentation the *Reticular Action Model* or RAM specification (McArdle & McDonald, 1984) is used.

One benefit of the RAM specification is that it only involves three matrices, labeled **S**, **A**, and **F** to fully specify a model. The first matrix, **S** holds the specifications for symmetric relations, such as variances, covariances, and correlations, referred to as the *slings* matrix. The second matrix, **A** holds asymmetric relations, such as regression paths or factor loadings, referred to as the *arrows* matrix. With these two matrices alone, expectations for the entire model, including latent variables and residuals, can be computed. To clearly separate observed or *manifest* variable terms from unobserved, or *latent* variable terms, the final matrix, **F**, also known as the *filter* matrix, is used. Again, all models result in a set of expectations, and the SEM based expectation matrix $\hat{\Sigma}$ of variances and covariances can be computed using the RAM matrices **S**, **A**, and **F** as,

$$\hat{\Sigma} = F(I-A)^{-1}S(I-A)^{-1T}F^{T}. \tag{1}$$

where $I$ represents the identity matrix, or the matrix equivalent of the number 1, the $-1$ superscript indicates matrix inversion or division by a matrix, and the $T$ superscript indicates the matrix transpose, or a swapping of rows for columns. Mean expectations are also specified as,

$$\hat{\mu} = F(I-A)^{-1}m.$$

where $m$ represents a vector of means of all variables, manifest and latent, in the model (Oertzen & Brick, 2014).

It is important to understand how these matrices are used and how parameters are placed in them in order to specify the desired variable relations. The placement of terms proceeds from the column that represents the starting point of the relation and ends on the row that represents the destination of the relation. This is true for both the **S** and **A** matrices, see Figure 1. In the figure, a simple regression model is specified with the terms listed as **X**, **Y**, and **e**. The specification of the all three matrices is shown below the path diagram of the regression of **Y** on **X**. The direct path from **X** to **Y** is specified in the **A** matrix using the $\beta_1$ parameter, whereas the variance of **X** can be seen in the **S** matrix by the placement of the $\sigma_x^2$ parameter. While there is nothing to keep one from listing variables in any order in these matrices it is important that whichever order is selected, it is the same for all of the matrices in the model. In this demonstration, the selected order of model variables in each instance is stated explicitly.

## 3. Using SymPy for a Deeper Understanding of Structural Equation Models

With the preliminaries out of the way, it's time to illustrate the steps above using the SymPy package in python3. Again, this module allows for mathematical manipulations of symbols specified by the user. The first step involves importing the package. This is done using the prefix `sm` to highlight which of the commands are drawn explicitly from the SymPy package. Therefore, any command with the prefix `sm`, like the latex printing initialization below, uses SymPy.

```
import sympy as sm # import Symbolic Python module
sm.init_printing() # initialize printing so the latex symbols render
```

## 4. Symbolic Expectations to Parameter Estimation

In order to illustrate the utility of the SymPy package, what follows is a replication of the model presented by Ferron and Hess (2007). Figure 2 presents a path diagram of the demonstration model.

Based on Figure 2, note that there are five variables total, both manifest and latent, that are listed. Thus the **S** and **A** matrices have dimension $5 \times 5$. Below is a labeled matrix to illustrate the selected variable arrangement.

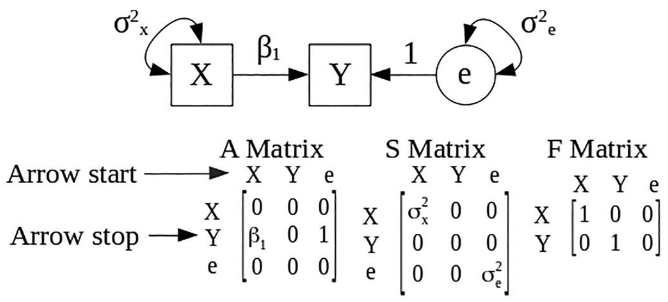


**Figure 1.** RAM arrow specification illustration.

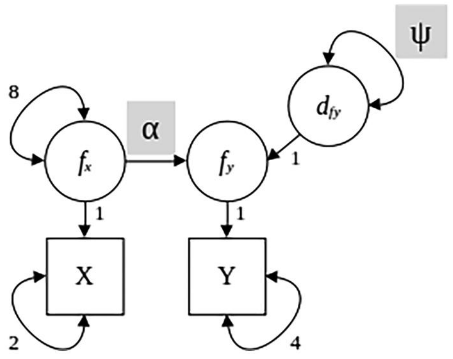


**Figure 2.** Replication of demonstration path model from Ferron and Hess (2007).

$$A\&S = \begin{array}{c|ccccc} & X & Y & f_x & f_y & d_{fy} \\ \hline X & 0 & 0 & 0 & 0 & 0 \\ Y & 0 & 0 & 0 & 0 & 0 \\ f_x & 0 & 0 & 0 & 0 & 0 \\ f_y & 0 & 0 & 0 & 0 & 0 \\ d_{fy} & 0 & 0 & 0 & 0 & 0 \end{array}$$

With a corresponding F matrix that is $2 \times 5$ in size,

$$F = \begin{array}{c|ccccc} & X & Y & f_x & f_y & d_{fy} \\ \hline X & 1 & 0 & 0 & 0 & 0 \\ Y & 0 & 1 & 0 & 0 & 0 \end{array}$$

Looking again at Figure 2, the two parameters to be estimated for this model represent the regression path of $f_y$ on $f_x$ represented by the parameter $\alpha$, as well as the variance of the disturbance in $f_y$ as a result of this regression represented by the parameter $\psi$. These symbols can be created in python using the `symbols()` function from SymPy.

```
#create our math symbols
alpha, psi = sm.symbols(('alpha', 'psi'))
```

$$[\alpha, \psi]$$

The next step is to create **S** and **A** as $5 \times 5$ matrices of zeros. Once created, the symbols $\alpha$ and $\psi$ along with any fixed parameters are placed in the appropriate location in these matrices to reflect the desired relation or quantity. Note that any non-symbol value, including zero, represents a fixed value on the specified path.

Below are the steps for creation of the *slings* matrix.

```
S = sm.zeros(5,5) # create the slings matrix
for i in [0,1,2]: # fill in the numeric constants
  S[i,i]=2**(i+1)

S[4,4]=psi # add in the free parameter
```

$$S = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \psi \end{bmatrix}$$

Next are the steps for the *arrows* matrix.

```
A = sm.zeros(5,5) # create the arrows matrix
A[0,2]=A[1,3]=A[3,4]=1 # fill in numeric constants
A[3,2]=alpha # add in the free parameter
```

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \alpha & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

It is important to note that python starts indexing at zero, thus the above positions in each matrix must be specified using a zero index start. Finally, a short cut is taken to create the *filter* matrix. The *filter* matrix has dimensions that reflect the number of manifest variables as the number of rows, and the total number of model variables, both manifest and latent as the number of columns. The syntax below joins a $2 \times 2$ identity matrix to a rectangular $2 \times 3$ matrix of zeros to create the appropriately sized filter.

```
F = sm.eye(2).row_join(sm.zeros(2,3)) # create the filter matrix
```

$$F = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

## 5. Model expectations

As a reminder, the RAM method for determining expectations based on the model parameters is listed in Equation (1) above and involves matrix inversion and transposition. The following steps provide the python coded needed for the computation of $(I - A)^{-1}$. This is the inverse of the $(I - A)$.

```
Ainv = (sm.eye(5)-A).inv() # subtract A from I and invert
```

$$(I - A)^{-1} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & \alpha & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The computation of the expectation matrix $(\hat{\Sigma})$ is illustrated next, this is a direct translation of Equation (1). This matrix represents the expected covariance matrix that results as a function of the model specification. Notice that it includes all of the specified parameters.

```
RAMexp = F*Ainv*S*Ainv.T*F.T
# model based expectation of covariance matrix
```

$$\hat{\Sigma} = \begin{bmatrix} 10 & 8\alpha \\ 8\alpha & 8\alpha^2 + \psi + 4 \end{bmatrix}$$

## 6. Loss Function Selection and Model Parameter Estimation

This model-based expectation matrix ($\hat{\Sigma}$) is used to compute the expected covariance based on different candidate values of the parameters. Once computed the values are compared to the observed covariance matrix ($S$), and the fit is assessed based on an objective, or cost, function. For this demonstration, the maximum likelihood (ML) fitting function is used to estimate parameters in $\hat{\Sigma}$. The ML fit function is specified as,

$$F_{ML} = \log\left|\hat{\Sigma}\right| + tr(S\hat{\Sigma}^{-1}) - \log|S| - (p+q), \tag{2}$$

where, $\hat{\Sigma}$ is the model implied covariance matrix, $S$ is the observed covariance matrix and the terms $p$ and $q$ represent the number of observed variables in the data. The $|S|$ and $\left|\hat{\Sigma}\right|$ terms represent the determinants of those matrices with the negative exponent on the model implied covariance matrix, $\hat{\Sigma}^{-1}$, signifies matrix inversion. Additionally, the $tr()$ portion represents the trace of a matrix and applies to the terms in parentheses. Importantly, the last term $(p+q)$ is a constant for any given search based on Equation (2), and can therefore be omitted when implemented in code. Below is a break down of each component involved.

### 6.1. Matrix Determinant

The determinant of a square matrix can be thought of as a substitute for the variance of a matrix, or rather, of the equations represented in the matrix.

```
mex_det = RAMexp.det() # determinant of the expectation matrix
```

$$\left|\hat{\Sigma}\right| = 16\alpha^2 + 10\psi + 40$$

### 6.2. Matrix Inverse

Conceptually, matrix inversion is a way of dividing by a square matrix. It is accomplished by utilizing the multiplicative inverse property from algebra, where any number when multiplied by its reciprocal (inverse) will result in 1. The same is true for a square matrix. A matrix multiplied by its inverse results in the identity matrix. The term *solution* comes from this fact the matrix inverse solves the the multiplicative inverse property for matrices. Thus, matrix inversion is an important matrix operation whenever division is needed.

```
mex_inv = RAMexp.inv() # inverse of the expectation matrix
```

$$\hat{\Sigma}^{-1} = \begin{bmatrix} \frac{8\alpha^2+\psi+4}{16\alpha^2+10\psi+40} & -\frac{4\alpha}{8\alpha^2+5\psi+20} \\ \frac{4\alpha}{8\alpha^2+5\psi+20} & \frac{10}{16\alpha^2+10\psi+40} \end{bmatrix}$$

Recalling Equation (2), which specifies the maximum likelihood function, which serves as the objective or cost function for this exercise, the sample based observed covariance matrix $S$ also needs to be specified. These values mirror those presented in Ferron and Hess (2007).

```
Sob = sm.Matrix([[5, 10, 20]]) # observed covariance matrix
```

$$S = \begin{bmatrix} 10 & 5 \\ 5 & 20 \end{bmatrix}$$

```
Sob_det = Sob.det() # determinant of observed covariance matrix
```

$$|S| = 175$$

### 6.3. Maximum Likelihood Objective Function

With each component of the objective function in place, the next step is to translate Equation (2). into workable python code.

```
# create the objective function
objective = sm.log(mex_det) + (Sob*mex_inv).
trace() sm.log(Sob_det)
ml_obj = objective.simplify()
```

$$F_{ML} = \frac{40\alpha^2 - 40\alpha + 5\psi + \left(8\alpha^2+5\psi+20\right)\log\left(\frac{16\alpha^2}{175}+\frac{2\psi}{35}+\frac{8}{35}\right)+120}{8\alpha^2+5\psi+20}$$

## 7. Numerical Optimization via Root Finding

The next step in fitting an SEM involves a search for values of the *free* parameters $(\alpha, \psi)$ that when substituted into the objective function will bring the model expectations as close a possible to the observed values. This can be accomplished in a number of ways. The code that follows utilizes root finding in order to identify and compare candidate values to the actual values of the observed covariance matrix. Specifically, Newton's method for root finding is used.

Briefly, the equation below indicates that the next best value to try $x_{[n+1]}$ is based on the current value $x_{[n]}$, the function in question evaluated at the current value $f(x_{[n]})$, and the first derivative, or slope, of the function evaluated at the current value $f'(x_{[n]})$.

$$x_{[n+1]} = x_{[n]} - \frac{f(x_{[n]})}{f'(x_{[n]})}$$

In this demonstration, the equation to be evaluated is the first derivative of the objective function. Thus, the roots of this equation represent specific combinations of the potential values for the model parameters ($\alpha$ and $\psi$) where the slope of a line tangent to the objective function is flat. The reason for this is that the goal is to locate the point(s) where the function is at a minimum. Think of being at the bottom of a trough, which indicates that there is minimal discrepancy between the model based values of the covariance matrix and the observed covariance matrix values. Thus, finding the function's roots, or where the function is equal to zero, indicates which parameter values minimize the objective function. In practice, this means using the first derivative $f'(x_{[n]})$ instead of the function $f(x_{[n]})$ and the second derivative $f''(x_{[n]})$ substituted for the first derivative $f'(x_{[n]})$ in the root finding equation above. The process below utilizes a multivariate extension of this approach.

For multivariate functions, as in SEM, the first and second derivatives are actually partial derivatives, meaning that these computations represent how the function changes with regard to a particular parameter holding the other parameters constant. The resulting partial derivatives are grouped into a vector known as the gradient, which represents the slope of the equation at a given point, and a matrix of partial second derivatives known as the Hessian. The steps of this process are outlined in python below.

## 7.1. Partial derivatives

Below is code to compute the partial derivative of the objective function for $\alpha$,

```
d1a = sm.diff(ml_obj,alpha).simplify() # first derivative with respect to alpha
```

$$\frac{\partial f}{\partial \alpha} = \frac{8\left(16\alpha^3 + 40\alpha^2 + 50\alpha\psi - 25\psi - 100\right)}{64\alpha^4 + 80\alpha^2\psi + 320\alpha^2 + 25\psi^2 + 200\psi + 400}.$$

The same process is done for the $\psi$ parameter,

```
d1p = sm.diff(ml_obj,psi).simplify() # first derivative with respect to psi
```

$$\frac{\partial f}{\partial \psi} = \frac{5\left(-24\alpha^2 + 40\alpha + 5\psi - 80\right)}{64\alpha^4 + 80\alpha^2\psi + 320\alpha^2 + 25\psi^2 + 200\psi + 400}.$$

These partial derivatives are then stacked into a $2 \times 1$ vector called the gradient ($\nabla f$) of the objective function.

```
fitgrad = sm.Matrix([d1a,d1p]) # use derivatives to create the gradient vector
```

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial \alpha} \\ \frac{\partial f}{\partial \psi} \end{bmatrix} = \begin{bmatrix} \frac{8\left(16\alpha^3+40\alpha^2+50\alpha\psi-25\psi-100\right)}{64\alpha^4+80\alpha^2\psi+320\alpha^2+25\psi^2+200\psi+400} \\ \frac{5\left(-24\alpha^2+40\alpha+5\psi-80\right)}{64\alpha^4+80\alpha^2\psi+320\alpha^2+25\psi^2+200\psi+400} \end{bmatrix}$$

The second partial derivatives of the $2 \times 1$ gradient results in a $4 \times 4$ matrix called the Hessian. A simplified view of this matrix is that it represents the curvature of the function at a given point. In practice, greater curvature for a minimum point would indicate high precision around this location in the parameter space, in that the steepness of the function changes rapidly in the immediate vicinity.

Below is the python code to take the derivatives of the first derivative functions from above.

```
d2a = sm.diff(d1a,alpha).simplify() # second derivative for alpha
```

$$\frac{\partial^2 f}{\partial \alpha^2} = \frac{16\left(-64\alpha^4 - 320\alpha^3 - 480\alpha^2\psi + 480\alpha^2 + 600\alpha\psi + 2400\alpha + 125\psi^2 + 500\psi\right)}{512\alpha^6 + 960\alpha^4\psi + 3840\alpha^4 + 600\alpha^2\psi^2 + 4800\alpha^2\psi + 9600\alpha^2 + 125\psi^3 + 1500\psi^2 + 6000\psi + 8000}$$

```
d2p = sm.diff(d1p,psi).simplify() # second derivative for psi
```

$$\frac{\partial^2 f}{\partial \psi^2} = \frac{25\left(56\alpha^2 - 80\alpha - 5\psi + 180\right)}{512\alpha^6 + 960\alpha^4\psi + 3840\alpha^4 + 600\alpha^2\psi^2 + 4800\alpha^2\psi + 9600\alpha^2 + 125\psi^3 + 1500\psi^2 + 6000\psi + 8000}$$

The off diagonal elements for the Hessian matrix can be computed in either of the following ways. As can be seen, the results are equivalent. This is not always guaranteed to be true, but if all of the second derivatives are continuous then these values are symmetric (this property is called the *equality of mixed partials* or the *symmetry of second derivatives*).

```
# mixed partial using first derivative for alpha, with respect to psi
d2ap = sm.diff(d1a,psi).simplify()
```

$$\frac{\partial^2 f}{\partial \psi \partial \alpha} = \frac{40\left(48\alpha^3 - 120\alpha^2 - 50\alpha\psi + 200\alpha + 25\psi + 100\right)}{512\alpha^6 + 960\alpha^4\psi + 3840\alpha^4 + 600\alpha^2\psi^2 + 4800\alpha^2\psi + 9600\alpha^2 + 125\psi^3 + 1500\psi^2 + 6000\psi + 8000}$$

```
# mixed partial using first derivative for psi, with respect to alpha
d2pa = sm.diff(d1p,alpha).simplify()
```

$$\frac{\partial^2 f}{\partial \alpha \partial \psi} = \frac{40\left(48\alpha^3 - 120\alpha^2 - 50\alpha\psi + 200\alpha + 25\psi + 100\right)}{512\alpha^6 + 960\alpha^4\psi + 3840\alpha^4 + 600\alpha^2\psi^2 + 4800\alpha^2\psi + 9600\alpha^2 + 125\psi^3 + 1500\psi^2 + 6000\psi + 8000}$$

The Hessian matrix is created by combining the second partial derivatives as outlined below,

$$\mathbf{H}_f = \begin{bmatrix} \dfrac{\partial^2 f}{\partial\alpha^2} & \dfrac{\partial^2 f}{\partial\alpha\partial\psi} \\ \dfrac{\partial^2 f}{\partial\psi\partial\alpha} & \dfrac{\partial^2 f}{\psi^2} \end{bmatrix}.$$

The python generated values are presented below.

```
# create the Hessian matrix
fithess = sm.Matrix([[d2a,d2ap],[d2pa,d2p]])
```

$$\mathbf{H}_f = \begin{bmatrix} \frac{16\left(-64\alpha^4-320\alpha^3-480\alpha^2\psi+480\alpha^2+600\alpha\psi+2400\alpha+125\psi^2+500\psi\right)}{512\alpha^6+960\alpha^4\psi+3840\alpha^4+600\alpha^2\psi^2+4800\alpha^2\psi+9600\alpha^2+125\psi^3+1500\psi^2+6000\psi+8000} & \frac{40\left(48\alpha^3-120\alpha^2-50\alpha\psi+200\alpha+25\psi+100\right)}{512\alpha^6+960\alpha^4\psi+3840\alpha^4+600\alpha^2\psi^2+4800\alpha^2\psi+9600\alpha^2+125\psi^3+1500\psi^2+6000\psi+8000} \\ \frac{40\left(48\alpha^3-120\alpha^2-50\alpha\psi+200\alpha+25\psi+100\right)}{512\alpha^6+960\alpha^4\psi+3840\alpha^4+600\alpha^2\psi^2+4800\alpha^2\psi+9600\alpha^2+125\psi^3+1500\psi^2+6000\psi+8000} & \frac{25\left(56\alpha^2-80\alpha-5\psi+180\right)}{512\alpha^6+960\alpha^4\psi+3840\alpha^4+600\alpha^2\psi^2+4800\alpha^2\psi+9600\alpha^2+125\psi^3+1500\psi^2+6000\psi+8000} \end{bmatrix}.$$

Focusing back on the specification of the Newton-Raphson formula for root finding based on the gradient and Hessian, the function can be compactly represented as,

$$f(\alpha_{t+1}, \psi_{t+1} | \alpha_t, \psi_t) = \begin{bmatrix} \alpha_t \\ \psi_t \end{bmatrix} - \mathbf{H}_\mathbf{f}^{-1} \nabla f. \tag{3}$$

Below is the python code to accomplish the specification. The results of the full specification are not displayed since it is quite large and does not fit on the page.

```
# Newton based root finding algorithm
root_func = sm.Matrix([alpha,psi]) - fithess.
inv()*fitgrad
```

It may be worth while to reflect on what has been accomplished to this point. The model parameters of interest,$(\alpha, \psi)$, have been initialized as have the three matrices of the RAM notation (**A**, **S**, and **F**) that hold the model specification to reflect the model in Figure 2. Additionally, the model-based expectation matrix $\hat{\Sigma}$ has also been computed using Equation (1), along with each component of the objective function specified using Equation (2). The objective function has been differentiated twice to produce the function gradient $\nabla f$ and the Hessian $\mathbf{H}_\mathbf{f}$ for root finding using the Newton-Raphson method.

The remaining steps include the process of root finding to identify the values of $\alpha$ and $\psi$ that minimize the objective function. The following section focuses on fitting the model and involves some illustrative python code to examine and store results from each iteration of the search.

## 7.2. Model Fit

With the objective function, gradient, and Hessian available it is possible to begin the search. To begin, starting values are passed into these functions. In the python code below the `.subs()` function is used to substitute in starting values of $\alpha = 0.5$ and $\psi = 5.0$ into the root finding function to generate the next set of candidate values for the model parameters.

```
# evaluation the root finding function with starting values
res1 = root_func.subs([(alpha,0.5),(psi,5.0)])
```

$$\begin{bmatrix} \alpha_{t+1} \\ \psi_{t+1} \end{bmatrix} = \begin{bmatrix} 0.610146471371505 \\ 7.74715312916112 \end{bmatrix}$$

The search continues with the Newton-Raphson algorithm based on the results of the gradient (first derivatives). With each iteration the gradient values are compared to zero, or a value that is close enough. Setting this criteria to an absolute zero is not practical due to computer precision, wherein a true zero value may never be reached. Instead a threshold value $\epsilon$ is specified that serves as the gradient convergence limit. Specifically, so long as neither of the gradient values are lower than $\epsilon = 3.00e - 10,$ the search for parameter estimates continues. Once the threshold is passed, the algorithm is considered to have *converged.* Of course this is a rather crude choice, but these steps illustrate some of the key aspects of fitting SEMs to empirical data. Of note, gradient based search methods are not guaranteed to converge and in most applications a limit is set on the number of iterations used in root finding. If the search exceeds the pre-specified number of iterations before the convergence threshold is met, most software programs will simply stop and report an error. In the code below, the iteration limit is set to 20. The following code specifies a small loop to complete each optimization step, take note of the comments in the code below.

```
# import pandas for easy data manipulation
import pandas as pd # for collecting results and display
res_df = pd.DataFrame(columns=['itr','a',
'p','gr_a','gr_p','f_obj'])
tcount = 0 # loop limit iterator
epsilon = 0.0000000003 # gradient convergence limit
pres = [0.5,5.0] # starting values for parameters
# initial evaluation for estimation
# another way to evaluate a function in sympy
# start with gradient at starting values
grad = fitgrad.evalf(subs={alpha:pres[0],
psi:pres[1]})
# objective function value at starting values
obj = ml_obj.evalf(subs={alpha:pres[0],
psi:pres[1]})
```

```
# package up results in a dictionary
outres = {"itr":tcount, "a":pres[0],"p":
pres[1],
  "gr_a":grad[0],"gr_p":grad[1], "f_obj":obj}
# add it to the results DataFrame for display later.
res_df = pd.concat([res_df,pd.DataFrame
([outres])])
```

Rather than code each step in turn, a while loop is used below to continue the process of updating values and testing results until one of the gradient values is close enough to zero to stop.

```
# track the gradient until one value is below epsilon or we
pass 20 iterations
while (abs(grad[0]) > epsilon or abs(grad[1])
> epsilon) and tcount < 20:
  # current results of root finding function (next values to
  try)
  cres = root_func.evalf(subs={alpha:p-
  res[0],psi:pres[1]})
  # gradient evaluation using current results, function slope
  at these points
  grad = fitgrad.evalf(subs={alpha:-
  cres[0],psi:cres[1]})
  # objective function evaluation using current results
  obj = ml_obj.evalf(subs={alpha:cres[0],-
  psi:cres[1]})
  # package up results for output
  outres = {"itr":tcount + 1,
    "a":cres[0],"p":cres[1],
    "gr_a":grad[0],"gr_p":grad[1],
    "f_obj":obj}
  # add results of this iteration and update for next loop
  res_df = pd.concat([res_df,pd.DataFrame
  ([outres])])
  # swap previous results for current results for next loop
  pres = cres
  # increment the iterations counter
  tcount += 1
```

Table 1 lists the results of each iteration of our root finding function based on the maximum likelihood objective function.

## 8. Checking against Lavaan

Below model estimates are checked against the values returned from the R package for latent variable analysis *lavaan* (Rosseel, 2012).

```
suppressPackageStartupMessages(library(lavaan))
FnH_mod = '
fx =~ 1*X
fy =~ 1*Y
dfy =~ 1*fy

fy ~ alph*fx
dfy ~~ psi*dfy
X ~~ 2*X
Y ~~ 4*Y
fy ~~ 0*fy
fx ~~ 8*fx
fx ~~ 0*dfy
'
# observed variance matrix
lwr = '
10
5 20'
cov_ob = getCov(lwr, lower = T,names=c('X','Y'))
# fit the data
fit = sem(FnH_mod,
          sample.cov = cov_ob,
          sample.nobs = 100, # arbitrary number of
          observations to fit the model.
          int.ov.free = F)

sem_check = parTable(fit)
sem_check_tab = sem_check[sem_check$-
free!=0,c('label','est')]
sem_check_tab$label = c('$\\alpha$','$\\psi$')
kable(sem_check_tab,row.names = F,
  caption='lavaan comparisons results',
col.names = c('Parameter','Estimate'))
```

Table 1. Newton-Raphson Root finding results.

| Iteration | $\alpha$ | $\psi$ | $\nabla\alpha$ | $\nabla\psi$ | $f_{ML}$ |
|---|---|---|---|---|---|
| 0 | 0.5 | 5.0 | −0.3187 | −9.280e-2 | 2.251e + 0 |
| 1 | 0.61015 | 7.7472 | −0.0815 | −3.386e-2 | 2.069e + 0 |
| 2 | 0.62296 | 10.5307 | −0.0221 | −1.023e-2 | 2.011e + 0 |
| 3 | 0.62483 | 12.3196 | −0.0040 | −1.936e-3 | 2.001e + 0 |
| 4 | 0.62500 | 12.8408 | −0.0002 | −1.121e-4 | 2.000e + 0 |
| 5 | 0.62500 | 12.8749 | −0.0000 | −4.359e-7 | 2.000e + 0 |
| 6 | 0.62500 | 12.8750 | −0.0000 | −6.649e-12 | 2.000e + 0 |

Table 2. Lavaan comparisons results.

| Parameter | Estimate |
|---|---|
| $\alpha$ | 0.625 |
| $\psi$ | 12.700 |

Comparing the results listed in Table 2 from *lavaan* to those from the python optimization steps yield similar estimates, $\alpha = .625$ from both methods and $\psi = 12.875$ for SymPy and 12.700 for *lavaan*. Other options are available for numerical optimization, including non-gradient based search methods, however, the estimates obtained from the Newton-Raphson steps above are notably close to those obtained from *lavaan*.

## 9. Adding in Estimation of the Mean Parameters

The above illustration can be classified as covariance structure modeling because no variable means were estimated. The addition of means to covariance modeling enables the development of growth curves and multigroup comparisons for latent variable models. Manifest variables means can be used to estimate means for latent variables as well. This important contribution to the SEM framework can be accomplished using the RAM framework as illustrated below and following the steps outlined in the work of Grimm and McArdle (2005).

Notably, the estimation of the means in the RAM notation utilizes the same three matrices **A**, **S**, and **F**. These matrices are expanded to accommodate the inclusion of a unit constant ($k = 1$) for estimation of the latent variable means. The constant term is included as a part of the model matrices and is specified as having zero covariance relations with any other variable, a mean square of one, and a regression path fixed at one to each variable for which means are estimated.

Below is an illustration of the order of variables for the estimation of the model presented in Figure 2 with the inclusion of the constant term ($k$) for mean estimation.

$$A\&S = \begin{array}{c|cccccc} & k & X & Y & f_x & f_y & d_{fy} \\ \hline k & 0 & 0 & 0 & 0 & 0 & 0 \\ X & 0 & 0 & 0 & 0 & 0 & 0 \\ Y & 0 & 0 & 0 & 0 & 0 & 0 \\ f_x & 0 & 0 & 0 & 0 & 0 & 0 \\ f_y & 0 & 0 & 0 & 0 & 0 & 0 \\ d_{fy} & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

In the code below, new parameter terms are specified which represent the mean of the latent variables $f_x$ and $f_y$. These are then placed in the appropriate locations in the $A$ matrix.

```
# adding in mean symbols for x and y
alpha, psi, mux, muy = sm.symbols(('alpha',
'psi','mu_x','mu_y'))
# expand the A and S matrices to include the unit constant
A = sm.zeros(6,6)
A[3,0]=mux # from constant to fx
A[4,0]=muy # from constant to fy
A[4,3]=alpha # regression term
A[1,3]=A[2,4]=A[4,5]=1 # latent variable identification
Ainv = (sm.eye(6)-A).inv() # prepare for expect-
ation function
```

As illustrated below, the choice was made to place the constant term as the first row and column of the three RAM notation matrices.

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ \mu_x & 0 & 0 & 0 & 0 & 0 \\ \mu_y & 0 & 0 & \alpha & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Specification of the remaining matrices is below.

```
S = sm.zeros(6,6)
S[0,0]=1 # unit variance for the constant term
S[1,1]=2
S[2,2]=4
S[3,3]=8
S[5,5]=psi
```

$$S = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \psi \end{bmatrix}$$

The inclusion of the constant term is specified in the **F** matrix as another observed variable.

```
# unit constant is considered observed
F = sm.eye(3).row_join(sm.zeros(3,3))

# expectation
rexp = F*Ainv*S*Ainv.T*F.T
```

$$\begin{bmatrix} 1 & \mu_x & \alpha\mu_x + \mu_y \\ \mu_x & \mu_x^2 + 10 & 8\alpha + \mu_x(\alpha\mu_x + \mu_y) \\ \alpha\mu_x + \mu_y & 8\alpha + \mu_x(\alpha\mu_x + \mu_y) & 8\alpha^2 + \psi + (\alpha\mu_x + \mu_y)^2 + 4 \end{bmatrix}$$

From the expectations computed above, the first row and column of the $\hat{\Sigma}$ matrix includes the constant and mean estimates. This requires that the observed matrix also include these terms, thus turning the $S$ matrix from a covariance matrix, into a moment matrix.

The accompanying python code outlines steps for the computation of a raw moment matrix from the number of observations $N = 100$, the vector of observed means for both $\bar{x} = 50$ and $\bar{y} = 125$, and the observed covariance matrix $S$. The first step involves adding together the cross-product of the mean vector weighted by the sample size $N$ and the $S$ matrix weighted by $N - 1$. This sum is then divided by the sample size. The next step involves adding the constant term along with mean vector in an appropriate place for the model, in this illustration these terms are added as the first row and column of the moment matrix.

```
#observed data
Sob = sm.Matrix([[5, 10, 20]])
N=100 # number of observations
Mob = sm.Matrix([50,125]) # observed mean vector
# sum of squares and cross products (beginning of moment
matrix)
rawSSCP = (N*(Mob * Mob.T) + (N-1)*Sob)/N
rawMM = rawSSCP.col_insert(0,Mob)
rawMM = rawMM.row_insert(0,sm.Matrix
([[1,50,125]]))
```

$$\begin{bmatrix} 1 & 50 & 125 \\ 50 & \frac{25099}{10} & \frac{125099}{20} \\ 125 & \frac{125099}{20} & \frac{78224}{5} \end{bmatrix}$$

With the mean estimates incorporated into the model parameters and the moment matrix, the steps for parameter estimation proceed as before.

```
# determinants
MM_det = rawMM.det()
rexp_det = rexp.det()
```

```
# inverse
rexp_inv = rexp.inv()
# objective function
obj_fun = sm.log(rexp_det) + (rawMM*rexp_
inv).trace() - sm.log(MM_det)
```

There is a notable programmatic difference in how this model is fit to the data. The following code utilizes the optimize routines available in the scipy package. This differs from the earlier example which provided code for the Newton-Raphson approach to root finding. To take advantage of these optimization tools, the objective function is converted to a numpy representation and is further used to define a function opt_fun() which simplifies the process of passing in multiple parameters to the optimization step.

```
# parameter list
parms = [mux,muy,alpha,psi]
# lambdify the objective function, turn it into a numpy rou-
tine
fit_fun = sm.lambdify((mux,muy,alpha,
psi),obj_fun,'numpy')
# define the optimization function (easier to pass to opti-
mizer)
def opt_fun(ps):
  mx, my, a, p = ps # un-tuple the parameters
  return fit_fun(mx,my,a,p)
```

The opt_fun() function defined above can then be used with the minimize routine from the scipy.optimize library, here prefixed with op. The specific search routine selected is the *L-BFGS-B* routine, a version of the Broyden–Fletcher–Goldfarb–Shanno algorithm (Broyde, 1970).

```
# begin optimization
# import scipy.optimize for numerical optimization
import scipy.optimize as op
# starting values
x0 = (40.0,100.0,0.5,10.0)
# find the function minimum using L-BFGS-B
opt_res = op.minimize(opt_fun, x0, method=
'L-BFGS-B')
# package up the results for display
opt_res_df = pd.DataFrame(opt_res.x,
  columns=['Estimate'],
  index=['$\\mu_x$','$\\mu_
y$','$\\alpha$','$\\psi$'])
```

Once again, the model results are compared to estimates obtained from *lavaan*.

```
FnH_mod2 = '
fx =~ 1*X
fy =~ 1*Y
dfy =~ 1*fy

fy ~ alph*fx
dfy ~~ psi*dfy
X ~~ 2*X
Y ~~ 4*Y
fy ~~ 0*fy
fx ~~ 8*fx

fx ~~ 0*dfy
# adding means
fx ~ 1
fy ~ 1
X ~ 0
Y ~ 0
'
# observed mean vector
mus = c(50,125)
cov_ob = getCov(lwr,lower = T,names=c('X','Y'))
# fit the data
fit2 = sem(FnH_mod2,
        sample.cov = cov_ob,
        sample.mean = mus,
        sample.nobs = 100, # arbitrary number of
        observations to fit the model.
        int.ov.free = F)
```

**Table 3.** Python model results with mean estimates.

| Parameter | Estimate |
|---|---|
| $\mu_x$ | 50.000 |
| $\mu_y$ | 93.804 |
| $\alpha$ | 0.624 |
| $\psi$ | 12.700 |

**Table 4.** Lavaan comparisons results with mean estimates.

| Parameter | Estimate |
|---|---|
| $\mu_x$ | 50.000 |
| $\mu_y$ | 93.750 |
| $\alpha$ | 0.625 |
| $\psi$ | 12.700 |

The results from both Python and *lavaan* agree exactly for, $\mu_x = 50.0$, and $\psi = 12.7$, with only minor discrepancies for $\mu_y = 93.804$ and $\alpha = 0.624$ from python compared to $\mu_y = 93.750$ and $\alpha = 0.625$ from *lavaan* (Tables 3–4).

## 10. Conclusion

In conclusion, this presentation aims to provide an accessible demonstration of SEM through the lens of Python programming. By presenting both the theoretical foundations and numerical implementation, the aim is to empower students, researchers, and practitioners with a deeper understanding of the steps involved in SEM. The Python code snippets presented throughout the manuscript serve as a valuable resource for readers seeking to explore SEM techniques at a much deeper level and with their own data.

Furthermore, the adoption of open-source tools, such as the ones demonstrated in this manuscript, fosters a collaborative and transparent research environment. The code shared here not only facilitates the replication of analyses but also encourages the community to contribute, refine, and extend the methodologies presented.

As the popularity and use of SEM continues to grow, the hope is that the code provided serves as a foundation for teaching and understanding the numerical steps involved in SEM. Readers are encouraged to explore, adapt, and build upon the examples presented, thereby contributing to the collective knowledge and refinement of SEM methodologies.

In closing, the hope that this presentation equips readers with the knowledge and practical steps needed to fill in any potential gaps in their understanding of SEM. May the insights gained from this work inspire continued exploration and application of SEM in diverse domains, ultimately advancing our understanding of complex relationships within intricate systems.

## Funding

Research reported in this publication was supported by the National Institute Of General Medical Sciences and the National Institutes on Aging of the National Institutes of Health under Award Numbers P20GM139759 and P30AG066518. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

## ORCID

Joel S. Steele http://orcid.org/0009-0008-4127-5664
Kevin J. Grimm http://orcid.org/0000-0002-8576-4469

## References

Andersen, M. M., & Højsgaard, S. (2019). Ryacas: A computer algebra system in r. *Journal of Open Source Software*, *4*, 1763. https://doi.org/10.21105/joss.01763

Andersen, M. M., & Højsgaard, S. (2021). Caracas: Computer algebra in r. *Journal of Open Source Software*, *6*, 3438. https://doi.org/10.21105/joss.03438

Asparouhov, T., & Muthen, B. (2006). Comparison of estimation methods for complex survey data analysis. *Mplus Web Notes*, 1–13. http://www.statmodel.com/download/SurveyComp21.pdf

Broyden, C. G. (1970). The convergence of a class of double-rank minimization algorithms 1. General considerations. *IMA Journal of Applied Mathematics*, *6*, 76–90. https://doi.org/10.1093/imamat/6.1.76

Cheung, M. (2020). *symSEM: Symbolic computation for structural equation models*. R Package Version, 0, 1.

Ferron, J. M., & Hess, M. R. (2007). Estimation in SEM: A concrete example. *Journal of Educational and Behavioral Statistics*, *32*, 110–120. https://doi.org/10.3102/1076998606298025

Goldsmith, K. A., MacKinnon, D. P., Chalder, T., White, P. D., Sharpe, M., & Pickles, A. (2018). Tutorial: The practical application of longitudinal structural equation mediation models in clinical trials. *Psychological Methods*, *23*, 191–207. https://doi.org/10.1037/met0000154

Grimm, K. J., & McArdle, J. J. (2005). A note on the computer generation of mean and covariance expectations in latent growth curve analysis. In *Multi-level issues in strategy and methods* (pp. 335–364). Emerald Group Publishing Limited.

Gunzler, D., Chen, T., Wu, P., & Zhang, H. (2013). Introduction to mediation analysis with structural equation modeling. *Shanghai Archives of Psychiatry*, *25*, 390.

Jöreskog, K. G., & Sörbom, D. (1982). Recent developments in structural equation modeling. *Journal of Marketing Research*, *19*, 404–416. https://doi.org/10.1177/002224378201900402

Ledermann, T., Macho, S., & Kenny, D. A. (2011). Assessing mediation in dyadic data using the actor-partner interdependence model. *Structural Equation Modeling: A Multidisciplinary Journal*, *18*, 595–612. https://doi.org/10.1080/10705511.2011.607099

McArdle, J. J., & Bell, R. Q. (2000). An introduction to latent growth models for developmental data analysis. In T. D. Little, K. U. Schnabel, & J. Baumert (Eds.), *Modeling longitudinal and multiple-group data: Practical issues, applied approaches, and scientific examples* (pp. 69–107). Erlbaum.

McArdle, J. J., & McDonald, R. P. (1984). Some algebraic properties of the reticular action model for moment structures. *The British Journal of Mathematical and Statistical Psychology*, *37 (Pt 2)*, 234–251. https://doi.org/10.1111/j.2044-8317.1984.tb00802.x

Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., Rathnayake, T., Vig, S., Granger, B. E., Muller, R. P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., … Scopatz, A. (2017). SymPy: Symbolic computing in python. *PeerJ Computer Science*, *3*, e103. https://doi.org/10.7717/peerj-cs.103

Muthén, B. O. (2002). Beyond SEM: General latent variable modeling. *Behaviormetrika*, *29*, 81–117. https://doi.org/10.2333/bhmk.29.81

Oertzen, T. V., & Brick, T. R. (2014). Efficient hessian computation using sparse matrix derivatives in RAM notation. *Behavior Research Methods*, *46*, 385–395. https://doi.org/10.3758/s13428-013-0384-4

Preacher, K. J. (2010). Latent growth curve models. *The Reviewer's Guide to Quantitative Methods in the Social Sciences*, *1*, 185–198.

Rosseel, Y. (2012). lavaan: An R package for structural equation modeling. *Journal of Statistical Software*, *48*, 1–36. https://doi.org/10.18637/jss.v048.i02

Van Rossum, G., & Drake, F. L. (2009). *Python 3 reference manual.* CreateSpace.

## Appendix A: Python Code for Ferron and Hess Example

### *Section 1: Netwon-Raphson approach*

```
import sympy as sm # import Symbolic Python module
sm.init_printing() # initialize printing so the latex symbols render

#create our math symbols
alpha, psi = sm.symbols(('alpha', 'psi'))

S = sm.zeros(5,5) # create the slings matrix
for i in [0,1,2]: # fill in the numeric constants
  S[i,i] = 2**(i+1)

S[4,4]=psi # add in the free parameter
A = sm.zeros(5,5) # create the arrows matrix
A[0,2]=A[1,3]=A[3,4] = 1 # fill in numeric constants
A[3,2]=alpha # add in the free parameter

F = sm.eye(2).row_join(sm.zeros(2,3)) # create the filter matrix

Ainv = (sm.eye(5)-A).inv() # subtract A from I and invert

RAMexp = F*Ainv*S*Ainv.T*F.T # model based expectation of
covariance matrix
mex_det = RAMexp.det() # determinant of the expectation matrix
mex_inv = RAMexp.inv() # inverse of the expectation matrix
Sob = sm.Matrix([[5, 10, 20]]) # observed covariance matrix
Sob_det = Sob.det() # determinant of observed covariance matrix

# create the objective function
objective = sm.log(mex_det) + (Sob*mex_inv).trace() -
sm.log(Sob_det)
ml_obj = objective.simplify()

d1a = sm.diff(ml_obj,alpha).simplify() # first derivative with
respect to alpha
d1p = sm.diff(ml_obj,psi).simplify() # first derivative with
```

```
respect to psi
fitgrad = sm.Matrix([d1a,d1p]) # use derivatives to create the
gradient vector

d2a = sm.diff(d1a,alpha).simplify() # second derivative for
alpha
d2p = sm.diff(d1p,psi).simplify() # second derivative for psi
# mixed partial using first derivative for alpha, with respect to psi
d2ap = sm.diff(d1a,psi).simplify()
# mixed partial using first derivative for psi, with respect to alpha
d2pa = sm.diff(d1p,alpha).simplify()
# create the Hessian matrix
fithess = sm.Matrix([[d2a,d2ap],[d2pa,d2p]])

# Newton based root finding algorithm
root_func = sm.Matrix([alpha,psi]) - fithess.inv()*fitgrad
# evaluation the root finding function with starting values
res1 = root_func.subs([(alpha,0.5),(psi,5.0)])

# import pandas for easy data manipulation
import pandas as pd # for collecting results and display
res_df = pd.DataFrame(columns = ['itr','a','p','gr_
a','gr_p','f_obj'])
tcount = 0 # loop limit iterator
epsilon = 0.0000000003 # gradient convergence limit
pres = [0.5,5.0] # starting values for parameters
# initial evaluation for estimation
# another way to evaluate a function in sympy
# start with gradient at starting values
grad = fitgrad.evalf(subs = {alpha:pres[0],psi:pres[1]})
# objective function value at starting values
obj = ml_obj.evalf(subs = {alpha:pres[0],psi:pres[1]})
# package up results in a dictionary
outres = {"itr":tcount, "a":pres[0],"p":pres[1],
         "gr_a":grad[0],"gr_p":grad[1], "f_obj":obj}
# add it to the results DataFrame for display later.
res_df = pd.concat([res_df,pd.DataFrame([outres])])

# track the gradient until one value is below epsilon or we pass 20 iterations
while (abs(grad[0]) > epsilon or abs(grad[1]) > epsilon)
and tcount < 20:
  # current results of root finding function (next values to try)
  cres = root_func.evalf(subs = {alpha:pres[0],psi:pres[1]})
  # gradient evaluation using current results, function slope at these points
  grad = fitgrad.evalf(subs = {alpha:cres[0],psi:cres[1]})
  # objective function evaluation using current results
  obj = ml_obj.evalf(subs = {alpha:cres[0],psi:cres[1]})
  # package up results for output
  outres = {"itr":tcount + 1,
           "a":cres[0],"p":cres[1],
           "gr_a":grad[0],"gr_p":grad[1],
           "f_obj":obj}
    # add results of this iteration and update for next loop
    res_df = pd.concat([res_df,pd.DataFrame([outres])])
    # swap previous results for current results for next loop
    pres = cres
    # increment the iterations counter
    tcount += 1
```

### Section 2: Mean estimates with L-BFGS-B Approach

```
# adding in mean symbols for x and y
alpha, psi, mux, muy = sm.symbols(('alpha','psi','mu_
x','mu_y'))

# expand the A and S matrices to include the unit constant
A = sm.zeros(6,6)
A[3,0]=mux # from constant to fx
A[4,0]=muy # from constant to fy
A[4,3]=alpha # regression term
A[1,3]=A[2,4]=A[4,5] = 1 # latent variable identification
Ainv = (sm.eye(6)-A).inv() # prepare for expectation function

S = sm.zeros(6,6)
S[0,0] = 1 # unit variance for the constant term
S[1,1] = 2
S[2,2] = 4
S[3,3] = 8
S[5,5]=psi
# unit constant is considered observed
F = sm.eye(3).row_join(sm.zeros(3,3))

# expectation
rexp = F*Ainv*S*Ainv.T*F.T

#observed data
Sob = sm.Matrix([[5, 10, 20]])
N = 100 # number of observations
Mob = sm.Matrix([50,125]) # observed mean vector
# sum of squares and cross products (beginning of moment matrix)
rawSSCP = (N*(Mob * Mob.T) + (N-1)*Sob)/N
rawMM = rawSSCP.col_insert(0,Mob)
rawMM = rawMM.row_insert(0,sm.Matrix([[1,50,125]]))

# determinants
MM_det = rawMM.det()
rexp_det = rexp.det()
# inverse
rexp_inv = rexp.inv()
# objective function
obj_fun = sm.log(rexp_det) + (rawMM*rexp_inv).trace() -
sm.log(MM_det)

# parameter list
parms = [mux,muy,alpha,psi]
# lambdify the objective function, turn it into a numpy routine
fit_fun = sm.lambdify((mux,muy,alpha,psi),obj_fun,
'numpy')
# define the optimization function (easier to pass to optimizer)
def opt_fun(ps):
  mx, my, a, p = ps # un-tuple the parameters
  return fit_fun(mx,my,a,p)

# begin optimization
# import scipy.optimize for numerical optimization
import scipy.optimize as op
# starting values
x0 = (40.0,100.0,0.5,10.0)
# find the function minimum using L-BFGS-B
opt_res = op.minimize(opt_fun, x0, method='L-BFGS-B')
# package up the results for display
opt_res_df = pd.DataFrame(opt_res.x,
  columns = ['Estimate'],
  index = ['$\\mu_x$','$\\mu_y$','$\\alpha$','$\\psi$'])
```

## Appendix B: R Code for Ferron and Hess Example

### Section 1: Covariance Estimates

```
suppressPackageStartupMessages(library(lavaan))
FnH_mod = '
fx =~ 1*X
fy =~ 1*Y
dfy =~ 1*fy

fy ~ alph*fx
dfy ~~ psi*dfy
```

```
X ~~ 2*X
Y ~~ 4*Y
fy ~~ 0*fy
fx ~~ 8*fx

fx ~~ 0*dfy
'
# observed variance matrix
lwr = '
10
5 20'
cov_ob = getCov(lwr,lower = T,names=c('X','Y'))
# fit the data
fit = sem(FnH_mod,
          sample.cov = cov_ob,
          sample.nobs = 100, # arbitrary number of observa-
          tions to fit the model.
          int.ov.free = F)
```

## Section 2: Covariance and Mean estimates

```
FnH_mod2 = '
fx =~ 1*X
fy =~ 1*Y
dfy =~ 1*fy

fy ~ alph*fx
dfy ~~ psi*dfy

X ~~ 2*X
Y ~~ 4*Y
fy ~~ 0*fy
fx ~~ 8*fx

fx ~~ 0*dfy
# adding means
fx ~ 1
fy ~ 1
X ~ 0
Y ~ 0
'

# observed mean vector
mus = c(50,125)
cov_ob = getCov(lwr,lower = T,names=c('X','Y'))
# fit the data
fit2 = sem(FnH_mod2,
          sample.cov = cov_ob,
          sample.mean = mus,
          sample.nobs = 100, # arbitrary number of observa-
          tions to fit the model.
          int.ov.free = F)
```