Appendix B

Symbolic Mathematics with `Sage`

B.1 Introduction to `Sage`

What is `Sage`, and why use it?

`Sage` is free, open source mathematics software. Lots of software can carry out numerical calculations, and so can `Sage`. What makes `Sage` special is that it can also do *symbolic* computation. That is, it is able to manipulate symbols as well as numbers.

If you think about it, you will realize that a lot of the "mathematics" you do in your statistics courses does not really require much mathematical thinking. Sometimes, all you are really doing is pushing symbols around. You might have to do something like partially differentiate a log likelihood function with respect to several variables, set all the expressions to zero and solve the resulting equations. To do this you need to know some rules, apply them accurately, and pay attention to detail. This kind of "thinking" is something that computers do a lot better than humans. So particularly for big, complicated tasks, why not let a computer do the grunt work? Symbolic mathematics software is designed for this purpose.

There are several commercial products that do symbolic math. The best known are Mathematica (http://www.wolfram.com) and Maple (http://www.maplesoft.com). There are also quite a few free, open source alternatives that are developed and maintained by volunteers. `Sage` is one of them. What makes `Sage` really special is that in addition to its own core capabilities, it incorporates and more or less unifies quite a few of the other mathematical programs using a single convenient interface. After all, why not? They are free and open source, so there are no legal obstacles (like copyrights) to prevent the Sage programmers from sending a particular task to the program that does it best[1].

It's all accomplished with Python scripts. In fact, `Sage` is largely a set of sophisticated Python functions. So if you know the Python programming language, you have a huge head start in learning `Sage`. If you want to do something in `Sage` and you can figure out how to do it in Python, try it. Probably the Python code will work.

[1]A by-product of this approach is that if you download a copy of `Sage`, you'll see that it's *huge*. This is because you're really downloading six or seven complete applications.

Reference Materials

This appendix is intended to be more or less complete. For further information and documentation, see the Sage project home page at http://www.sagemath.org. Another useful source of information is the Wikipedia article:

http://en.wikipedia.org/wiki/Sage_(mathematics_software)

A Guided tour

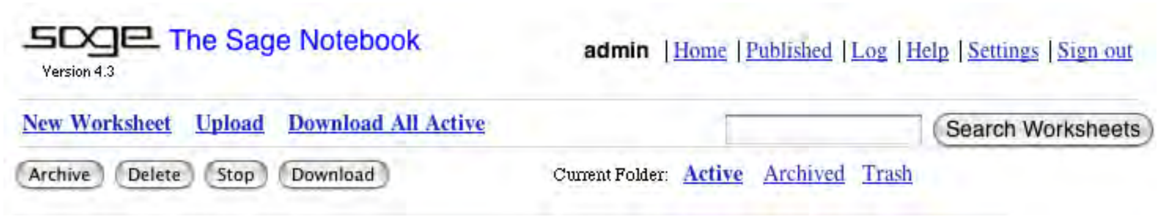
To follow this tour actively by trying things out as you read about them, you will need access to Sage, either on your computer or on a server. For more information, see Section B.3: Using Sage on your Computer.

The Interface

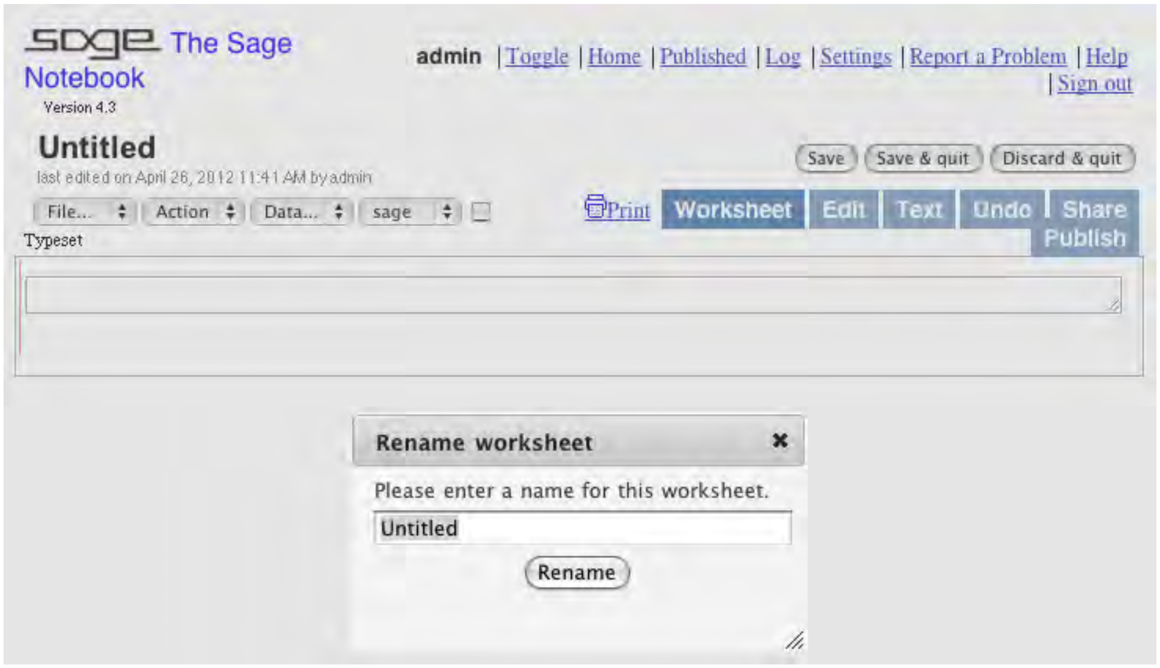
Sage has a browser interface. So, whether the software resides on a remote server or you have downloaded and installed your own free copy as described in Section B.3, you type your input and see your output using an ordinary Web browser like Firefox.

Sage also has a text-only interface, in which the output as well as input is in plain text format. Many mathematicians who use Sage prefer the simplicity of plain text, and most Sage documentation uses plan text. But a great strength of Sage, and our main reason for using it, is that we can manipulate and view the results of calculations using Greek symbols. This capability depends on the browser interface, so we'll stick exclusively to that.

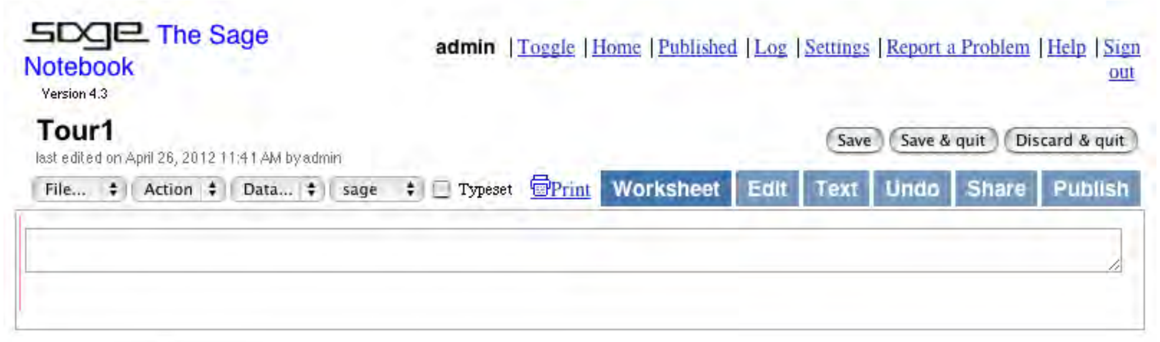
When you first start up Sage, you'll see the Sage *Notebook* with a list of your active *Worksheets*. You can save your worksheets and go back to them later. It's great, but right now you don't have any worksheets. Your screen looks roughly like this:


sage The Sage Notebook
Version 4.3
admin | Home | Published | Log | Help | Settings | Sign out
New Worksheet Upload Download All Active
Search Worksheets
Archive Delete Stop Download
Current Folder: Active Archived Trash

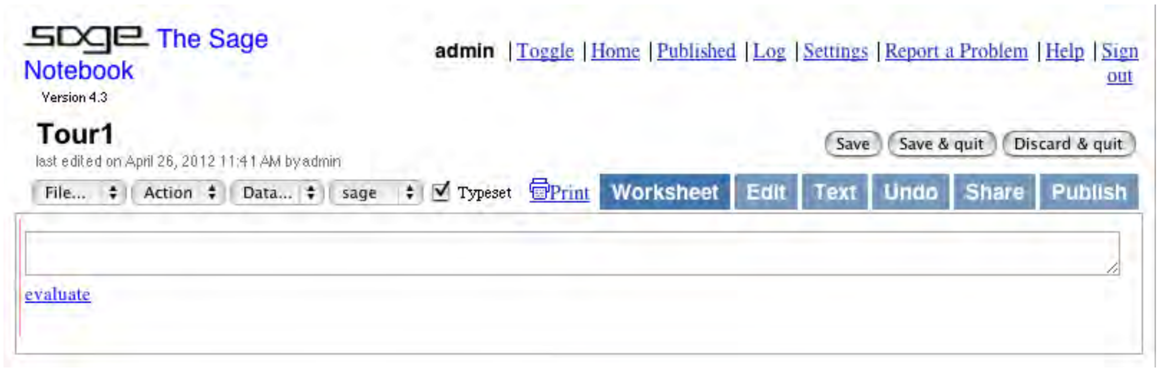
Click on "New Worksheet." A new window opens. It looks like this:


The Sage
Notebook
Version 4.3
admin | Toggle | Home | Published | Log | Settings | Report a Problem | Help | Sign out
Untitled
last edited on April 26, 2012 11:41 AM by admin
Save | Save & quit | Discard & quit
File... | Action | Data... | sage | Typeset
Print | Worksheet | Edit | Text | Undo | Share | Publish
Rename worksheet
Please enter a name for this worksheet.
Untitled
Rename

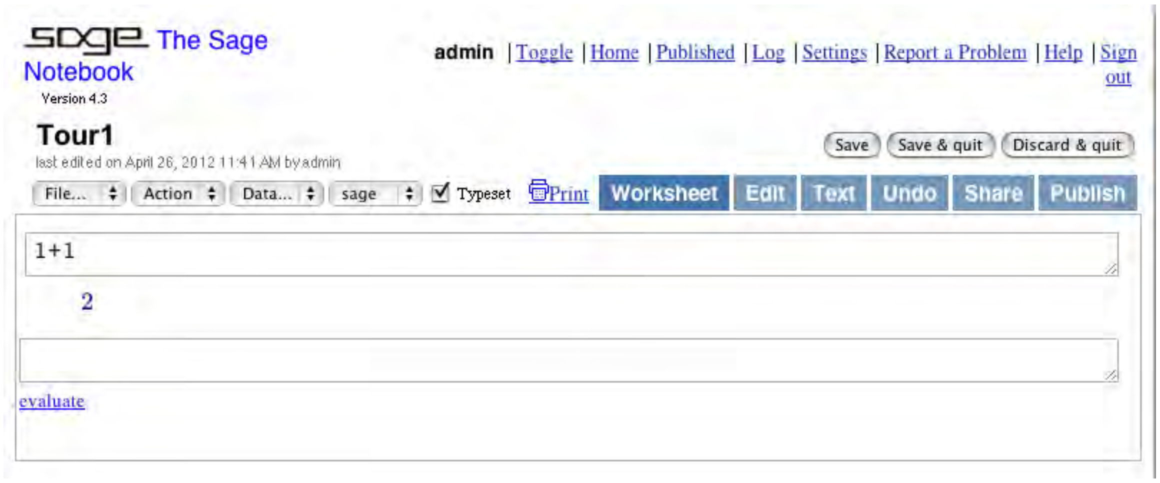
Type in a nice informative name and click Rename. I called mine Tour1, because we're on a guided tour of Sage. Now the browser window looks like something like this:


The Sage
Notebook
Version 4.3
admin | Toggle | Home | Published | Log | Settings | Report a Problem | Help | Sign out
Tour1
last edited on April 26, 2012 11:41 AM by admin
Save | Save & quit | Discard & quit
File... | Action | Data... | sage | Typeset
Print | Worksheet | Edit | Text | Undo | Share | Publish

You definitely want to check the "Typeset" box, so you can see nice Greek letters. Now, the way it works is that you type (or paste) your commands into the upper box and Sage writes the output in the box below it. As soon as you click in the upper box, the underlined word evaluate appears below. It looks like this.

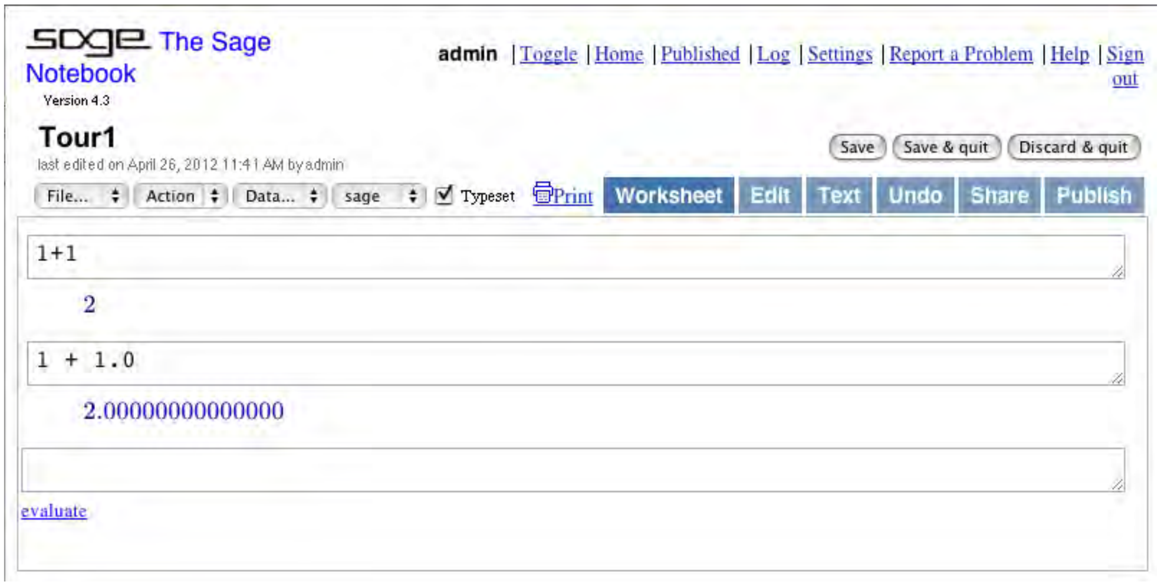

The Sage
Notebook
Version 4.3
admin | Toggle | Home | Published | Log | Settings | Report a Problem | Help | Sign out
Tour1
last edited on April 26, 2012 11:41 AM by admin
Save | Save & quit | Discard & quit
File... | Action | Data... | sage | ✓ Typeset
Print | Worksheet | Edit | Text | Undo | Share | Publish
evaluate

Now you type your input, which in this case is numerical as well as mathematically profound. Pressing the Enter (or Return) key just lets you type another line of input. To execute the command(s), click evaluate. An alternative to clicking evaluate is to hold down the Shift key and press Enter. Here is the result.



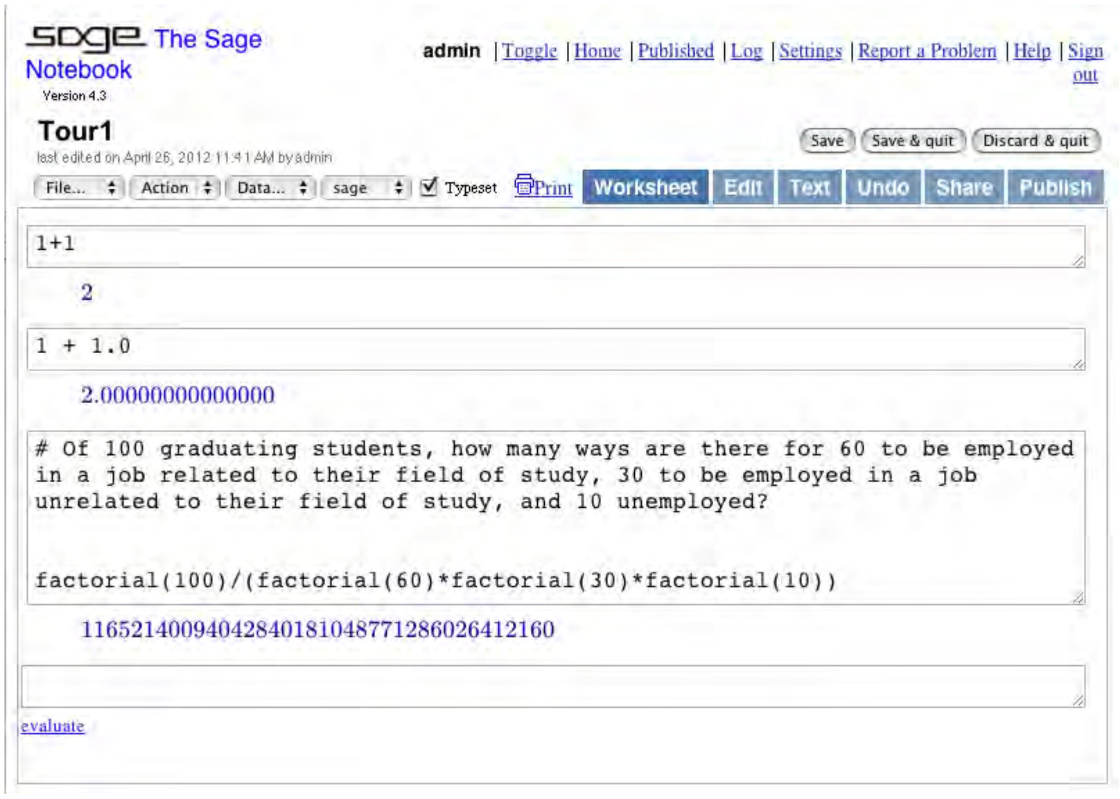
```
1+1
    2
```

Notice that now there's another box for your next set of input. Here's a variation on $1 + 1 = 2$.



```
1+1
    2
1 + 1.0
    2.00000000000000
```

In the first case, Sage was doing integer arithmetic. In the second case, part of the input was interpreted as real-valued because it had a decimal point. Integer plus real is real, so Sage converted the 1 to 1.0 and did a floating-point calculation. This kind of "dynamic typing" is a virtue that Sage shares with Python. Sage is very good at integer arithmetic. In the next example, everything following # is a comment.



```
1+1
```

```
2
```

```
1 + 1.0
```

```
2.00000000000000
```

```
# Of 100 graduating students, how many ways are there for 60 to be employed
in a job related to their field of study, 30 to be employed in a job
unrelated to their field of study, and 10 unemployed?


factorial(100)/(factorial(60)*factorial(30)*factorial(10))
```

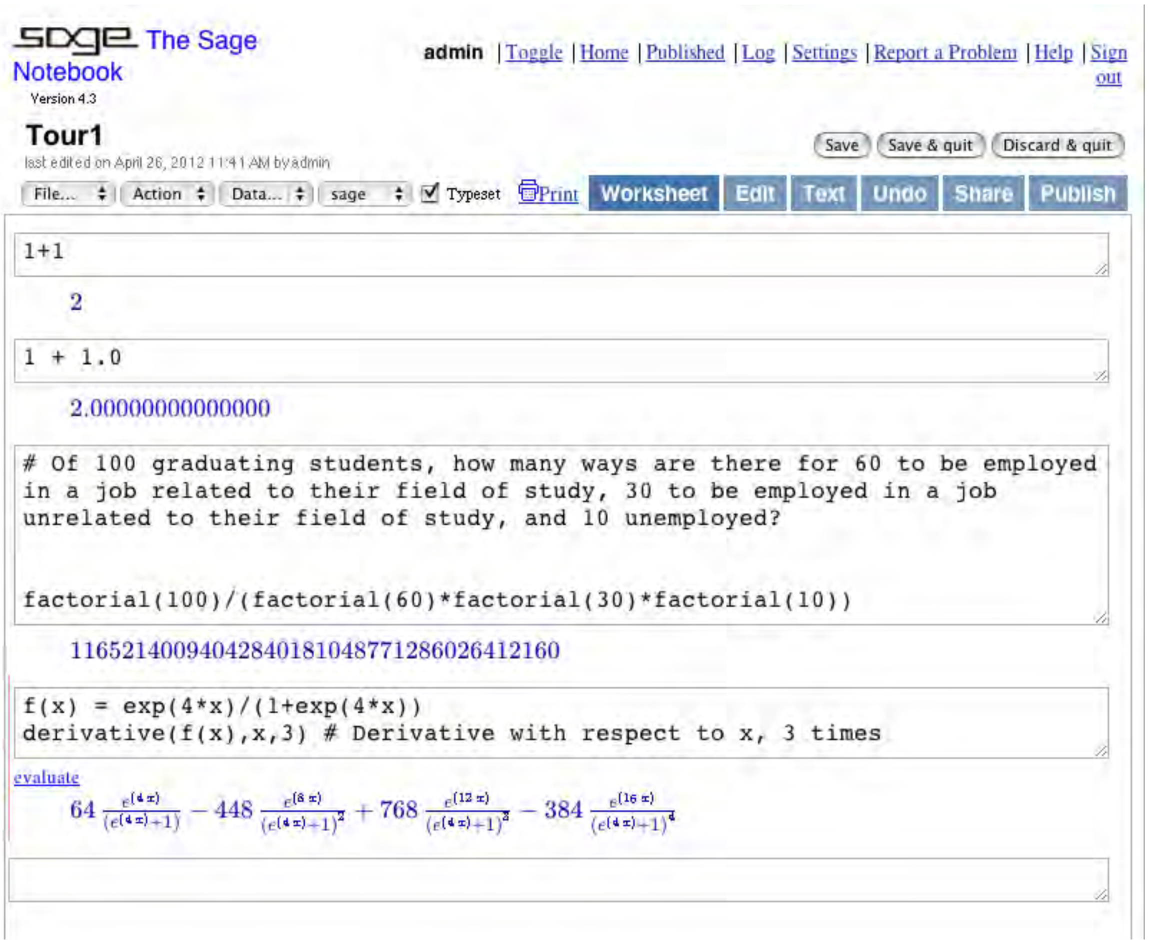
```
1165214009404284018104877128602641216O
```

For comparison, this is how the calculation goes in R.

```
> prod(1:100)/(prod(1:60)*prod(1:30)*prod(1:10))
> prod(1:100)/(prod(1:60)*prod(1:30)*prod(1:10))
[1] 1.165214e+37
```

The whole thing is a floating point calculation, and R returns the answer in an imprecise scientific notation.

Exact integer arithmetic is nice, but it's not why we're using Sage. Let's calculate the third derivative $\frac{\partial^3}{\partial x^3}\left(\frac{e^{4x}}{1+e^{4x}}\right)$. This is something you could do by hand, but would you want to?



```
1+1
```

```
2
```

```
1 + 1.0
```

```
2.00000000000000
```

```
# Of 100 graduating students, how many ways are there for 60 to be employed
in a job related to their field of study, 30 to be employed in a job
unrelated to their field of study, and 10 unemployed?


factorial(100)/(factorial(60)*factorial(30)*factorial(10))
```

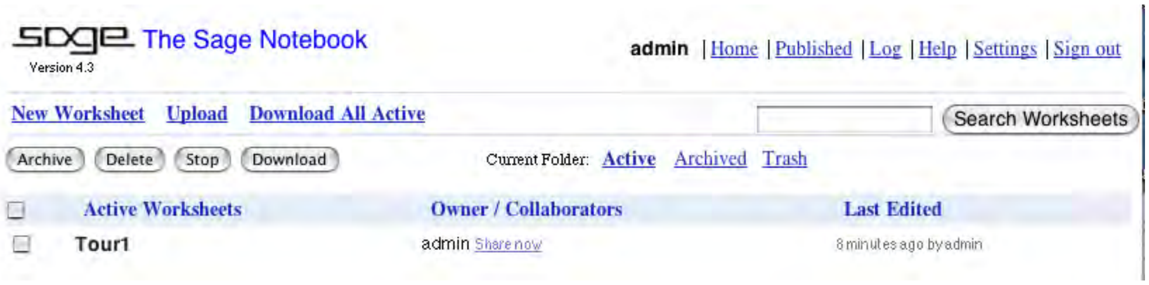
```
11652140094042840181048771286026412160
```

```
f(x) = exp(4*x)/(1+exp(4*x))
derivative(f(x),x,3) # Derivative with respect to x, 3 times
```

$$64 \frac{e^{(4x)}}{(e^{(4x)}+1)} - 448 \frac{e^{(8x)}}{(e^{(4x)}+1)^2} + 768 \frac{e^{(12x)}}{(e^{(4x)}+1)^3} - 384 \frac{e^{(16x)}}{(e^{(4x)}+1)^4}$$

You can see how the worksheet grows. At any time, you can click on the Save button if you like what you have. You can also print it just as you would any other Web page.

You can edit the contents of an input box by clicking in the box. When you do, evaluate appears beneath the box. Click on it, and the code in the box is executed. You can re-do all the calculations in order by choosing Evaluate All from the Action menu (upper left). When you quit Sage and come back to a worksheet later, you may want to Evaluate All so all the objects you've defined – like $f(x)$ above – are available. When you're done (for the present), click the Save & Quit button. If you click Discard & Quit, all the material since the last Save will be lost; sometimes this is what you want. When you Save & Quit, you see something like this:



	Active Worksheets	Owner / Collaborators	Last Edited
☐	Tour1	admin Share now	8 minutes ago by admin

Click on Sign out (upper right) and you're done. Next time you run Sage the worksheet will be available. You can double-click on it to work on it some more, or start a new one.

The guided tour will resume now, but without continuing to illustrate the interface. Instead, the input will be given in a typewriter typeface `like this`, and then the output will given, usually in typeset form[2].

Limits, Integrals and Derivatives (Plus a little plotting and solving)

Now we return to the Tour1 worksheet and choose Evaluate All from the Action menu. Then

```
f(x)
```

and clicking on evaluate yields

$$\frac{e^{\left(4 \, x\right)}}{{\left(e^{\left(4 \, x\right)} + 1\right)}}$$

This really looks like a cumulative distribution function. Is it? Let's try $\lim_{x \to -\infty} f(x)$.

```
limit(f(x),x=-Infinity);limit(f(x),x=Infinity)
```

evaluate

0
1

Okay! So it's a distribution function. Notice the two commands on the same line, separated by a semi-colon. Without the semi-colon, only the last item is displayed. An alternative to the semi-colon is the `show command`:

```
show(limit(f(x),x=-Infinity))
show(limit(f(x),x=Infinity))
```

evaluate

0

1

The (single) derivative of $f(x)$ is a density.

```
derivative(f(x),x)
```

[2]In case you are interested in how this works, Sage uses the open source LaTeX typesetting system to produce output in mathematical script. The LaTeX code produced by Sage is available. So, in the Tour1 worksheet, if I enter `f(x)` in the input box, I get nice-looking mathematical output (see above). Then if I type `print(latex(_))` in the next input box, I get the LaTeX code for the preceding expression. Since this book is written in LaTeX, I can directly paste in the machine-generated LaTeX code without having to typeset it myself. My code might be a bit cleaner and more human-readable, but this is very convenient.

evaluate

$$4\,\frac{e^{(4\,x)}}{\left(e^{(4\,x)}+1\right)} - 4\,\frac{e^{(8\,x)}}{\left(e^{(4\,x)}+1\right)^2}$$

Here is another way to get the same thing.

```
# Another way
f(x).derivative(x)
```

evaluate

$$\frac{4\,e^{(4\,x)}}{e^{(4\,x)}+1} - \frac{4\,e^{(8\,x)}}{\left(e^{(4\,x)}+1\right)^2}$$

This second version of the syntax is more like Python, and makes it clear that the derivative is an *attribute*, or *method* associated with the object $f(x)$. Many tasks can be requested either way, but frequently only the second form (object followed by a dot, followed by the attribute) is available. It is preferable from a programming perspective.

The expression for $f'(x)$ could and should be simplified. `Sage` has a `simplify` command that does nothing in this case and in many others, because `simplify` is automatically applied before any expression is displayed. But `factor` does the trick nicely.

```
g(x) = factor(f(x).derivative(x)); g(x)
```

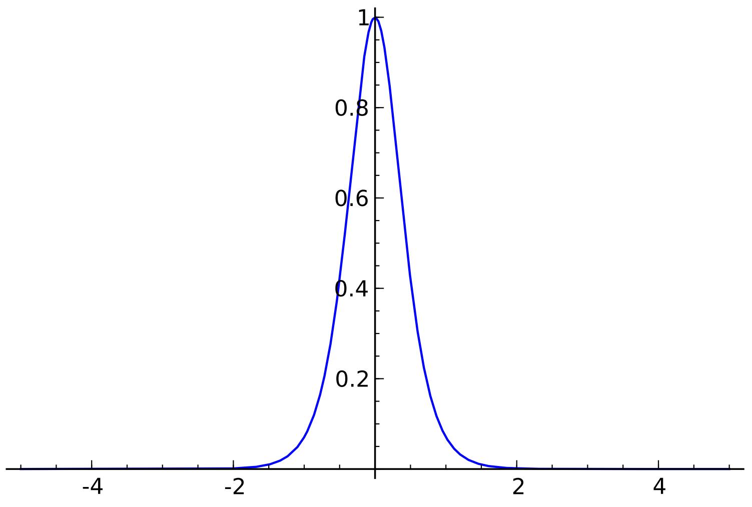
evaluate

$$\frac{4\,e^{(4\,x)}}{\left(e^{(4\,x)}+1\right)^2}$$

Want to see what it looks like? Plotting functions is straightforward.

```
plot(g(x),x,-5,5)
```

evaluate



It's easy to add labels and so on to make the plot look nicer, but that's not the point here. The objective was just to take a quick look to see what's going on.

Actually, the picture is a bit surprising. It *looks* like the density is symmetric around $x = 0$, which would make the median and the mean both equal to zero. But the formula for $g(x)$ above does not suggest symmetry. Well, it's easy to verify that the median is zero.

```
f(0)
```

evaluate

$\frac{1}{2}$

How about symmetry? The first try is unsuccessful, because the answer is not obviously zero (though it is). But then `factor` works.

```
g(x)-g(-x)
```

evaluate

$$\frac{4\,e^{(4\,x)}}{\left(e^{(4\,x)}+1\right)^2} - \frac{4\,e^{(-4\,x)}}{\left(e^{(-4\,x)}+1\right)^2}$$

```
factor(g(x)-g(-x))
```

evaluate

0

Is this right? Yes. To see it, just multiply numerator and denominator of $g(-x)$ by e^{8x}. Sage does not show its work, but it's a lot less likely to make a mistake than you are. And even if you're the kind of person who likes to prove everything, Sage is handy because it can tell you what you should try to prove.

Clearly, the number 4 in $f(x)$ is arbitrary, and could be any positive number. So we'll replace 4 with θ. Now Sage, like most software, will usually complain if you try to use variables that have not been defined yet. So we have to declare θ as a symbolic variable, using a `var` statement. The variable x is the only symbolic variable that does not have to be declared. It comes pre-defined as symbolic[3].

```
var('theta')
F(x) = exp(theta*x)/(1+exp(theta*x)); F(x)
```

evaluate

$$\frac{e^{(\theta x)}}{e^{(\theta x)}+1}$$

Is $F(x)$ a distribution function? Let's see.

[3]In Mathematica, all variables are symbolic by default unless they are assigned a numeric value. I wish Sage did this too, but I'm not complaining. Sage has other strengths that Mathematica lacks.

```
limit(F(x),x=-Infinity)
```

evaluate

```
Traceback (click to the left of this block for traceback)
...
Is  theta  positive, negative, or zero?
```

This is how error messages are displayed. You can click on the blank space to the left of the error message for more information, but in this case it's unnecessary. Sage asks a very good question about θ. Well, actually, the question is asked by the excellent open-source calculus program Maxima, and Sage relays the question. In Maxima, you could answer the question interactively through the console and the calculation would proceed, but this capability is not available in Sage. The necessary information can be provided non-interactively. Go back into the box and edit the text.

```
assume(theta>0)
F(x).limit(x=-oo); F(x).limit(x=oo)
```

evaluate

```
0
1
```

Notice how two small letter o characters can be used instead of typing out Infinity. Now we'll differentiate $F(x)$ to get the density. It will be called $f(x)$, and that will *replace the existing definition* of $f(x)$.

```
f(x) = factor(F(x).derivative(x)); f(x)
```

evaluate

$$\frac{\theta e^{(\theta x)}}{\left(e^{(\theta x)}+1\right)^2}$$

Of course this density is also symmetric about zero, just like the special case with $\theta = 4$. It's easy to verify.

```
factor(f(x)-f(-x))
```

evaluate

```
0
```

Symmetry of the density about zero implies that the expected value is zero, because the expected value is the physical balance point. Direct calculation confirms this.

```
# Expected value
integrate(x*f(x),x,-oo,oo)
```

evaluate

```
0
```

It would be nice to calculate the variance too, but the variance emerges in terms of an obscure function called the `polylog`. The calculation will not be shown.

This distribution (actually, a version of the logistic distribution) is a good source of cute homework problems because the parameter θ has to be estimated numerically. So, for the benefit of some lucky future students, let's figure out how to simulate a random sample from $F(x)$. First, we'll add a location parameter, because two-parameter problems are more fun. The following definition rubs out the previous $F(x)$.

```
# Add a location parameter
var('mu')
F(x) = exp(theta*(x-mu))/(1+exp(theta*(x-mu))); F(x)
```

evaluate

$$\frac{e^{(-(\mu-x)\theta)}}{e^{(-(\mu-x)\theta)}+1}$$

I can't control the order of variables in `Sage` output. It looks alphabetical, with the `m` in `mu` coming before x.

It's well known that if U is a random variable with a uniform density on the interval $(0, 1)$ and $F(x)$ is the cumulative distribution function of a continuous random variable, then if you transform U with the *inverse* of $F(x)$, the result is a random variable with distribution function $F(x)$. Symbolically,

$$F^{-1}(U) = X \sim F(x)$$

Of course this is something you could do by hand, but it's so fast and easy with `Sage`:

```
# Inverse of cdf
var('X U')
solve(F(X)==U,X) # Solve F(X)=U for X
```

evaluate

$$\left[X = \frac{\mu\theta+\log\left(-\frac{U}{U-1}\right)}{\theta}\right]$$

It might be a bit better to write this as

$$X = \mu + \frac{1}{\theta}\log\left(\frac{U}{1-U}\right),$$

but what `Sage` gives us is quite nice. A few technical comments are in order. First, the double equal sign in `F(X)==U` indicates a *logical* relation. For example,

```
1==4
```

evaluate

False

Second, the `solve` returns a *list* of solutions. `Sage` uses brackets to indicate a list. In this case, there is only one solution so the list contains only one element. It's element *zero* in the list, not element one. Like Python, `Sage` starts all lists and array indices with element zero. It's a hard-core computer science feature, and mildly irritating for the ordinary user. Here's how one can extract element zero from the list of solutions.

```
solve(F(X)==U,X)[0]
```

evaluate

$$X = \frac{\mu\theta + \log\left(-\frac{U}{U-1}\right)}{\theta}$$

The equals sign in that last expression is actually a double equals. If you're going to use something like that solution in later calculations, it can matter. In `Sage`, the underscore character always refers to the output of the preceding command. It's quite handy. The `print` function means "Please don't typeset it."

```
print(_)
```

evaluate

```
X == (mu*theta + log(-U/(U - 1)))/theta
```

Just for completeness, here's how that inverse function could be used to simulate data from $F(x)$ in `R`.

```
> n = 20; mu = -2; theta = 4
> U = runif(n)
> X = mu + log(U/(1-U))/theta; X
 [1] -1.994528 -2.455775 -2.389822 -2.996261 -1.477381 -2.422011 -1.855653
 [8] -2.855570 -2.358733 -1.712423 -2.075641 -1.908347 -2.018621 -2.019441
[15] -1.956178 -2.015682 -2.846583 -1.727180 -1.726458 -2.207717
```

Random number generation is available from within `Sage` too, and in fact `R` is one of the programs incorporated in `Sage`, but to me it's more convenient to use `R` directly – probably just because I'm used to it.

You have to declare most variables (like θ, μ, X, U and so on) before you can use them, but there are exceptions. The pre-defined symbolic variable x is one. Here is another.

```
pi
```

evaluate

π

Is that really the ratio of a circle's circumference to its diameter, or just the Greek letter?

```
cos(pi)
```

evaluate

-1

That's pretty promising. Evaluate it numerically.

```
n(pi) # Could also say pi.n()
```

evaluate

3 : 14159265358979

```
gamma(1/2)
```

evaluate

$\sqrt{\pi}$

So it's really π. Let's try using `pi` in the normal distribution.

```
# Normal density
var('mu, sigma')
assume(sigma>0)
f(x) = 1/(sigma*sqrt(2*pi)) * exp(-(x-mu)^2/(2*sigma^2)); f(x)
```

evaluate

$$\frac{\sqrt{2}e^{\left(-\frac{(\mu-x)^2}{2\,\sigma^2}\right)}}{2\,\sqrt{\pi}\sigma}$$

```
# Integrate the density
integrate(f(x),x,-oo,oo)
```

evaluate

1

Calculate the expected value.

```
# E(X)
integrate(x*f(x),x,-oo,oo)
```

evaluate

μ

Obtain the variance directly.

```
# E(X-mu)^2
integrate((x-mu)^2*f(x),x,-oo,oo)
```

evaluate

σ^2

Calculate the moment-generating function and use it to get $E(X^4)$.

```
# Moment-generating function M(t) = E(e^{Xt})
var('t')
M(t) = integrate(exp(x*t)*f(x),x,-oo,oo); M(t)
```

evaluate

$e^{\left(\frac{1}{2}\sigma^2 t^2 + \mu t\right)}$

```
# Differentiate four times, set t=0
derivative(M(t),t,4)(t=0)
```

evaluate

$\mu^4 + 6\,\mu^2\sigma^2 + 3\,\sigma^4$

Discrete distributions are easy to work with, too. In the geometric distribution, a coin with $Pr\{\text{Head}\} = \theta$ is tossed repeatedly, and X is the number of tosses required to get the first head. Notice that two separate `assume` statements are required to establish $0 < \theta < 1$. All the commands work as expected, but only the output from the last one is displayed.

```
# Geometric
var('theta')
assume(0<theta); assume(theta<1)
p(x) = theta*(1-theta)^(x-1); p(x)
p(x).sum(x,1,oo)                   # Sum the pmf
(x*p(x)).sum(x,1,oo)               # Expected value
((x-1/theta)^2*p(x)).sum(x,1,oo)  # Variance
```

evaluate

$-\frac{\theta - 1}{\theta^2}$

In the next example, the parameter λ of the Poisson distribution must be treated specially because it has a specific advanced programming meaning and the word is reserved. It can still be used as a symbol if it is assigned to a variable *and* used with an underscore as illustrated. Lambdas with subscripts present no problems. In fact, `lambda_` can be viewed as a λ with an invisible subscript.

```
# Poisson - lambda has a special meaning. But if you assign
# it to a variable and define it WITH AN UNDERSCORE you can
# still use it as a symbol.
L = var('lambda_')
p(x) = exp(-L) * L^x / factorial(x) ; p(x)
```

evaluate

$\frac{\lambda^x e^{(-\lambda)}}{x!}$

```
p(x).sum(x,0,oo)        # Sums nicely to one
(x*p(x)).sum(x,0,oo)    # Expected value
```

evaluate

λ

Here is some sample code for the Gamma distribution. Note the use of `full_simplify` on ratios of gamma functions.

```
# Gamma
var('alpha beta')
assume(alpha>0); assume(beta>0)
assume(alpha,'noninteger'); assume(beta,'noninteger')
f(x) = 1/(beta^alpha*gamma(alpha)) * exp(-x/beta) * x^(alpha-1)
integrate(f(x),x,0,oo) # Equals one
integrate(x*f(x),x,0,oo) # E(X)
```

evaluate

$\frac{\beta\Gamma(\alpha+1)}{\Gamma(\alpha)}$

```
_.full_simplify() # Underscore refers to the preceding expression.
```

evaluate

$\alpha\beta$

Now for the the moment-generating function. When I first tried it `Sage` asked "`Is beta*t-1 positive, negative, or zero?`" Because the moment-generating function only needs be defined in a neighbourhood of zero. I said `assume(beta*t<1)`, which is equivalent to $t < \frac{1}{\beta}$. In this way, `Sage` makes us specify the *radius of convergence* of the moment-generating function, but only when the radius of convergence is not the whole real line. `Sage` may be just a calculator, but it's a very smart calculator. It helps keep us mathematically honest. You have to love it.

```
# Moment-generating function
var('t'); assume(beta*t<1)
M(t) = integrate(exp(x*t)*f(x),x,0,oo).full_simplify(); M(t)
derivative(M(t),t,2)(t=0).full_simplify() # Lovely
```

evaluate

$(\alpha^2+\alpha)\beta^2$

Here is some sample code for the Binomial distribution. Only the input is given.

```
# Binomial
var('n theta')
assume(n,'integer'); assume(n>-1)
assume(0<theta); assume(theta<1)
p(x) = factorial(n)/(factorial(x)*factorial(n-x)) * theta^x * (1-theta)^(n-x)
p(x).sum(x,0,n)                                  # Adds to one
(x*p(x)).sum(x,0,n).full_simplify()              # E(X)
(x^2*p(x)).sum(x,0,n).full_simplify()            # E(X^2)
((x-n*theta)^2*p(x)).sum(x,0,n).full_simplify() # cov(X) directly
```

Maxima and Minima in Several Variables (Maximum Likelihood)

The standard way to derive maximum likelihood estimators is to partially differentiate the log likelihood with respect to each parameter, set the resulting expressions to zero, and solve for the parameters. This task is routine with Sage, except for one part. The "one part" is actually a nasty clerical chore that a symbolic math program like Sage *should* be able to do for us. Writing the likelihood function as

$$L(\theta) = \prod_{i=1}^{n} f(x_i|\theta),$$

the task is to carry out the multiplication, using the fact that multiplication is addition of exponents. The result is often an expression in the parameter θ and a a set of (sufficient) *statistics* – that is, functions of the sample data that could be calculated without knowing any of the parameters. I'm not insisting this step cannot be done with Sage, only that I've tried hard, I can't do it with Mathematica either, and other knowledgeable users[4] can't seem to make Sage do it either.

The Univariate Normal Distribution For the normal distribution, one version of the calculation goes like this.

$$\begin{aligned}
L(\mu,\sigma) &= \prod_{i=1}^{n}\left(= \frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(x_i-\mu)^2}{2\sigma^2}}\right) \\
&= \frac{1}{\sigma^n(2\pi)^{n/2}}e^{-\frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i-\mu)^2} \\
&= \frac{1}{\sigma^n(2\pi)^{n/2}}e^{-\frac{1}{2\sigma^2}\sum_{i=1}^{n}\left(x_i^2-2x_i\mu+\mu^2\right)} \\
&= \frac{1}{\sigma^n(2\pi)^{n/2}}e^{-\frac{1}{2\sigma^2}\left(\sum_{i=1}^{n}x_i^2-2\mu\sum_{i=1}^{n}x_i+n\mu^2\right)}
\end{aligned}$$

[4]Somebody is a statistician in New Zealand who uses Sage in her classes. I have not asked her directly, but in the material she posts online she simplifies likelihood functions by hand, just as I am forced to do here.

This is not actually the best way to do the calculation. Better is to add and subtract $\overline{x}$ in the exponent. But this way requires a bit less insight (or experience), and leads to a more complicated problem that illustrates Sage's power. Continuing, the minus log likelihood function is

$$-\ell(\mu,\sigma) = n\log\sigma + \frac{n}{2}\log 2\pi + \frac{1}{2\sigma^2}\left(\left(\sum_{i=1}^n x_i^2\right) - 2\mu\left(\sum_{i=1}^n x_i\right) + n\mu^2\right).$$

Notice how the likelihood has been simplified to an expression that depends on the sample data only through a two-dimensional sufficient statistic[5]. This is what we need to minimize over the pair (μ,σ). In the Sage code, $\sum_{i=1}^n x_i$ will be denoted by s_1 and $\sum_{i=1}^n x_i^2$ will be denoted by s_2.

```
# Minus Log Likelihood for univariate normal
# s1 is sum of x, s2 is sum of x^2
var('mu sigma s1 s2 n')
mLL = n*log(sigma) + n/2 * log(2*pi) + 1/(2*sigma^2) * (s2 - 2*mu*s1 + n*mu^2)
mLL
```

evaluate

$$\frac{1}{2}\,n\log\left(2\,\pi\right) + n\log\left(\sigma\right) + \frac{\mu^2 n - 2\,\mu s_1 + s_2}{2\,\sigma^2}$$

Now partially differentiate the minus log likelihood with respect to μ and σ, set the derivates to zero, and solve.

```
d1 = derivative(mLL,mu); d2 = derivative(mLL,sigma)
eq = [d1==0,d2==0]; eq
```

evaluate

$$\left[\frac{\mu n - s_1}{\sigma^2} = 0, \frac{n}{\sigma} - \frac{\mu^2 n - 2\,\mu s_1 + s_2}{\sigma^3} = 0\right]$$

```
# Solution is a list of lists
sol1 = solve(eq,[mu,sigma]); sol1
```

evaluate

$$\left[\left[\mu = \frac{s_1}{n}, \sigma = -\frac{\sqrt{n s_2 - s_1^2}}{n}\right], \left[\mu = \frac{s_1}{n}, \sigma = \frac{\sqrt{n s_2 - s_1^2}}{n}\right]\right]$$

Notice that there is only one solution for μ; it's $\mu = \frac{s_1}{n} = \overline{x}$. But there are two solutions for σ; they simplify to plus and minus the sample standard deviation (with n rather than $n-1$ in the denominator).

Of course we discard the negative solution because it's outside the parameter space, but this illustrates a feature of Sage that can be easy to forget. It doesn't know as much

[5]The fact that the sufficient statistic has the same dimension as the parameter suggests that we will live happily ever after.

about the problem as you do. Not only does it not know that variances can't be negative, it does not know that the quantity under the square root sign has to be positive, or even that all the symbols represent real numbers rather than complex numbers. I tried playing around with `assume`, but to no avail. There were always two solutions. It's easy enough to get the one we want. It's element one in the list of lists – the second one.

```
# Extract the second list of solutions
sol1[1]
```

evaluate

$$\left[\mu = \frac{s_1}{n}, \sigma = \frac{\sqrt{ns_2 - s_1^2}}{n}\right]$$

Later, it will be handy to evaluate the parameter vector at the vector of MLEs. So, this time, get the solution in the form of a dictionary (exactly like a Python dictionary). Actually, `solve` returns a *list* of dictionaries, and we want the second one.

```
# This time, get the solutions in the form of a LIST of dictionaries.
#  Save item one, the second one. (Indices begin with zero, not one.)
mle = solve(eq,[mu,sigma],solution_dict=True)[1]; mle
```

evaluate

$$\left\{\sigma : \frac{\sqrt{ns_2 - s_1^2}}{n}, \mu : \frac{s_1}{n}\right\}$$

```
# Refer to the elements of a dictionary using the keys.
mle[mu]  # MLE of mu
```

evaluate

$$\frac{s_1}{n}$$

For this particular case, it's not hard to show by elementary methods that the likelihood function attains its maximum at the sample mean and standard deviation, rather than a minimum or saddle point. But the general method is of interest. For a function $g(\theta_1, \ldots, \theta_t)$, define the *Hessian* as the $t \times t$ matrix of mixed partial derivatives whose i, j element is

$$\frac{\partial^2 g}{\partial\theta_i \partial\theta_j}. \tag{B.1}$$

If the eigenvalues of the Hessian are all positive at a critical point, the function is concave up there. If they are all negative, it's concave down. If some are positive and some are negative, it's a saddle point.

In `Sage`, functions have a built-in Hessian attribute, but unfortunately, it applies to *all* symbolic variables. So `mLL.hessian()` returns a 5×5 matrix, corresponding to $(\mu, n, s_1, s_2, \sigma)$, in alphabetical order. And `mLL.hessian([mu,sigma])` (which is natural, and similar to expressions that work with gradients and Jacobians) yields `TypeError:`

`hessian() takes no arguments (1 given)`. So we'll construct the Hessian from scratch. Start by making an empty matrix that will be filled with partial derivates. It's critical that the matrix be of the right *type* (symbolic). Also, note that a lot of burdensome High School algebra is avoided by the quiet use of `factor` in the calculations below.

```
# H will be hessian of MINUS log likelihood
H = identity_matrix(SR,2); H # SR is the Symbolic Ring
```

evaluate

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

```
# Fill it with mixed partial derivatives
H[0,0] = derivative(mLL,mu,2); H[0,1] = derivative(mLL,[mu,sigma])
H[1,0] = H[0,1]                ; H[1,1] = derivative(mLL,sigma,2)
H = factor(H); H
```

evaluate

$$\begin{pmatrix} \frac{n}{\sigma^2} & -2\,\frac{(\mu n - s_1)}{\sigma^3} \\ -2\,\frac{(\mu n - s_1)}{\sigma^3} & \frac{(3\,\mu^2 n - n\sigma^2 - 6\,\mu s_1 + 3\,s_2)}{\sigma^4} \end{pmatrix}$$

```
# Evaluate at mle
hmle = factor(H(mle)); hmle
```

evaluate

$$\begin{pmatrix} \frac{n^3}{(n s_2 - s_1^2)} & 0 \\ 0 & 2\,\frac{n^3}{(n s_2 - s_1^2)} \end{pmatrix}$$

```
# Function is concave up at critical point iff all eigenvalues > 0 there.
hmle.eigenvalues()
```

evaluate

$$\left[\frac{n^3}{(n s_2 - s_1^2)}, 2\,\frac{n^3}{(n s_2 - s_1^2)}\right]$$

The denominator of both eigenvalues equals

$$n\sum_{i=1}^n x_i^2 - \left(\sum_{i=1}^n x_i\right)^2 = n\sum_{i=1}^n (x_i - \overline{x})^2,$$

so both eigenvalues are positive and the minus log likelihood is concave up at the MLE.

The Multinomial Distribution The multinomial distribution is based on a statistical experiment in which one of k outcomes occurs, with probability $\theta_j, j = 1, \ldots, k$, where

$\sum_{j=1}^k \theta_j = 1$. For example, consumers might be asked to smell six perfumes, and indicate which one they like most. The probability of preferring perfume j is θ_j, for $j = 1, \ldots, 6$.

The likelihood function may be written in terms of multinomial random vectors made up of k indicators random variables: For case i, $x_{ij} = 1$ if event j occurs, and zero otherwise. $\sum_{j=1}^k x_{ij} = 1$. The likelihood function is

$$\begin{aligned} L(\boldsymbol{\theta}) &= \prod_{i=1}^n \theta_1^{x_{i,1}} \theta_2^{x_{i,2}} \cdots \theta_k^{x_{i,k}} \\ &= \theta_1^{\sum_{i=1}^n x_{i,1}} \theta_2^{\sum_{i=1}^n x_{i,2}} \cdots \theta_k^{\sum_{i=1}^n x_{i,k}}. \end{aligned}$$

Using x_j to represent the sum $\sum_{i=1}^n x_{i,j}$, the likelihood may be expressed in a non-redundant way in terms of $k-1$ parameters and $k-1$ sufficient statistics, as follows:

$$\begin{aligned} L(\boldsymbol{\theta}) &= \theta_1^{x_1} \theta_2^{x_2} \cdots \theta_k^{x_k} \\ &= \theta_1^{x_1} \cdots \theta_{k-1}^{x_{k-1}} \left(1 - \sum_{j=1}^{k-1} \theta_j \right)^{n - \sum_{j=1}^{k-1} x_j}. \end{aligned}$$

Here's an example with $k = 6$ (six perfumes).

```
# Multinomial Maximum likelihood - 6 categories
var('theta1 theta2 theta3 theta4 theta5  x1 x2 x3 x4 x5 n')
theta = [theta1, theta2, theta3, theta4, theta5]
LL = x1*log(theta1) + x2*log(theta2) + x3*log(theta3) +
x4*log(theta4)  + x5*log(theta5) +
(n-x1-x2-x3-x4-x5)*log(1-theta1-theta2-theta3-theta4-theta5)
LL
```

evaluate

$(n - x_1 - x_2 - x_3 - x_4 - x_5) \log(-\theta_1 - \theta_2 - \theta_3 - \theta_4 - \theta_5 + 1) + x_1 \log(\theta_1) + x_2 \log(\theta_2) + x_3 \log(\theta_3) + x_4 \log(\theta_4) + x_5 \log(\theta_5)$

Instead of calculating all five partial derivatives, it's easier to request the gradient – which is the same thing. Then we loop through the element of the gradient list, setting each derivative to zero, displaying the equation, and appending it to a list of equations that need to be solved. Notice the use of the colon (:) and indentation for looping. Sage shares this syntax with Python.

```
# Gradient is zero at MLE. It's a tuple, not a list.
gr = LL.gradient(theta)
# Setting the derivatives to zero ...
eq = [] # Start with empty list
for a in gr :
    equation = (a==0)
    show(equation)  # Display the equation
    eq.append(equation) # Append equation to list eq.
```

evaluate

$$\frac{n-x_1-x_2-x_3-x_4-x_5}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1} + \frac{x_1}{\theta_1} = 0$$

$$\frac{n-x_1-x_2-x_3-x_4-x_5}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1} + \frac{x_2}{\theta_2} = 0$$

$$\frac{n-x_1-x_2-x_3-x_4-x_5}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1} + \frac{x_3}{\theta_3} = 0$$

$$\frac{n-x_1-x_2-x_3-x_4-x_5}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1} + \frac{x_4}{\theta_4} = 0$$

$$\frac{n-x_1-x_2-x_3-x_4-x_5}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1} + \frac{x_5}{\theta_5} = 0$$

Now we will solve for $\theta_1, \ldots, \theta_5$. While it's true that the Sage calculation is specific to $k = 6$ categories, the list of equations to solve makes the pattern clear, and points the way to a general solution. Here is the specific solution:

```
# Get the solutions in the form of a LIST of dictionaries.
# Dictionary items are not in any particular order.
# Save item zero, the first dictionary.
ThetaHat = solve(eq,theta,solution_dict=True)[0]
ThetaHat  # The mean (vector)
```

evaluate

$$\left\{\theta_3 : \frac{x_3}{n}, \theta_2 : \frac{x_2}{n}, \theta_1 : \frac{x_1}{n}, \theta_5 : \frac{x_5}{n}, \theta_4 : \frac{x_4}{n}\right\}$$

So for $j = 1, \ldots, 5$, the MLE is $\widehat{\theta}_j = \frac{\sum_{i=1}^n x_{ij}}{n} = \overline{x}_j$, or the sample proportion. There's little doubt that this is really where the likelihood function achieves its maximum, and not a minimum or saddle point. But it's instructive to check. Here is the Hessian of the minus log likelihood.

```
# Is it really the maximum?
 # H will be hessian of MINUS log likelihood
 H = identity_matrix(SR,5) # SR is the Symbolic Ring
for i in interval(0,4) :
    for j in interval(0,i) :
        H[i,j] = derivative(-LL,[theta[i],theta[j]])
        H[j,i] = H[i,j] # It's symmetric
H
```

evaluate

$$\begin{pmatrix}
\frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} + \frac{x_1}{\theta_1^2} & \frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & \frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & \frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & \frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} \\
\frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & \frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} + \frac{x_2}{\theta_2^2} & \frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & \frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & \frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} \\
\frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & \frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & \frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} + \frac{x_3}{\theta_3^2} & \frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & \frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} \\
\frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & \frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & \frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & \frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} + \frac{x_4}{\theta_4^2} & \frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} \\
\frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & \frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & \frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & \frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & \frac{n-x_1-x_2-x_3-x_4-x_5}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} + \frac{x_5}{\theta_5^2}
\end{pmatrix}$$

All its eigenvalues should be positive at the critical point where the derivates simultaneously equal zero.

```
# Evaluate at critical point
Hmle = factor(H(ThetaHat)); Hmle
```

evaluate

$$\left(\begin{array}{ccccc}
\frac{(n-x_2-x_3-x_4-x_5)n^2}{(n-x_1-x_2-x_3-x_4-x_5)x_1} & \frac{n^2}{n-x_1-x_2-x_3-x_4-x_5} & \frac{n^2}{n-x_1-x_2-x_3-x_4-x_5} & \frac{n^2}{n-x_1-x_2-x_3-x_4-x_5} & \frac{n^2}{n-x_1-x_2-x_3-x_4-x_5} \\
\frac{n^2}{n-x_1-x_2-x_3-x_4-x_5} & \frac{(n-x_1-x_3-x_4-x_5)n^2}{(n-x_1-x_2-x_3-x_4-x_5)x_2} & \frac{n^2}{n-x_1-x_2-x_3-x_4-x_5} & \frac{n^2}{n-x_1-x_2-x_3-x_4-x_5} & \frac{n^2}{n-x_1-x_2-x_3-x_4-x_5} \\
\frac{n^2}{n-x_1-x_2-x_3-x_4-x_5} & \frac{n^2}{n-x_1-x_2-x_3-x_4-x_5} & \frac{(n-x_1-x_2-x_4-x_5)n^2}{(n-x_1-x_2-x_3-x_4-x_5)x_3} & \frac{n^2}{n-x_1-x_2-x_3-x_4-x_5} & \frac{n^2}{n-x_1-x_2-x_3-x_4-x_5} \\
\frac{n^2}{n-x_1-x_2-x_3-x_4-x_5} & \frac{n^2}{n-x_1-x_2-x_3-x_4-x_5} & \frac{n^2}{n-x_1-x_2-x_3-x_4-x_5} & \frac{(n-x_1-x_2-x_3-x_5)n^2}{(n-x_1-x_2-x_3-x_4-x_5)x_4} & \frac{n^2}{n-x_1-x_2-x_3-x_4-x_5} \\
\frac{n^2}{n-x_1-x_2-x_3-x_4-x_5} & \frac{n^2}{n-x_1-x_2-x_3-x_4-x_5} & \frac{n^2}{n-x_1-x_2-x_3-x_4-x_5} & \frac{n^2}{n-x_1-x_2-x_3-x_4-x_5} & \frac{(n-x_1-x_2-x_3-x_4)n^2}{(n-x_1-x_2-x_3-x_4-x_5)x_5}
\end{array}\right)$$

```
# Concave up iff all eigenvalues > 0
Hmle.eigenvalues()
```

evaluate

```
Traceback (click to the left of this block for traceback)
...
ArithmeticError: could not determine eigenvalues exactly using symbolic
matrices; try using a different type of matrix via self.change_ring(),
if possible
```

It seems that Sage cannot solve for the eigenvalues symbolically. A *numerical* solution for a particular set of sample data would be routine. But there is another way out. A real symmetric matrix has all positive eigenvalues if and only if it's positive definite. And *Sylvester's Criterion*[6] is a necessary and sufficient condition for a real symmetric matrix to be positive definite. A *minor* of a matrix is the determinant of a square sub-matrix that is formed by deleting selected rows and columns from the original matrix. The *principal minors* of a square matrix are the determinants of the upper left 1×1 matrix, the upper left 2×2 matrix, and so on. Sylvester's Criterion says that the matrix is positive definite if and only if all the principal minors are positive.

Here, there are five determinants to evaluate, one of which is just the upper left matrix element. We'll do it in a loop. The `submatrix(h,i,j,k)` attribute returns the submatrix starting in row h and column i, consisting of j rows and k columns. As usual, index numbering starts with zero. For full documentation, try something like `Hmle.submatrix?`

```
Hmle.submatrix(0,0,2,2) # Upper left 2x2, just to see
```

evaluate

[6]The Wikipedia has a nice article on this, including a formal proof. See http://www.en.wikipedia.org/.

$$\begin{pmatrix} \frac{(n-x_2-x_3-x_4-x_5)n^2}{(n-x_1-x_2-x_3-x_4-x_5)x_1} & \frac{n^2}{n-x_1-x_2-x_3-x_4-x_5} \\ \frac{n^2}{n-x_1-x_2-x_3-x_4-x_5} & \frac{(n-x_1-x_3-x_4-x_5)n^2}{(n-x_1-x_2-x_3-x_4-x_5)x_2} \end{pmatrix}$$

```
# Calculate and display determinants
for j in interval(1,5) :
    show(Hmle.submatrix(0,0,j,j).determinant().factor())
```

evaluate

$$\frac{(n-x_2-x_3-x_4-x_5)n^2}{(n-x_1-x_2-x_3-x_4-x_5)x_1}$$

$$\frac{(n-x_3-x_4-x_5)n^4}{(n-x_1-x_2-x_3-x_4-x_5)x_1x_2}$$

$$\frac{(n-x_4-x_5)n^6}{(n-x_1-x_2-x_3-x_4-x_5)x_1x_2x_3}$$

$$\frac{(n-x_5)n^8}{(n-x_1-x_2-x_3-x_4-x_5)x_1x_2x_3x_4}$$

$$\frac{n^{11}}{(n-x_1-x_2-x_3-x_4-x_5)x_1x_2x_3x_4x_5}$$

Assuming the sample size is large enough so that there's at least one observation in each category, these quantities are obviously all positive. You can also see that while Sage performs calculations that are very specific to the problem at hand, the answers can reveal regular patters that could be exploited in something like a proof by induction. And the effort involved is tiny, compared to doing it by hand.

Incidentally, the `submatrix` function can be used to obtain Hessians a bit more easily. Recall that Sage functions have a `hessian` attribute, but it's calculated with respect to *all* the variables, which is never what you want for likelihood calculations. But the rows and columns are in alphabetical order, which in the present case is $n, \theta_1, \ldots, \theta_5, x_1, \ldots, x_5$. So the 5×5 Hessian we want is easy to extract. Check and see if it's what we calculated earlier in a double loop.

```
-LL.hessian().submatrix(1,1,5,5) == H
```

evaluate

```
True
```

Ho Ho!

Fisher Information

There are many places in mathematical Statistics where Sage can save a lot of tedious calculation. One of these is in conjunction with *Fisher Information* (See Appendix A for some discussion). For a model with parameter vector $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_t)'$, the Fisher information matrix is a $t \times t$ matrix $I(\boldsymbol{\theta})$ whose (i, j) element is

$$-E\left(\frac{\partial^2}{\partial\theta_i\partial\theta_j}\log f(X|\boldsymbol{\theta})\right).$$

This is the information about $\boldsymbol{\theta}$ in a single observation. The information in n independent and identically distributed observations is $n\,I(\boldsymbol{\theta})$. Under some regularity conditions that amount to smoothness of the functions involved, the vector of MLEs is approximately multivariate normal for large samples, with mean $\boldsymbol{\theta}$ and covariance matrix $(n\,I(\boldsymbol{\theta}))^{-1}$. This is a source of large-sample tests and confidence intervals.

The Univariate Normal Distribution Comparing the formula for the Fisher Information to Expression (B.1), it is clear that the Fisher information is just the expected value of the Hessian of the minus log density[7]. We'll start by calculating the Hessian. The last line says "Take minus the log of $f(X)$, calculate the Hessian, extract the 2×2 matrix with upper left entry $(1, 1)$, and factor it. Then put the result in `h`; display `h`." In this case and many others, factoring yields a lot of simplification.

```
# Normal
var('mu, sigma, X, n'); assume(sigma>0)
f(x) = 1/(sigma*sqrt(2*pi)) * exp(-(x-mu)^2/(2*sigma^2))
# Extract lower right 2x2 of Hessian of minus log density
# That is, of Hessian with respect to X, mu, sigma.
# X is alphabetically first because it's capitalized.
h = -log(f(X)).hessian().submatrix(1,1,2,2).factor(); h
```

evaluate

$$\left(\begin{array}{cc} \frac{1}{\sigma^2} & \frac{2\,(X-\mu)}{\sigma^3} \\ \frac{2\,(X-\mu)}{\sigma^3} & \frac{3\,X^2-6\,X\mu+3\,\mu^2-\sigma^2}{\sigma^4} \end{array}\right)$$

Now take the expected value. In the lower right we'll directly integrate, though it could also be done by substituting in known quantities and then simplifying. The other cells can be done by inspection.

```
# Fisher information in one observation is expected h
info = h
info[0,1]=0; info[1,0]=0 # Because E(X)=mu
info[1,1] = integrate(info[1,1]*f(X),X,-oo,oo)
info
```

evaluate

[7]The Hessian reflects *curvature* of the function. Fisher's insight was that the greater the curvature of the log likelihood function at the true parameter value, the more information the data provide about the parameter. Further discussion of the connection between the Hessian and the Fisher Information may be found in Appendix A.

$$\begin{pmatrix} \frac{1}{\sigma^2} & 0 \\ 0 & \frac{2}{\sigma^2} \end{pmatrix}$$

That's the Fisher Information in one observation. To get the asymptotic (approximate, for large n) covariance matrix, multiply by n and invert the matrix.

```
# Fisher info in n observations is n * info in one observation.
# MLEs are asymptotically multivariate normal with mean theta
# and variance-covariance matrix the inverse of the Fisher info.
avar = (n*info).inverse(); avar
```

evaluate

$$\begin{pmatrix} \frac{\sigma^2}{n} & 0 \\ 0 & \frac{\sigma^2}{2\,n} \end{pmatrix}$$

That's a standard example that can be done by hand, though perhaps it's a little unusual because the model is parameterized in terms of the standard deviation rather than the variance. This next one, however, would be fearsome to do by hand.

The Multinomial Distribution We'll stay with the case of six categories. Now, because the MLE equals the sample mean vector in this case, the multivariate Central Limit Theorem (see Appendix A) can be used directly without going through the Fisher Information. We'll do it this way first, because it's a good way to check Sage's final answer.

The multivariate Central Limit Theorem says that if $\mathbf{X}_1, \ldots, \mathbf{X}_n$ are i.i.d. random vectors with expected value vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$. Then $\sqrt{n}(\overline{\mathbf{X}}_n - \boldsymbol{\mu})$ converges in distribution to a multivariate normal with mean $\mathbf{0}$ and covariance matrix $\boldsymbol{\Sigma}$. That is, for large n, $\overline{\mathbf{X}}_n$ has a distribution that is approximately multivariate normal, with mean $\boldsymbol{\mu}$ and covariance matrix $\frac{1}{n}\boldsymbol{\Sigma}$.

Here, each of the i.i.d. random vectors is filled with $k-1=5$ zeros and possibly a single 1 , with the number 1 indicating which event occurred. If all five entries of $\mathbf{X}_i$ equal zero, then the sixth event occurred. The marginal distributions are Bernoulli, so $E(X_{i,j}) = \theta_j$ and $\boldsymbol{\mu} = (\theta_1, \ldots, \theta_5)'$. The variances are $Var(X_{i,j}) = \theta_j(1-\theta_j)$, for $j = 1, \ldots, 5$. Since, $Pr\{X_{i,j}X_{i,m} = 0\}$ for $j \neq m$, $E(X_{i,j}X_{i,m}) = 0$, and

$$\begin{aligned} Cov(X_{i,j}X_{i,m}) &= E(X_{i,j}X_{i,m}) - E(X_{i,j})E(X_{i,m}) \\ &= -\theta_j\theta_m. \end{aligned}$$

So by the Central Limit Theorem, the asymptotic mean of the MLE is $\boldsymbol{\mu} = (\theta_1, \ldots, \theta_5)'$, and the asymptotic covariance matrix is

$$\frac{1}{n}\boldsymbol{\Sigma} = \begin{pmatrix} \frac{\theta_1(1-\theta_1)}{n} & -\frac{\theta_1\theta_2}{n} & -\frac{\theta_1\theta_3}{n} & -\frac{\theta_1\theta_4}{n} & -\frac{\theta_1\theta_5}{n} \\ -\frac{\theta_1\theta_2}{n} & \frac{\theta_2(1-\theta_2)}{n} & -\frac{\theta_2\theta_3}{n} & -\frac{\theta_2\theta_4}{n} & -\frac{\theta_2\theta_5}{n} \\ -\frac{\theta_1\theta_3}{n} & -\frac{\theta_2\theta_3}{n} & \frac{\theta_3(1-\theta_3)}{n} & -\frac{\theta_3\theta_4}{n} & -\frac{\theta_3\theta_5}{n} \\ -\frac{\theta_1\theta_4}{n} & -\frac{\theta_2\theta_4}{n} & -\frac{\theta_3\theta_4}{n} & \frac{\theta_4(1-\theta_4)}{n} & -\frac{\theta_4\theta_5}{n} \\ -\frac{\theta_1\theta_5}{n} & -\frac{\theta_2\theta_5}{n} & -\frac{\theta_3\theta_5}{n} & -\frac{\theta_4\theta_5}{n} & \frac{\theta_5(1-\theta_5)}{n} \end{pmatrix} \qquad (B.2)$$

To compare this to what we get from the likelihood approach, first calculate the Hessian of the minus log probability mass function.

```
# Multinomial - 6 categories again
var('theta1 theta2 theta3 theta4 theta5  X1 X2 X3 X4 X5  n')
Lp = X1*log(theta1) + X2*log(theta2) + X3*log(theta3)
+ X4*log(theta4) + X5*log(theta5) + (1-X1-X2-X3-X4-X5)
* log(1-theta1-theta2-theta3-theta4-theta5)
h = -Lp.hessian().submatrix(5,5,5,5); h
```

evaluate

$$\left(\begin{array}{llll}
-\frac{X_1+X_2+X_3+X_4+X_5-1}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2}+\frac{X_1}{\theta_1^2} & -\frac{X_1+X_2+X_3+X_4+X_5-1}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & -\frac{X_1+X_2+X_3+X_4+X_5-1}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & -\frac{X_1+X_2+X}{(\theta_1+\theta_2+\theta_3} \\
-\frac{X_1+X_2+X_3+X_4+X_5-1}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & -\frac{X_1+X_2+X_3+X_4+X_5-1}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2}+\frac{X_2}{\theta_2^2} & -\frac{X_1+X_2+X_3+X_4+X_5-1}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & -\frac{X_1+X_2+X}{(\theta_1+\theta_2+\theta_3} \\
-\frac{X_1+X_2+X_3+X_4+X_5-1}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & -\frac{X_1+X_2+X_3+X_4+X_5-1}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & -\frac{X_1+X_2+X_3+X_4+X_5-1}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2}+\frac{X_3}{\theta_3^2} & -\frac{X_1+X_2+X}{(\theta_1+\theta_2+\theta_3} \\
-\frac{X_1+X_2+X_3+X_4+X_5-1}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & -\frac{X_1+X_2+X_3+X_4+X_5-1}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & -\frac{X_1+X_2+X_3+X_4+X_5-1}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & -\frac{X_1+X_2+X_3+X_4+}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta} \\
-\frac{X_1+X_2+X_3+X_4+X_5-1}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & -\frac{X_1+X_2+X_3+X_4+X_5-1}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & -\frac{X_1+X_2+X_3+X_4+X_5-1}{(\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1)^2} & -\frac{X_1+X_2+X}{(\theta_1+\theta_2+\theta_3}
\end{array}\right.$$

Sometimes, Sage output runs off the right side of the screen and you have to scroll to see it all. In this document, it just gets chopped off. But you can still see that all the X_j quantities appear in the numerator, and taking the expected values would be easy by hand.

```
# Computing expected values is just substituting theta_j for X_j
info = h(X1=theta1,X2=theta2,X3=theta3,X4=theta4,X5=theta5)
info
```

evaluate

$$\left(\begin{array}{lllll}
-\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1}+\frac{1}{\theta_1} & -\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1} & -\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1} & -\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1} & -\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+} \\
-\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1} & -\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1}+\frac{1}{\theta_2} & -\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1} & -\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1} & -\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+} \\
-\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1} & -\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1} & -\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1}+\frac{1}{\theta_3} & -\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1} & -\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+} \\
-\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1} & -\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1} & -\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1} & -\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1}+\frac{1}{\theta_4} & -\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+} \\
-\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1} & -\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1} & -\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1} & -\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1} & -\frac{1}{\theta_1+\theta_2+\theta_3+\theta_4+\theta_5-1}
\end{array}\right.$$

The asymptotic covariance matrix is obtained by multiplying by n and taking the inverse. Inverting the matrix by hand is possible, but it would be a brutal experience. With Sage, it takes a few seconds, including the typing.

```
# Asymptotic covariance matrix
avar = (n*info).inverse().factor(); avar
```

evaluate

$$\left(\begin{array}{ccccc}
-\frac{(\theta_1-1)\theta_1}{n} & -\frac{\theta_1\theta_2}{n} & -\frac{\theta_1\theta_3}{n} & -\frac{\theta_1\theta_4}{n} & -\frac{\theta_1\theta_5}{n} \\
-\frac{\theta_1\theta_2}{n} & -\frac{(\theta_2-1)\theta_2}{n} & -\frac{\theta_2\theta_3}{n} & -\frac{\theta_2\theta_4}{n} & -\frac{\theta_2\theta_5}{n} \\
-\frac{\theta_1\theta_3}{n} & -\frac{\theta_2\theta_3}{n} & -\frac{(\theta_3-1)\theta_3}{n} & -\frac{\theta_3\theta_4}{n} & -\frac{\theta_3\theta_5}{n} \\
-\frac{\theta_1\theta_4}{n} & -\frac{\theta_2\theta_4}{n} & -\frac{\theta_3\theta_4}{n} & -\frac{(\theta_4-1)\theta_4}{n} & -\frac{\theta_4\theta_5}{n} \\
-\frac{\theta_1\theta_5}{n} & -\frac{\theta_2\theta_5}{n} & -\frac{\theta_3\theta_5}{n} & -\frac{\theta_4\theta_5}{n} & -\frac{(\theta_5-1)\theta_5}{n}
\end{array}\right)$$

This is the same as Expression B.2, which came from the Central Limit Theorem. It's an unqualified success.

Taylor Expansions

There are many versions of Taylor's Theorem. Here is a useful one. Let the nth derivative $f^{(n)}$ of the function $f(x)$ be continuous in $[a,b]$ and differentiable in (a,b), with x and x_0 in (a,b). Then there exists a point ξ between x and x_0 such that

$$\begin{aligned} f(x) \;&=\; f(x_0) \;+\; f'(x_0)\,(x-x_0) \;+\; \frac{f''(x_0)(x-x_0)^2}{2!} \;+\; \ldots \;+\; \frac{f^{(n)}(x_0)(x-x_0)^n}{n!} \\ &+\; \frac{f^{(n+1)}(\xi)(x-x_0)^{n+1}}{(n+1)!} \end{aligned} \tag{B.3}$$

where $R_n = \frac{f^{(n+1)}(\xi)(x-x_0)^{n+1}}{(n+1)!}$ is called the *remainder term*. If $R_n \to 0$ as $n \to \infty$, the resulting infinite series is called the *Taylor Series* for $f(x)$.

In certain styles of applied statistics, when people are having trouble with a function, they approximate it by just taking the first two or three terms of a Taylor expansion, and discarding the remainder. Sometimes, the approximation can be quite helpful. Consider, for example, a simple[8] logistic regression in which a linear model for the log odds of $Y=1$ leads to

$$Pr\{Y=1|X=x\} = E(Y|X=x) = \frac{e^{\beta_0+\beta_1 x}}{1+e^{\beta_0+\beta_1 x}}.$$

Under this model, what is the covariance between X and Y? It's easy to wonder, but not easy to calculate. Suppose X has a distribution with expected value μ and variance σ^2. Perhaps X is normal. Let's use the formula $Cov(X,Y) = E(XY) - E(X)E(Y)$, and try double expectation. That is,

$$\begin{aligned} E[Y] &= E[E(Y|X)] \\ &= \int_{-\infty}^{\infty} E(Y|X=x)\,f(x)\,dx \\ &= \int_{-\infty}^{\infty} \frac{e^{\beta_0+\beta_1 x}}{1+e^{\beta_0+\beta_1 x}}\,f(x)\,dx. \end{aligned} \tag{B.4}$$

If X is normal, I certainly can't do this integral. I have tried many times and failed. `Sage` can't do it either. Details are omitted.

[8]One explanatory variable.

Let's approximate $g(X) = E(Y|X)$ with the first few terms of a Taylor series. Then it's easier to work with. Note that you can find out what atributes the function g has with `print(dir(g))`, and then get details about the `taylor` attribute with `g.taylor?` .

```
# Cov(X,Y) for logistic regression (Taylor)

var('X beta0 beta1 mu sigma')
g = exp(beta0 + beta1*X)/(1+exp(beta0 + beta1*X))
# print(dir(g))
# g.taylor?
t1 = g.taylor(X,mu,2); t1 # Expand function of X about mu, degree 2 (3 terms)
```

evaluate

$$\frac{(X-\mu)\beta_1 e^{(\beta_1\mu+\beta_0)}}{2\,e^{(\beta_1\mu+\beta_0)}+e^{(2\,\beta_1\mu+2\,\beta_0)}+1} + \frac{(X-\mu)^2\left(\beta_1^2 e^{(\beta_1\mu+\beta_0)}-\beta_1^2 e^{(2\,\beta_1\mu+2\,\beta_0)}\right)}{2\left(3\,e^{(\beta_1\mu+\beta_0)}+3\,e^{(2\,\beta_1\mu+2\,\beta_0)}+e^{(3\,\beta_1\mu+3\,\beta_0)}+1\right)} + \frac{e^{(\beta_1\mu+\beta_0)}}{e^{(\beta_1\mu+\beta_0)}+1}$$

Taking the expected value with respect to X will cause the first term to disappear, and replace $(X-\mu)^2$ with σ^2 in the second term. We'll integrate with respect to the normal distribution, but that's just for convenience. Any distribution with expected value μ and variance σ^2 would yield the same result.

```
# Use normal to take expected value Just a convenience :
f = 1/(sigma*sqrt(2*pi)) * exp(-(X-mu)^2/(2*sigma^2))
assume(sigma>0)
EY = (t1*f).integrate(X,-oo,oo).factor(); EY
```

evaluate

$$-\frac{\left(\beta_1^2\sigma^2 e^{(\beta_1\mu+\beta_0)}-\beta_1^2\sigma^2-4\,e^{(\beta_1\mu+\beta_0)}-2\,e^{(2\,\beta_1\mu+2\,\beta_0)}-2\right)e^{(\beta_1\mu+\beta_0)}}{2\left(e^{(\beta_1\mu+\beta_0)}+1\right)^3}$$

That's pretty messy, but maybe there will be some simplification when we calculate $Cov(X,Y) = E(XY) - E(X)E(Y)$. First we need an approximation of $E(XY)$.

```
# Double expectation for E(XY) - First, approximate XE(Y|X)
t2 = (X*g).taylor(X,mu,2); t2 # Looks pretty hairy
EXY = (t2*f).integrate(X,-oo,oo).factor(); EXY
```

evaluate

$$-\frac{\left(\beta_1^2\mu\sigma^2 e^{(\beta_1\mu+\beta_0)}-\beta_1^2\mu\sigma^2-2\,\beta_1\sigma^2 e^{(\beta_1\mu+\beta_0)}-2\,\beta_1\sigma^2-4\,\mu e^{(\beta_1\mu+\beta_0)}-2\,\mu e^{(2\,\beta_1\mu+2\,\beta_0)}-2\,\mu\right)e^{(\beta_1\mu+\beta_0)}}{2\left(e^{(\beta_1\mu+\beta_0)}+1\right)^3}$$

```
# Finally, approximate the covariance
Cov = (EXY-mu*EY).factor(); Cov
```

evaluate

$$\frac{\beta_1\sigma^2 e^{(\beta_1\mu+\beta_0)}}{\left(e^{(\beta_1\mu+\beta_0)}+1\right)^2}$$

Well, you have to admit that's nice! Some of the intermediate steps were fiercely complicated, but the final result is clean and simple. Sage has saved us a lot of unpleasant work. Furthermore, the result makes sense because the sign of the covariance is the same as the sign of β_1, as it should be.

However, *we really don't know if it's a good approximation or not.* That's right. Taylor expansions are more accurate closer to the point about which you expand the function, and they are more accurate the more terms you take. Beyond that, it's generally unknown, unless you have more information (like perhaps the remainder you've discarded approaches zero as the sample size increases, or something).

So we need to investigate it a bit more, and the easiest thing to do is to try some numerical examples. With specific numbers for the parameters, Sage will be able to calculate $E(Y)$ and $E(XY)$ by numerical integration. First, we'll try $\mu = 0, \sigma = 2, \beta_0 = 0, \beta_1 = 1$. The approximation is

```
# Example 1, with  mu=0,beta0=0,sigma=2,beta1=1
Cov(mu=0,beta0=0,sigma=2,beta1=1)
```

evaluate

$$1$$

The calculation of $Cov(X,Y) = E(XY)$ by double expectation is similar to (B.4).

$$\begin{aligned}
E[XY] &= E[E(XY|X)] \\
&= \int_{-\infty}^{\infty} E(XY|X=x)\, f(x)\, dx \\
&= \int_{-\infty}^{\infty} E(xY|X=x)\, f(x)\, dx \\
&= \int_{-\infty}^{\infty} x\, E(Y|X=x)\, f(x)\, dx \\
&= \int_{-\infty}^{\infty} x\, \frac{e^{\beta_0+\beta_1 x}}{1+e^{\beta_0+\beta_1 x}}\, f(x)\, dx. \qquad \text{(B.5)}
\end{aligned}$$

In the material below, the result of `show(EXY1)` tells us that $E(XY)$, though it's simplified a bit, is an integral that Sage cannot take any farther, even with specific numerical values. Then, `EXY1.n()` says please evaluate it numerically. The numerical evaluation attribute, in the case of an integral, is a sophisticated numerical integration algorithm.

```
# This will be the covariance, since mu=0
EXY1 = (X*g*f)(mu=0,beta0=0,sigma=2,beta1=1).integrate(X,-oo,oo)
show(EXY1)
EXY1.n()
```

evaluate

$$\frac{\sqrt{2}\int_{-\infty}^{+\infty}\frac{Xe^{\left(-\frac{1}{8}X^2+X\right)}}{e^X+1}\,dX}{4\sqrt{\pi}}$$

0.605705509602159

That's not too promising. Is the approximation really this bad? While Sage is extremely accurate compared to almost any human being, mistakes in the input can cause big problems. Typos are the main source of trouble, but misunderstandings are possible too, and the results can be even worse. So, when a result is a bit surprising like this, it's important to cross-check it somehow. Let's try a simulation with R. The idea is to first simulate a large collection of X values from a normal distribution with mean $\mu = 0$ and standard deviation $\sigma = 2$, calculate $Pr\{Y = 1|X_i\}$, using $\beta_0 = 0$ and $\beta_1 = 1$. Finally, generate binary Y values using those probabilities, and calculate the sample covariance. By the Strong Law of Large Numbers, the probability equals one that the sample covariance approaches the true covariance as $n \to \infty$, like an ordinary limit. So with a very large n, we'll get a good approximation of $Cov(X, Y)$. Is it closer to 1, or 0.606? Here is the R calculation, without further comment.

```
> n = 100000; mu=0; beta0=0; sigma=2; beta1=1
> x = rnorm(n,mu,sigma)
> xb = beta0 + beta1*x
> p = exp(xb)/(1+exp(xb))
> y = rbinom(n,1,p)
> var(cbind(x,y))
          x         y
x 3.9687519 0.6039358
y 0.6039358 0.2499991
```

Now we can be confident that the numerical integration (and the double expectation reasoning behind it) produced correct results, and the Taylor series approximation was poor. It can easily get worse. For example, with $\mu = 1, \sigma = 10, \beta_0 = 1, \beta_1 = 1$, the Taylor series approximation of the covariance is 10.499, while the correct answer by numerical integration is 3.851.

The story has a two-part moral. Taylor series approximations are easy with Sage, but whether they are accurate enough to be useful is another matter. This point is sometimes overlooked in applied Statistics. To be clear, this is not a problem with Sage; the problem is with the practice of blindly linearizing everything.

To leave a better taste about Taylor series approximations, let $X_1, \ldots, X_n$ be a random sample from a Bernoulli distribution, with $Pr\{X_i = 1\} = \theta$. A quantity that is useful in categorical data analysis is the *log odds*:

$$\text{Log Odds} = \log\frac{\theta}{1-\theta},$$

where log refers to the natural logarithm.

The best estimator of θ is the sample proportion: $\overline{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$. The log odds is estimated by

$$Y = \log \frac{\overline{X}}{1-\overline{X}}.$$

The variance of $\overline{X}$ is $\frac{\theta(1-\theta)}{n}$, but what is the variance of the estimated log odds Y? As we shall see, it's possible to give an exact answer for any given n, but the expression is very complicated and hard to use in later calculations.

Instead, for any statistic T_n that estimates θ, and any differentiable function $g(t)$ (of which $g(t) = \log \frac{t}{1-t}$ is an example), expand $g(t)$ about θ, taking just the first two terms of a Taylor expansion (see Expression B.3) and discarding the remainder. Then

$$\begin{aligned} Var\,(g(T_n)) &\approx Var\,(g(\theta) + g'(\theta)(T_n - \theta)) \\ &= 0 + g'(\theta)^2 Var(T_n) + 0 \\ &= g'(\theta)^2 Var(T_n). \end{aligned} \tag{B.6}$$

The only reason for making T_n a statistic that estimates θ is so it will be reasonable to expand $g(t)$ about θ. Actually, T_n could be any random variable and θ could be any real number, but in that case the approximation could be arbitrarily bad.

Formula (B.6) for the variance of a function is quite general. We don't need `taylor`; instead, we'll just use `Sage` to take the derivative, square it, multiply by the variance of T_n, and simplify.

```
# Variance of log odds
var('n theta')
g = log(theta/(1-theta))
vTn = theta*(1-theta)/n
v = ( g.derivative(theta)^2 * vTn ).factor(); v
```

evaluate

$$-\frac{1}{(\theta-1)n\theta}$$

Let's try a numerical example, with $\theta = 0.1$ and $n = 200$.

```
v(theta=0.1,n=200)
```

evaluate

0.0555555555555556

Is this a good approximation? We certainly can't take it for granted. Now, for any fixed n, the random variable $\overline{X}_n$ (also known as T_n) is just $\frac{X}{n}$, where X is binomial with parameters n and θ. So,

$$\begin{aligned} Y = Y(X) &= \log \frac{\overline{X}}{1-\overline{X}} \\ &= \log \frac{X/n}{1-X/n} \\ &= \log \frac{X}{n-X}, \end{aligned}$$

and we can calculate

$$\begin{aligned} E(Y) &= \sum_{x=0}^{n} y(x) Pr\{X=x\} \\ &= \sum_{x=0}^{n} \log\left(\frac{X}{n-X}\right) Pr\{X=x\} \\ &= \sum_{x=0}^{n} \log\left(\frac{X}{n-X}\right) \binom{n}{x} \theta^x (1-\theta)^{n-x}. \end{aligned}$$

The calculation of $E(Y^2)$ is similar, and then $Var(Y) = E(Y^2) - [E(Y)]^2$.

Because we're actually going to do it (an insane proposition by hand), we notice that *the variance of the estimated log odds is not even defined* for any finite n. Everything falls apart for $x = 0$ and $x = n$.

Now in standard categorical data analysis, it assumed that θ is strictly between zero and one, and the sample size is large enough so that the events $X = 0$ and $X = n$ (whose probability goes to zero as $n \to \infty$ do not occur. In practice if they did occur, the statistician would move to a different technology. So, the variance we want is actually *conditional* on $1 \leq X \leq n-1$.

Adjusting $Pr\{X = x\}$ to make it a conditional probability involves dividing by $1 - Pr\{X = 0\} - Pr\{X = n\}$, which for $n = 200$ is a number extremely close to one. So will it be okay to just discard $x = 0$ and $x = n$ rather than really adjusting? Take a look at how small the probabilities are.

```
# Is it okay to just drop x=0 and x=200?
p(x) = n.factorial()/(x.factorial() * (n-x).factorial()) * theta^x * (1-theta)^(n-x)
p(0)(theta=0.1); p(200)(theta=0.1)
```

evaluate

$7.05507910865537 \times 10^{-10}$
$1.00000000000001 \times 10^{-200}$

Okay, we'll just sum from $x = 1$ to $x = n-1$, and call it an "exact" calculation. In the Sage work below, note that because n is so large, the binomial coefficient in $p(x)$ can be big enough to overflow the computer's memory, while at the same time the product of θ and $(1-\theta)$ values can be small enough to underflow. To avoid the numerical inaccuracy that would come from this, θ is written as a ratio of two integers. Then inside the loop, $p(x)$ is evaluated by exact integer arithmetic and then factored, resulting in numerous cancellations so that the result is as accurate as possible before it is numerically evaluated and multiplied by the numerical version of $log\frac{x}{n-x}$. By the way, it's a *lot* faster to do it this way rather than doing the whole calculation symbolically and then numerically evaluating the final result.

```
# Calculate exactly, trying to minimize rounding error
y(x) = log(x/(n-x))
n=200; EY=0.0; EYsq=0.0
for x in interval(1,n-1) :
    EY = EY + y(x).n()*(p(x)(theta=1/10).factor().n())
    EYsq = EYsq + (y(x)^2).n()*(p(x)(theta=1/10).factor().n())
vxact = EYsq-EY^2; vxact
```

evaluate

```
0.0595418877731042
```

As a check on this, one can randomly generate a large number of Binomial(n, θ) pseudo-random numbers. Dividing each one by n gives a random sample of $\overline{X}_n$ values, and then computing any function of the $\overline{X}_n$ values yields a collection of random variables that is a nice estimate of the sampling distribution of the statistic in question. With ten million Binomial(n, θ) values, this approach is used to approximate $Var\left(\log\left(\frac{\overline{X}_n}{1-\overline{X}_n}\right)\right)$.

```
> set.seed(9999)
> n = 200; theta = 0.1; m=10000000
> xbar = rbinom(m,n,theta)/n
> logodds = log(xbar/(1-xbar))
> var(logodds)
[1] 0.05955767
```

So the "exact" calculation is right, and the Taylor series approximation is pretty close. Is it a coincidence? No. By the Law of Large Numbers, the probability distribution of the sample proportion $\overline{X}_n$ becomes increasingly concentrated around θ as the sample size increases, so that within a tiny interval enclosing θ, the linear approximation of $g(t)$ in (B.6) is very accurate in the neighbourhood where most of the probability distribution resides. As the sample size increases, it becomes even better, and the approximation of the variance becomes even better.

As a final note about Taylor series, Sage can easily calculate truncated Taylor series approximations of functions of several variables, in which derivatives are replaced by matrices of partial derivatives (Jacobians).

Matrices and linear algebra

Sage is very good at matrix calculations with numbers, but Sage's ability to do matrix calculations with symbols is what makes it useful for structural equation modeling. The algorithm that Sage uses for a particular task will depend on the *ring* (a concept from Algebra) to which the matrix belongs. When the contents of a matrix are symbols, the matrix belongs to the symbolic ring, abbreviated SR. As in Python, a matrix is a list of rows, and the rows are lists of matrix elements.

```
var('alpha beta gamma delta')
A = matrix( SR, [[alpha, beta],[gamma, delta]] ); A
```

evaluate

$$\begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix}$$

Also as in Python, index numbering begins with zero, not one. This may be easy to forget.

```
A[0,1]
```

evaluate

$$\beta$$

Of course you need not be bound by this awkward convention, but in the following example you do need to remember that A[0,0] $= x_{11}$. By the way, I cannot figure out how to get nice-looking double subscripts separated by commas; I don't even know if it's possible. However, it's not a problem for small examples.

```
# Note the nice subscripts
var('x11 x12 x13 x21 x22 x23')
B = matrix(SR, [[x11, x12, x13], [x21, x22, x23]])
B
```

evaluate

$$\begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \end{pmatrix}$$

Multiplication by a scalar does what you would hope.

```
a*A
```

evaluate

$$\begin{pmatrix} 2\,\alpha & 2\,\beta \\ 2\,\gamma & 2\,\delta \end{pmatrix}$$

Matrix multiplication also uses asterisks. Of course the matrices must be the right size or Sage raises an error.

```
C = A*B; C
```

evaluate

$$\begin{pmatrix} \alpha x_{11} + \beta x_{21} & \alpha x_{12} + \beta x_{22} & \alpha x_{13} + \beta x_{23} \\ \gamma x_{11} + \delta x_{21} & \gamma x_{12} + \delta x_{22} & \gamma x_{13} + \delta x_{23} \end{pmatrix}$$

Transpose, inverse, trace, determinant — all are available using a notation that quickly becomes natural if it is not already. First look at **A** again, and then the transpose.

```
show(A)
A.transpose()
```

evaluate

$$\begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix}$$

$$\begin{pmatrix} \alpha & \gamma \\ \beta & \delta \end{pmatrix}$$

```
A.trace()
```

evaluate

$$\alpha + \delta$$

```
A.determinant()
```

evaluate

$$\alpha\delta - \beta\gamma$$

The following result runs off the page (Sage has a scrollbar) and is a reminder of Sage's ability to expressions that are almost too complicated to look at.

```
D = C.transpose() # C is 2x3, D is 3x2
E = (C*D).inverse()   # Inverse of C*D
factor(E) # E is HUGE! This is not as bad. Factor is a good way to simplify.
```

evaluate

$$\left(\begin{array}{ll} \frac{\gamma^2 x_{11}^2+\gamma^2 x_{12}^2+\gamma^2 x_{13}^2+2\,\delta\gamma x_{11}x_{21}+\delta^2 x_{21}^2+2\,\delta\gamma x_{12}x_{22}+\delta^2 x_{22}^2+2\,\delta\gamma x_{13}x_{23}+\delta^2 x_{23}^2}{\left(x_{12}^2x_{21}^2+x_{13}^2x_{21}^2-2\,x_{11}x_{12}x_{21}x_{22}+x_{11}^2x_{22}^2+x_{13}^2x_{22}^2-2\,x_{11}x_{13}x_{21}x_{23}-2\,x_{12}x_{13}x_{22}x_{23}+x_{11}^2x_{23}^2+x_{12}^2x_{23}^2\right)(\alpha\delta-\beta\gamma)^2} & -\frac{\alpha\gamma x_{11}^2+\alpha\gamma x_{12}^2+\alpha\gamma x_{13}^2+\alpha}{(x_{12}^2x_{21}^2+x_{13}^2x_{21}^2-2\,x_{11}x_1} \\ -\frac{\alpha\gamma x_{11}^2+\alpha\gamma x_{12}^2+\alpha\gamma x_{13}^2+\alpha\delta x_{11}x_{21}+\beta\gamma x_{11}x_{21}+\beta\delta x_{21}^2+\alpha\delta x_{12}x_{22}+\beta\gamma x_{12}x_{22}+\beta\delta x_{22}^2+\alpha\delta x_{13}x_{23}+\beta\gamma x_{13}x_{23}+\beta\delta x_{23}^2}{\left(x_{12}^2x_{21}^2+x_{13}^2x_{21}^2-2\,x_{11}x_{12}x_{21}x_{22}+x_{11}^2x_{22}^2+x_{13}^2x_{22}^2-2\,x_{11}x_{13}x_{21}x_{23}-2\,x_{12}x_{13}x_{22}x_{23}+x_{11}^2x_{23}^2+x_{12}^2x_{23}^2\right)(\alpha\delta-\beta\gamma)^2} & \frac{\alpha^2 x_{11}^2+\alpha^2}{(x_{12}^2x_{21}^2+x_{13}^2x_{21}^2-2\,x_{11}x_1} \end{array}\right.$$

```
A.inverse() # Here is something we can look at without a scrollbar.
```

evaluate

$$\left(\begin{array}{rr} \frac{1}{\alpha}+\frac{\beta\gamma}{\alpha^2\left(\delta-\frac{\beta\gamma}{\alpha}\right)} & -\frac{\beta}{\alpha\left(\delta-\frac{\beta\gamma}{\alpha}\right)} \\ -\frac{\gamma}{\alpha\left(\delta-\frac{\beta\gamma}{\alpha}\right)} & \frac{1}{\delta-\frac{\beta\gamma}{\alpha}} \end{array}\right)$$

```
Ainverse = factor(_) # Factor the last expression.
Ainverse
```

evaluate

$$\left(\begin{array}{rr} \frac{\delta}{\alpha\delta-\beta\gamma} & -\frac{\beta}{\alpha\delta-\beta\gamma} \\ -\frac{\gamma}{\alpha\delta-\beta\gamma} & \frac{\alpha}{\alpha\delta-\beta\gamma} \end{array}\right)$$

That's better. Notice how Sage quietly assumes that $\alpha\delta \neq \beta\gamma$. This is typical behaviour, and usually what you want.

For a numerical (or partly numerical) example, just treat the matrix as a function.

```
Ainverse(alpha=1,gamma=2)
```

evaluate

$$\left(\begin{array}{rr} -\frac{\delta}{2\,\beta-\delta} & \frac{\beta}{2\,\beta-\delta} \\ \frac{2}{2\,\beta-\delta} & -\frac{1}{2\,\beta-\delta} \end{array}\right)$$

It's easy to get at the contents.

```
denominator(Ainverse[0,1])
```

evaluate

$$\alpha\delta-\beta\gamma$$

Recall the earlier example.

```
C
```

evaluate

$$\left(\begin{array}{rrr} \alpha x_{11} + \beta x_{21} & \alpha x_{12} + \beta x_{22} & \alpha x_{13} + \beta x_{23} \\ \gamma x_{11} + \delta x_{21} & \gamma x_{12} + \delta x_{22} & \gamma x_{13} + \delta x_{23} \end{array}\right)$$

We had $\mathbf{D} = \mathbf{C}^\top$, so $\mathbf{D}$ is 3×2.

```
(D.nrows(),D.ncols()) # A tuple
```

evaluate

$$(3, 2)$$

This means **DC** is 3×3. It's awful to look at, but since the rank of a product is the minimum of the rank of the matrices being multiplied, the rank of **DC** must be two (with Sage's usual optimistic assumptions about symbolic functions not being equal to zero unless there is more information).

```
DC = D*C
DC.rank()
```

evaluate

$$2$$

```
A.eigenvalues() # Returns a list
```

evaluate

$$\left[\frac{1}{2}\,\alpha + \frac{1}{2}\,\delta - \frac{1}{2}\,\sqrt{\alpha^2 - 2\,\alpha\delta + \delta^2 + 4\,\beta\gamma}, \frac{1}{2}\,\alpha + \frac{1}{2}\,\delta + \frac{1}{2}\,\sqrt{\alpha^2 - 2\,\alpha\delta + \delta^2 + 4\,\beta\gamma}\right]$$

The eigenvalues of a real symmetric matrix are real, and observe that in the last result the expression under the square root sign will be non-negative if **A** is symmetric — that is, if $\beta = \gamma$. Sage doesn't care about this; imaginary numbers are fine.

This is really just the basics. Sage's capabilities in linear algebra go much deeper, including Cholesky and Jordan decompositions, vector spaces and subspaces – the list goes on. As usual, you need to know the math to use it effectively. We have all we need for now.

Applications to structural equation modeling In structural equation modeling, we often find ourselves calculating the covariance matrix of the observable data as a function of the model parameters. For real-world models with lots of variables this can be a big, tedious job. It's largely a clerical task that Sage can do for you. Here, we'll just calculate the covariance matrices for a couple of structural equation models to illustrate how it goes. It's even easier with the `sem` package of Section B.2.

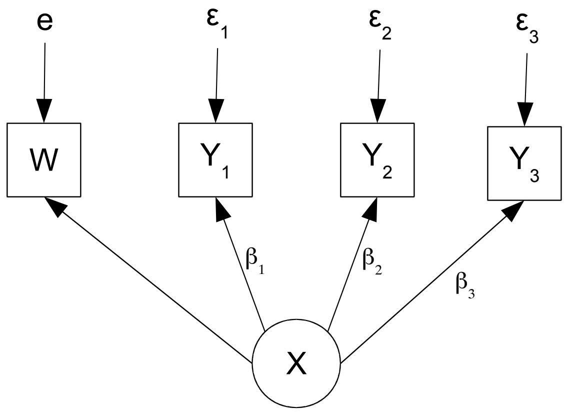
Example B.1.1

The first example is a small regression model with one latent explanatory variable and three observable response variables. A path diagram is shown in Figure B.1. Independently for $i = 1, \ldots, n$,

$$\begin{aligned} W_i &= X_i + e_i \\ Y_{i,1} &= \beta_1 X_i + \epsilon_{i,1} \\ Y_{i,2} &= \beta_2 X_i + \epsilon_{i,2} \\ Y_{i,3} &= \beta_3 X_i + \epsilon_{i,3}, \end{aligned}$$

where X_i, e_i, $\epsilon_{i,1}$, $\epsilon_{i,2}$ and $\epsilon_{i,3}$ are all independent, $Var(X_i) = \phi$, $Var(e_i) = \omega$, $Var(\epsilon_{i,1}) = \psi_1$, $Var(\epsilon_{i,2}) = \psi_2$, $Var(\epsilon_{i,3}) = \psi_3$, all expected values are zero, and the regression coefficients β_1, β_2 and β_3 are fixed constants.

Figure B.1: Path diagram for Example B.1.1



To calculate the covariance matrix, write the model equations in matrix form as

$$\mathbf{Y}_i = \boldsymbol{\beta} \mathbf{X}_i + \boldsymbol{\epsilon}_i,$$

with $\mathbf{X}_i$ and $\boldsymbol{\epsilon}_i$ independent, $cov(\mathbf{X}_i) = \boldsymbol{\Phi}$, and $cov(\boldsymbol{\epsilon}_i) = \boldsymbol{\Psi}$. In the present case, this means

$$\begin{pmatrix} W_i \\ Y_{i,1} \\ Y_{i,2} \\ Y_{i,3} \end{pmatrix} = \begin{pmatrix} 1 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{pmatrix} (X_i) + \begin{pmatrix} e_i \\ \epsilon_{i,1} \\ \epsilon_{i,2} \\ \epsilon_{i,3} \end{pmatrix},$$

with $cov(X_i) = \boldsymbol{\Phi}$ equal to the 1×1 matrix (ϕ), and

$$cov \begin{pmatrix} e_i \\ \epsilon_{i,1} \\ \epsilon_{i,2} \\ \epsilon_{i,3} \end{pmatrix} = \boldsymbol{\Psi} = \begin{pmatrix} \omega & 0 & 0 & 0 \\ 0 & \psi_1 & 0 & 0 \\ 0 & 0 & \psi_2 & 0 \\ 0 & 0 & 0 & \psi_3 \end{pmatrix}.$$

The variance-covariance matrix of the observable variables is then

$$\begin{aligned} cov(\mathbf{Y}_i) &= cov\left(\boldsymbol{\beta}\mathbf{X}_i + \boldsymbol{\epsilon}_i\right) \\ &= \boldsymbol{\beta}\boldsymbol{\Phi}\boldsymbol{\beta}^\top + \boldsymbol{\Psi}. \end{aligned}$$

This is the quantity we'll compute with Sage.

```
# Ex 1 - Single measurement but 3 response variables
beta = matrix(SR,4,1) # SR is the Symbolic Ring. Want 4 rows, 1 col.
beta[0,0] = 1 ; beta[1,0] = var('beta1'); beta[2,0] = var('beta2');
beta[3,0] = var('beta3')
beta
```

evaluate

$$\begin{pmatrix} 1 \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{pmatrix}$$

```
Phi = matrix(SR,1,1); Phi[0,0] = var('phi')
show(Phi)
Psi = matrix(SR,4,4)
Psi[0,0] = var('omega'); Psi[1,1] = var('psi1')
Psi[2,2] = var('psi2'); Psi[3,3] = var('psi3')
Psi
```

evaluate

$$\begin{pmatrix} \phi \end{pmatrix}$$

$$\begin{pmatrix} \omega & 0 & 0 & 0 \\ 0 & \psi_1 & 0 & 0 \\ 0 & 0 & \psi_2 & 0 \\ 0 & 0 & 0 & \psi_3 \end{pmatrix}$$

```
Sigma = beta*Phi*beta.transpose() + Psi ; Sigma
```

evaluate

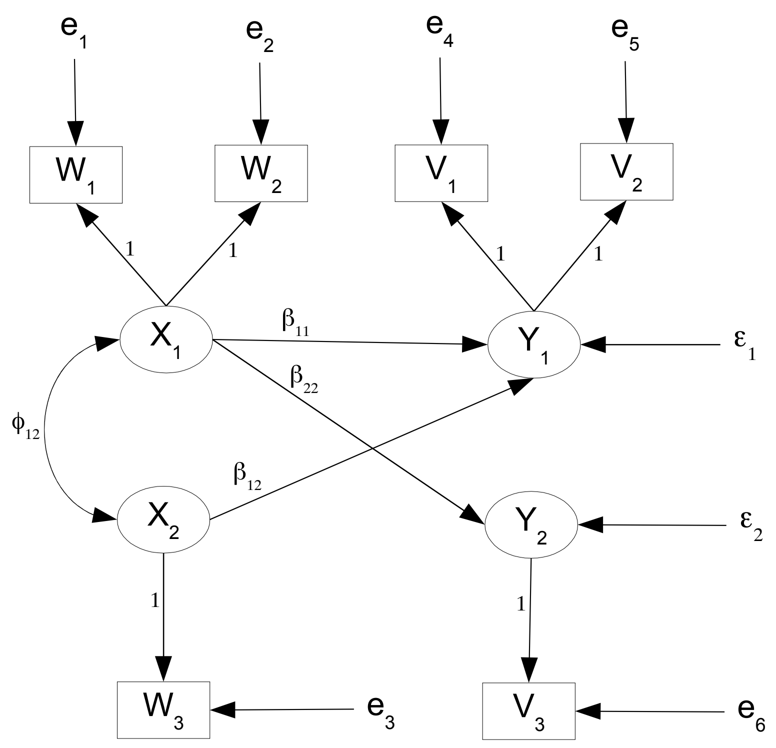
$$\begin{pmatrix} \omega + \phi & \beta_1\phi & \beta_2\phi & \beta_3\phi \\ \beta_1\phi & \beta_1^2\phi + \psi_1 & \beta_1\beta_2\phi & \beta_1\beta_3\phi \\ \beta_2\phi & \beta_1\beta_2\phi & \beta_2^2\phi + \psi_2 & \beta_2\beta_3\phi \\ \beta_3\phi & \beta_1\beta_3\phi & \beta_2\beta_3\phi & \beta_3^2\phi + \psi_3 \end{pmatrix}$$

It is clear that all the parameters will be identifiable provided that at least two of the three regression coefficients are non-zero. This condition could be verified in practice by testing whether simple correlations are different from zero.

Example B.1.2

This example is a latent variable regression that does not fit the standard rules. The latent variable component is over-identified while the measurement component is under-identified. Parameter identifiability for the combined model is unknown, and it's back to the drawing board. Here is the path diagram.

Figure B.2: Path Diagram for Example B.1.2



The distinctive features of this model are that while Y_1 depends on both X_1 and X_2, Y_2 depends only on X_2 — and at the same time, there is double measurement of X_1 and Y_1, but only single measurement of X_2 and Y_2. There are 14 unknown parameters and $6(6+1)/2 = 21$ covariance structure equations, so the model passes the test of the Parameter Count Rule. Identifiability is possible, but not guaranteed. The first step is to calculate the 21 unique variances and covariances, a substantial amount of work if the calculation is done by hand.

It's a lot easier with Sage, but still a bit more challenging than Example B.1.1. First, we calculate the covariance matrix for a latent model, stitching together a partitioned matrix consisting of the variance of the exogenous variables, the covariance of the exogenous and endogenous variables, and the variance of the endogenous variables. Then that matrix is used as the covariance matrix of the latent variables ("factors") in a measurement model. The model equations are (independently for $i = 1, \ldots, n$)

$$\begin{pmatrix} Y_{i,1} \\ Y_{i,2} \end{pmatrix} = \begin{pmatrix} \beta_{1,1} & \beta_{1,2} \\ 0 & \beta_{2,2} \end{pmatrix} \begin{pmatrix} X_{i,1} \\ X_{i,2} \end{pmatrix} + \begin{pmatrix} \epsilon_{i,1} \\ \epsilon_{i,2} \end{pmatrix}$$

and

$$\mathbf{D}_i = \begin{pmatrix} W_{i,1} \\ W_{i,2} \\ W_{i,3} \\ V_{i,1} \\ V_{i,2} \\ V_{i,3} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_{i,1} \\ X_{i,2} \\ Y_{i,1} \\ Y_{i,2} \end{pmatrix} + \begin{pmatrix} e_{i,1} \\ e_{i,2} \\ e_{i,3} \\ e_{i,4} \\ e_{i,5} \\ e_{i,6} \end{pmatrix},$$

where

- $cov \begin{pmatrix} X_{i,1} \\ X_{i,2} \end{pmatrix} = \mathbf{\Phi}_x = \begin{pmatrix} \phi_{11} & \phi_{12} \\ \phi_{12} & \phi_{22} \end{pmatrix}$,
- $\mathbf{\Phi}_x$ is positive definite,
- $cov(\epsilon_{i,1}) = \psi_2$, $cov(\epsilon_{i,2}) = \psi_2$,
- $cov(e_{i,j}) = \omega_j$ for $j = 1, \ldots, 6$ and
- All the error terms are independent of one another, and independent of $X_{i,1}$ and $X_{i,2}$.

To calculate the covariance matrix of the observed data $\mathbf{D}_i$, write the model equations as

$$\begin{aligned} \mathbf{Y}_i &= \boldsymbol{\beta}\mathbf{X}_i + \boldsymbol{\epsilon}_i \\ \mathbf{D}_i &= \mathbf{\Lambda}\mathbf{F}_i + \mathbf{e}_i, \end{aligned}$$

where $\mathbf{F}_i = \left(\begin{array}{c} \mathbf{X}_i \\ \hline \mathbf{Y}_i \end{array} \right)$. That is, the vector of latent variables or "factors" is just $\mathbf{X}_i$ stacked on top of $\mathbf{Y}_i$. Denoting the variance-covariance matrices by $cov(\mathbf{X}_i) = \mathbf{\Phi}_x$, $cov(\boldsymbol{\epsilon}_i) = \mathbf{\Psi}$ and $cov(\mathbf{e}_i) = \mathbf{\Omega}$, we first calculate the variance-covariance matrix of $\mathbf{F}_i$ as the partitioned matrix

$$cov(\mathbf{F}_i) = \mathbf{\Phi} = \left(\begin{array}{c|c} \mathbf{\Phi}_x & \mathbf{\Phi}_x\boldsymbol{\beta}^\top \\ \hline \boldsymbol{\beta}\mathbf{\Phi}_x & \boldsymbol{\beta}\mathbf{\Phi}_x\boldsymbol{\beta}^\top + \mathbf{\Psi} \end{array} \right),$$

and then using that, the variance-covariance matrix of the observed data:

$$cov(\mathbf{D}_i) = \mathbf{\Sigma} = \mathbf{\Lambda}\mathbf{\Phi}\mathbf{\Lambda}^\top + \mathbf{\Omega}.$$

Here is the calculation in `Sage`.

```
# Ex 2 - More challenging

# Y = beta X + epsilon
# F = (X,Y)'
# D = Lambda F + e
# cov(X) = Phi11, cov(epsilon) = Psi, cov(e) = Omega

# Set up matrices
beta = matrix(SR,2,2)
beta[0,0] = var('beta11'); beta[0,1] = var('beta12')
beta[1,0] = var('beta21');  beta[1,1] = var('beta22')
beta[1,0] = 0
show(beta)
```

evaluate

$$\begin{pmatrix} \beta_{11} & \beta_{12} \\ 0 & \beta_{22} \end{pmatrix}$$

```
Phi11 =  matrix(SR,2,2) # cov(X), Symmetric
Phi11[0,0] = var('phi11'); Phi11[0,1] = var('phi12')
Phi11[1,0] = var('phi12'); Phi11[1,1] = var('phi22')
show(Phi11)
```

evaluate

$$\begin{pmatrix} \phi_{11} & \phi_{12} \\ \phi_{12} & \phi_{22} \end{pmatrix}$$

```
Psi =  matrix(SR,2,2) # cov(epsilon)
Psi[0,0] = var('psi1') ; Psi[1,1] = var('psi2')
show(Psi)
```

evaluate

$$\begin{pmatrix} \psi_{1} & 0 \\ 0 & \psi_{2} \end{pmatrix}$$

```
Omega =  matrix(SR,6,6) # cov(e)
Omega[0,0] = var('omega1') ; Omega[1,1] = var('omega2')
Omega[2,2] = var('omega3') ; Omega[3,3] = var('omega4')
Omega[4,4] = var('omega5'); Omega[5,5] = var('omega6')
show(Omega)
```

evaluate

$$\left(\begin{array}{rrrrrr} \omega_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \omega_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & \omega_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & \omega_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & \omega_5 & 0 \\ 0 & 0 & 0 & 0 & 0 & \omega_6 \end{array}\right)$$

```
Lambda = matrix(SR,6,4)
Lambda[0,0]=1; Lambda[1,0]=1; Lambda[2,1]=1
Lambda[3,2]=1; Lambda[4,2]=1; Lambda[5,3]=1
show(Lambda)
```

evaluate

$$\left(\begin{array}{rrrr} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right)$$

```
# Calculate Phi = cov(F)
EXY = Phi11 * beta.transpose()
VY = beta*Phi11*beta.transpose() + Psi
top = Phi11.augment(EXY) # Phi11 on left, EXY on right
bot = EXY.transpose().augment(VY)
Phi = (top.stack(bot)).factor() # Stack top over bot, then factor
show(Phi)
```

evaluate

$$\left(\begin{array}{rrrr} \phi_{11} & \phi_{12} & \beta_{11}\phi_{11} + \beta_{12}\phi_{12} & \beta_{22}\phi_{12} \\ \phi_{12} & \phi_{22} & \beta_{11}\phi_{12} + \beta_{12}\phi_{22} & \beta_{22}\phi_{22} \\ \beta_{11}\phi_{11} + \beta_{12}\phi_{12} & \beta_{11}\phi_{12} + \beta_{12}\phi_{22} & \beta_{11}^2\phi_{11} + 2\,\beta_{11}\beta_{12}\phi_{12} + \beta_{12}^2\phi_{22} + \psi_1 & (\beta_{11}\phi_{12} + \beta_{12}\phi_{22})\beta_{22} \\ \beta_{22}\phi_{12} & \beta_{22}\phi_{22} & (\beta_{11}\phi_{12} + \beta_{12}\phi_{22})\beta_{22} & \beta_{22}^2\phi_{22} + \psi_2 \end{array}\right)$$

```
# Calculate Sigma = cov(D)
Sigma = Lambda * Phi * Lambda.transpose() + Omega
show(Sigma)
```

evaluate

$$\left(\begin{array}{rrrrr} \omega_1 + \phi_{11} & \phi_{11} & \phi_{12} & \beta_{11}\phi_{11} + \beta_{12}\phi_{12} & \beta_{11}\phi_{11} + \\ \phi_{11} & \omega_2 + \phi_{11} & \phi_{12} & \beta_{11}\phi_{11} + \beta_{12}\phi_{12} & \beta_{11}\phi_{11} + \\ \phi_{12} & \phi_{12} & \omega_3 + \phi_{22} & \beta_{11}\phi_{12} + \beta_{12}\phi_{22} & \beta_{11}\phi_{12} + \\ \beta_{11}\phi_{11} + \beta_{12}\phi_{12} & \beta_{11}\phi_{11} + \beta_{12}\phi_{12} & \beta_{11}\phi_{12} + \beta_{12}\phi_{22} & \beta_{11}^2\phi_{11} + 2\,\beta_{11}\beta_{12}\phi_{12} + \beta_{12}^2\phi_{22} + \omega_4 + \psi_1 & \beta_{11}^2\phi_{11} + 2\,\beta_{11}\beta_{12}\phi_{12} + \beta_{12}^2\phi_{2} \\ \beta_{11}\phi_{11} + \beta_{12}\phi_{12} & \beta_{11}\phi_{11} + \beta_{12}\phi_{12} & \beta_{11}\phi_{12} + \beta_{12}\phi_{22} & \beta_{11}^2\phi_{11} + 2\,\beta_{11}\beta_{12}\phi_{12} + \beta_{12}^2\phi_{22} + \psi_1 & \beta_{11}^2\phi_{11} + 2\,\beta_{11}\beta_{12}\phi_{12} + \beta_{12}^2\phi_{22} + \omega \\ \beta_{22}\phi_{12} & \beta_{22}\phi_{12} & \beta_{22}\phi_{22} & (\beta_{11}\phi_{12} + \beta_{12}\phi_{22})\beta_{22} & (\beta_{11}\phi_{12} + \beta_{12}\phi \end{array}\right.$$

Again, this is the covariance matrix of the observable data vector $\mathbf{D}_i = (W_{i,1}, W_{i,2}, W_{i,3}, V_{i,1}, V_{i,2}, V_{i,3})^\top$. The covariance matrix is big and the last two columns got cut off, but in `Sage` you can scroll to the right and see something like the following:

$$\left. \cdots \begin{array}{rrr}
\beta_{11}\phi_{11} + \beta_{12}\phi_{12} & \beta_{11}\phi_{11} + \beta_{12}\phi_{12} & \beta_{22}\phi_{12} \\
\beta_{11}\phi_{11} + \beta_{12}\phi_{12} & \beta_{11}\phi_{11} + \beta_{12}\phi_{12} & \beta_{22}\phi_{12} \\
\beta_{11}\phi_{12} + \beta_{12}\phi_{22} & \beta_{11}\phi_{12} + \beta_{12}\phi_{22} & \beta_{22}\phi_{22} \\
\beta_{11}^2\phi_{11} + 2\,\beta_{11}\beta_{12}\phi_{12} + \beta_{12}^2\phi_{22} + \omega_4 + \psi_1 & \beta_{11}^2\phi_{11} + 2\,\beta_{11}\beta_{12}\phi_{12} + \beta_{12}^2\phi_{22} + \psi_1 & (\beta_{11}\phi_{12} + \beta_{12}\phi_{22})\beta_{22} \\
\beta_{11}^2\phi_{11} + 2\,\beta_{11}\beta_{12}\phi_{12} + \beta_{12}^2\phi_{22} + \psi_1 & \beta_{11}^2\phi_{11} + 2\,\beta_{11}\beta_{12}\phi_{12} + \beta_{12}^2\phi_{22} + \omega_5 + \psi_1 & (\beta_{11}\phi_{12} + \beta_{12}\phi_{22})\beta_{22} \\
(\beta_{11}\phi_{12} + \beta_{12}\phi_{22})\beta_{22} & (\beta_{11}\phi_{12} + \beta_{12}\phi_{22})\beta_{22} & \beta_{22}^2\phi_{22} + \omega_6 + \psi_2
\end{array} \right)$$

Now it appears that at points in the parameter space where $\phi_{12} \neq 0$, the regression parameters β_{11}, β_{12} and β_{22} may be identifiable in spite of the single measurement. This is just a tentative conclusion based on inspecting the equations without actually doing all the work. We will continue to work on this example using the tools of the `sem` package.

B.2 The `sem` Package

B.2.1 Introduction and Examples

Example B.1.2 showed how Sage can be used to carry out useful symbolic calculations that are too tedious to perform by hand. Still, parts of the job can be repetitive and this can be a barrier to using the technology. In Sage, it is easy for users to write special purpose functions. The `sem` package is a collection of functions for structural equation modeling. Currently, it is limited to symbolic calculation. For numerical model fitting, it is necessary to use specialized statistical software[9].

To load the `sem` package,

```
sem = 'http://www.utstat.toronto.edu/~brunner/openSEM/sage/sem.sage'
load(sem)
# load('~/sem.sage') # To load a local version in your home directory
```

evaluate

After the package is loaded, `Contents()` will display a list of the available functions. For help on a particular function, type the function name followed by a question mark, like `PathVar?`

The `sem` package currently includes the following functions. You can go directly to the documentation for a particular function, or continue reading to see how the functions are used together in context.

[9]Sage has very strong numerical capabilities, and it would not be very difficult to write a function to do numerical maximum likelihood estimation. What holds me back is the issue of starting values. Programs like `Amos`, `Lisrel` and SAS `proc calis` have extensive bags of tricks for generating automatic starting values, and typically they are very good. It is difficult to appreciate how convenient they are until you have tried to come up with your own starting values for a few models.

1. Matrix Creation

   1a) `DiagonalMatrix(size,symbol='psi',double=False)`

   1b) `GeneralMatrix(nrows,ncols,symbol)`

   1c) `IdentityMatrix(size)`

   1d) `SymmetricMatrix(size,symbol,corr=False)`

   1e) `ZeroMatrix(nrows,ncols)`

2. Covariance Matrix Calculation

   2a) `EqsVar(beta,gamma,Phi,oblist,simple=True)`

   2b) `FactorAnalysisVar(Lambda,Phi,Omega)`

   2c) `NoGammaVar(Beta,Psi)`

   2d) `PathVar(Phi,Beta,Gamma,Psi,simple=True)`

   2e) `RegressionVar(Phi,Gamma,Psi,simple=True)`

3. Manipulation

   3a) `GroebnerBasis(polynomials,variables)`

   3b) `LSTarget(M,x,y)`

   3c) `Parameters(M)`

   3d) `SigmaOfTheta(M,symbol='sigma')`

   3e) `Simplify(x)`

4. Utility

   4a) `BetaCheck(Beta)`

   4b) `Contents()`

   4c) `CovCheck(Psi)`

   4d) `MultCheck(Beta,Psi)`

   4e) `Pad(M)`

Here is Example B.1.2 again from the beginning, using the `sem` package. Repeating the model equations,

$$
\begin{array}{ccccccc}
\mathbf{Y}_i & = & \boldsymbol{\beta} & \mathbf{X}_i & + & \boldsymbol{\epsilon}_i \\
\begin{pmatrix} Y_{i,1} \\ Y_{i,2} \end{pmatrix} & = & \begin{pmatrix} \beta_{1,1} & \beta_{1,2} \\ 0 & \beta_{2,2} \end{pmatrix} & \begin{pmatrix} X_{i,1} \\ X_{i,2} \end{pmatrix} & + & \begin{pmatrix} \epsilon_{i,1} \\ \epsilon_{i,2} \end{pmatrix}
\end{array}
$$

$$
\begin{array}{ccccccc}
\mathbf{D}_i & = & \boldsymbol{\Lambda} & \mathbf{F}_i & + & \mathbf{e}_i \\
\begin{pmatrix} W_{i,1} \\ W_{i,2} \\ W_{i,3} \\ V_{i,1} \\ V_{i,2} \\ V_{i,3} \end{pmatrix} & = & \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} X_{i,1} \\ X_{i,2} \\ Y_{i,1} \\ Y_{i,2} \end{pmatrix} & + & \begin{pmatrix} e_{i,1} \\ e_{i,2} \\ e_{i,3} \\ e_{i,4} \\ e_{i,5} \\ e_{i,6} \end{pmatrix}
\end{array}
$$

```
sem = 'http://www.utstat.toronto.edu/~brunner/openSEM/sage/sem.sage'
load(sem)
# Set up matrices (Remember, indices begin with zero)
beta = GeneralMatrix(2,2,'beta'); beta[1,0]=0
Phi11 = SymmetricMatrix(2,'phi')  # cov(X)
Psi = DiagonalMatrix(2,'psi')     # cov(epsilon)
Omega = DiagonalMatrix(6,'omega') # cov(e)
Lambda = ZeroMatrix(6,4) # Factor loadings
Lambda[0,0]=1; Lambda[1,0]=1; Lambda[2,1]=1
Lambda[3,2]=1; Lambda[4,2]=1; Lambda[5,3]=1
```

evaluate

The GeneralMatrix function generates doubly subscripted symbols by default; it is easy to replace the lower left entry with a zero. The other functions are pretty much self-explanatory, and very handy. In general, native Sage functions are lower case, while functions in the sem package are capitalized. This makes them easy to distinguish in the examples. Next we calculate $\boldsymbol{\Sigma}$ the easy way. The output is not shown because it is big and you have seen it before.

```
# Calculate Phi = cov(F)
Phi = RegressionVar(Phi11,beta,Psi) # The first argument is cov(X)
# Calculate Sigma = cov(D)
Sigma = FactorAnalysisVar(Lambda,Phi,Omega); Sigma
```

evaluate

Based on inspection of $\boldsymbol{\Sigma}$, I tentatively concluded that the parameters were identifiable, at least in most of the parameter space. Now we will nail it down. The SetupEqns function assembles a list of covariance structure equations. Each equation is displayed with its index as a tuple – not very pretty, but useful when one needs to refer to equations by number (starting with zero). Note the Python syntax for looping.

```
eqlist = SetupEqns(Sigma); k = len(eqlist)
for index in range(k): index,eqlist[index]
```

evaluate

$$(0,\ \omega_1 + \phi_{11} = \sigma_{11})$$
$$(1,\ \phi_{11} = \sigma_{12})$$
$$(2,\ \phi_{12} = \sigma_{13})$$
$$(3,\ \beta_{11}\phi_{11} + \beta_{12}\phi_{12} = \sigma_{14})$$
$$(4,\ \beta_{11}\phi_{11} + \beta_{12}\phi_{12} = \sigma_{15})$$
$$(5,\ \beta_{22}\phi_{12} = \sigma_{16})$$
$$(6,\ \omega_2 + \phi_{11} = \sigma_{22})$$
$$(7,\ \phi_{12} = \sigma_{23})$$
$$(8,\ \beta_{11}\phi_{11} + \beta_{12}\phi_{12} = \sigma_{24})$$
$$(9,\ \beta_{11}\phi_{11} + \beta_{12}\phi_{12} = \sigma_{25})$$
$$(10,\ \beta_{22}\phi_{12} = \sigma_{26})$$
$$(11,\ \omega_3 + \phi_{22} = \sigma_{33})$$
$$(12,\ \beta_{11}\phi_{12} + \beta_{12}\phi_{22} = \sigma_{34})$$
$$(13,\ \beta_{11}\phi_{12} + \beta_{12}\phi_{22} = \sigma_{35})$$
$$(14,\ \beta_{22}\phi_{22} = \sigma_{36})$$
$$(15,\ \beta_{11}^2\phi_{11} + 2\,\beta_{11}\beta_{12}\phi_{12} + \beta_{12}^2\phi_{22} + \omega_4 + \psi_1 = \sigma_{44})$$
$$(16,\ \beta_{11}^2\phi_{11} + 2\,\beta_{11}\beta_{12}\phi_{12} + \beta_{12}^2\phi_{22} + \psi_1 = \sigma_{45})$$
$$(17,\ (\beta_{11}\phi_{12} + \beta_{12}\phi_{22})\beta_{22} = \sigma_{46})$$
$$(18,\ \beta_{11}^2\phi_{11} + 2\,\beta_{11}\beta_{12}\phi_{12} + \beta_{12}^2\phi_{22} + \omega_5 + \psi_1 = \sigma_{55})$$
$$(19,\ (\beta_{11}\phi_{12} + \beta_{12}\phi_{22})\beta_{22} = \sigma_{56})$$
$$(20,\ \beta_{22}^2\phi_{22} + \omega_6 + \psi_2 = \sigma_{66})$$

The next step is to assemble a list of model parameters. The function `Parameters` returns a list of the parameters in a parameter matrix — that is, a list of the unique elements that are not one or zero. Unfortunately, it cannot operate on a computed covariance matrix, just on the parmeter matrices that are used as input. Still, it's better than doing the job by hand.

```
# Assemble a list of model parameters. I count 14 by hand.
param = Parameters(beta) # Start with parameters in beta
param.extend(Parameters(Phi11)) # Add the parameters in Phi11
param.extend(Parameters(Psi)) # Add the parameters in Psi
param.extend(Parameters(Omega)) # Add the parameters in Omega
param.extend(Parameters(Lambda)) # Add the parameters in Lambda
show(param)
len(eqlist), len(param) # This many equations in this many unknowns
```

evaluate

$[\beta_{11}, \beta_{12}, \beta_{22}, \phi_{11}, \phi_{12}, \phi_{22}, \psi_1, \psi_2, \omega_1, \omega_2, \omega_3, \omega_4, \omega_5, \omega_6]$
$(21, 14)$

So there are 21 equations in 14 unknowns. Sage's very powerful `solve` function requires the same number of equations as unknowns and will not work here. However, we'll try it anyway to see what happens.

```
solve(eqlist,param,solution_dict=True)
```

evaluate

$[]$

That little rectangle is a left square bracket followed by a right square bracket; it's an empty list (empty set), meaning that the system of equations has no general solution. This happens because, for example, equation number two in the list says $\phi_{12} = \sigma_{13}$, while equation seven says $\phi_{12} = \sigma_{23}$. To Sage, σ_{13} and σ_{23} are just numbers, and there is no reason to assume they are equal. Thus there is no *general* solution.

Actually, because we think of the σ_{ij} values as arising from a single, fixed point in the parameter space, we recognize $\sigma_{13} = \sigma_{24}$ as a distinctive feature that the model imposes on the covariance matrix $\boldsymbol{\Sigma}$. But Sage does not know this, and I don't know how to tell it without specifying exactly what the restrictions are. One solution is to set aside the redundant equations and then give the solve function a system with the same number of equations and unknowns. Unfortunately, this is not automatic because it is not always obvious which equations are redundant. Groebner basis methods (to be discussed later in this appendix) can do the job automatically when they work.

Because there are 21 equations in 14 unknowns, there should be seven equality constraints; seven equations should be redundant. Carefully inspecting the covariance structure equations, I conclude

- σ_{15}, σ_{24} and σ_{25} are redundant with σ_{14}.
- σ_{26} is redundant with σ_{16}.
- σ_{23} is redundant with σ_{13}.
- σ_{35} is redundant with σ_{34}.
- σ_{56} is redundant with σ_{46}.

```
# Set redundant equations aside.
extra = [4,8,9,7,10,13,19] # Indices of redundant equations
extra.sort() # Sort them (change in place)
# Save and display the redundant equations
aside = [] # Empty list to start
for index in extra:
    extraeq = eqlist[index]
    show(extraeq)
    aside.append(extraeq)
```

evaluate

$$\beta_{11}\phi_{11} + \beta_{12}\phi_{12} = \sigma_{15}\phi_{12} = \sigma_{23}$$
$$\beta_{11}\phi_{11} + \beta_{12}\phi_{12} = \sigma_{24}$$
$$\beta_{11}\phi_{11} + \beta_{12}\phi_{12} = \sigma_{25}$$
$$\beta_{22}\phi_{12} = \sigma_{26}$$
$$\beta_{11}\phi_{12} + \beta_{12}\phi_{22} = \sigma_{35}$$
$$(\beta_{11}\phi_{12} + \beta_{12}\phi_{22})\beta_{22} = \sigma_{56}$$

```
# Remove extra equations
for item in aside: eqlist.remove(item)
len(eqlist) # Should be 14 now
```

evaluate

14

```
# Solve, returning solutions as a list of dictionaries
solist = solve(eqlist,param,solution_dict=True)
len(solist) # Should have one item (unique solution)
```

evaluate

0

The length of the list is zero; there are no solutions, meaning no general solutions according to Sage. This is a sure sign of redundancy in the covariance structure equations we are trying to solve. They still imply one or more constraints on the σ_{ij} quantities – constraints that Sage does not accept. In other words, we missed something. Looking at the covariance structure equations again,

```
for item in eqlist: item
```

evaluate

$$\omega_1 + \phi_{11} = \sigma_{11}$$
$$\phi_{11} = \sigma_{12}$$
$$\phi_{12} = \sigma_{13}$$
$$\beta_{11}\phi_{11} + \beta_{12}\phi_{12} = \sigma_{14}$$
$$\beta_{22}\phi_{12} = \sigma_{16}$$
$$\omega_2 + \phi_{11} = \sigma_{22}$$
$$\omega_3 + \phi_{22} = \sigma_{33}$$
$$\beta_{11}\phi_{12} + \beta_{12}\phi_{22} = \sigma_{34}$$
$$\beta_{22}\phi_{22} = \sigma_{36}$$
$$\beta_{11}^2\phi_{11} + 2\,\beta_{11}\beta_{12}\phi_{12} + \beta_{12}^2\phi_{22} + \omega_4 + \psi_1 = \sigma_{44}$$
$$\beta_{11}^2\phi_{11} + 2\,\beta_{11}\beta_{12}\phi_{12} + \beta_{12}^2\phi_{22} + \psi_1 = \sigma_{45}$$
$$(\beta_{11}\phi_{12} + \beta_{12}\phi_{22})\beta_{22} = \sigma_{46}$$
$$\beta_{11}^2\phi_{11} + 2\,\beta_{11}\beta_{12}\phi_{12} + \beta_{12}^2\phi_{22} + \omega_5 + \psi_1 = \sigma_{55}$$
$$\beta_{22}^2\phi_{22} + \omega_6 + \psi_2 = \sigma_{66}$$

To be honest, it took me a while to see it. The parameters ω_6 and ψ_2 appear only in the last equation, as a sum. This means that infinitely many pairs (ω_6, ψ_2) will satisfy the system of equations. Those parameters are not identifiable. A glance at the path diagram on 246 shows why. Because Y_2 does not influence any other variables in the latent model, measuring it just once means that the variance of V_2 is just the variance of Y_2 plus ω_6, with no hope of separating ω_6 from ψ_2.

The solution is easy; re-parameterize by combining ω_6 and ψ_2 into a single variance parameter. This could be accomplished by re-writing the path diagram and running an arrow directly from X_1 to V_3. When a purely endogenous variable (that is, purely endogenous in the latent model) is measured once, pretending that it is measured without error is a standard, harmless trick. Here, it's unnecessary to make a new path diagram and calculate the covariance structure equations again. Just setting $\omega_6 = 0$ would effectively treat $\omega_6 + \psi_2$ as a single parameter now called ψ_2.

But now there are more equations than unknowns, implying another equality constraint I missed. After looking at the equations for a while, I finally saw it. It's the third equation from the bottom. Starting at the third equation from the top, ϕ_{12} is identified from σ_{13}, and using that, β_{22} is identified from σ_{16}. The equation for σ_{46} (third from the bottom) is β_{22} multiplied by the expression for σ_{34}. So the third equation from the bottom is redundant, and induces an equality constraint. Starting with zero, that should be equation eleven. Check it.

```
sig46 = eqlist[11]; sig46
```

evaluate

$$(\beta_{11}\phi_{12} + \beta_{12}\phi_{22})\beta_{22} = \sigma_{46}$$

Now we'll remove that equation from the list of covariance structure equations and add it to the list of equations we set aside. Once we finally get a list of explicit solutions of the covariance structure equations, we can obtain the equality constraints by substituting the solutions into the the equations that were set aside.

```
aside.append(sig46)
eqlist.remove(sig46); len(eqlist)
```

evaluate

```
13
```

We now have thirteen equations in fourteen unknown parameters. Before re-parameterizing by setting $\omega_6 = 0$, let's see how Sage deals with infinitely many solutions. One might expect it to hang up, but the task is completed instantly.

```
# Now there are 13 equations in 14 unknown parameters. See what happens
# when we try to solve. Return the solutions as a list of dictionaries.
solist = solve(eqlist,param,solution_dict=True)
len(solist)
```

evaluate

```
1
```

One dictionary (essentially a Python dictionary) looks like one solution – unique. This is odd. How many items are in the dictionary?

```
sol = solist[0]
len(sol)
```

evaluate

```
14
```

There are fourteen items in the dictionary, suggesting one solution for each of the 14 parameters. This is unexpected, because we know there are infinitely many solutions. Let's take a look. The keys of the dictionary are the parameters, and the corresponding values are the solutions in terms of the σ_{ij}s. As in Python, `dictionary[key]` yields `value`.

```
# Display the solutions. item==sol[item] just causes that equation
# to be displayed.
for item in param: item==sol[item]
```

evaluate

$$\beta_{11} = \frac{\sigma_{16}\sigma_{34}-\sigma_{14}\sigma_{36}}{\sigma_{13}\sigma_{16}-\sigma_{12}\sigma_{36}}$$

$$\beta_{12} = \frac{\sigma_{13}\sigma_{14}\sigma_{16}-\sigma_{12}\sigma_{16}\sigma_{34}}{\sigma_{13}^2\sigma_{16}-\sigma_{12}\sigma_{13}\sigma_{36}}$$

$$\beta_{22} = \frac{\sigma_{16}}{\sigma_{13}}$$

$$\phi_{11} = \sigma_{12}$$

$$\phi_{12} = \sigma_{13}$$

$$\phi_{22} = \frac{\sigma_{13}\sigma_{36}}{\sigma_{16}}$$

$$\psi_1 = -\frac{2\,\sigma_{13}\sigma_{14}\sigma_{16}\sigma_{34}-\sigma_{12}\sigma_{16}\sigma_{34}^2-\sigma_{13}\sigma_{14}^2\sigma_{36}-\left(\sigma_{13}^2\sigma_{16}-\sigma_{12}\sigma_{13}\sigma_{36}\right)\sigma_{45}}{\sigma_{13}^2\sigma_{16}-\sigma_{12}\sigma_{13}\sigma_{36}}$$

$$\psi_2 = r_1$$

$$\omega_1 = \sigma_{11} - \sigma_{12}$$

$$\omega_2 = -\sigma_{12} + \sigma_{22}$$

$$\omega_3 = \frac{\sigma_{16}\sigma_{33}-\sigma_{13}\sigma_{36}}{\sigma_{16}}$$

$$\omega_4 = \sigma_{44} - \sigma_{45}$$

$$\omega_5 = -\sigma_{45} + \sigma_{55}$$

$$\omega_6 = -\frac{r_1\sigma_{13}+\sigma_{16}\sigma_{36}-\sigma_{13}\sigma_{66}}{\sigma_{13}}$$

Scanning down the list, we see $\psi_2 = r_1$. The quantity r_1 (which we have not seen before) is an arbitrary variable that could be anything[10]. I believe the Sage people call it a *parameter*, which is vastly different from a parameter in statistical estimation. Right at the bottom of the list is the solution $\omega_6 = -\frac{r_1\sigma_{13}+\sigma_{16}\sigma_{36}-\sigma_{13}\sigma_{66}}{\sigma_{13}}$. This neatly expresses the infinitely many solutions to the covariance structure equations. All the other solutions are unique (provided that denominators are non-zero), but the pair (ω_6, ψ_2) can be recovered from $\mathbf{\Sigma}$ in infinitely many ways, one for each $r_1 > 0$.

This is so nice that we will not bother to re-parameterize and obtain a unique solution. Of course with real data, one would have to re-parameterize ω_6 and ψ_2 in order to estimate the other parameters by maximum likelihood, because otherwise the maximum would not be unique and there would be unpleasant numerical consequences.

Our main interest is in β_{11}, β_{12} and β_{22}. The existence of unique solutions means that these parameters are identifiable (and as a practical matter, estimable) as long as the denominators are non-zero. The natural thing is to substitute for those σ_{ij} quantities in the denominators, in terms of model parameters. Perhaps the denominators are never zero, or perhaps the βijs can be identified in some other way when they are.

The formula for β_{22} is simplest. Scanning the list of solutions, we see $\phi_{12} = \sigma_{13}$. So, the solution for β_{22} does not apply when the two latent explanatory variables have zero covariance. Perhaps there is another way.

[10]Sage does not know that $r_1 = \psi_2$ is positive, or even that it's a real number.

```
# Can beta22 be identified when phi12=0?
factor(Sigma(phi12=0))
```

evaluate

$$\left(\begin{array}{rrrrrr}
\omega_1+\phi_{11} & \phi_{11} & 0 & \beta_{11}\phi_{11} & \beta_{11}\phi_{11} & 0 \\
\phi_{11} & \omega_2+\phi_{11} & 0 & \beta_{11}\phi_{11} & \beta_{11}\phi_{11} & 0 \\
0 & 0 & \omega_3+\phi_{22} & \beta_{12}\phi_{22} & \beta_{12}\phi_{22} & \beta_{22}\phi_{22} \\
\beta_{11}\phi_{11} & \beta_{11}\phi_{11} & \beta_{12}\phi_{22} & \beta_{11}^2\phi_{11}+\beta_{12}^2\phi_{22}+\omega_4+\psi_1 & \beta_{11}^2\phi_{11}+\beta_{12}^2\phi_{22}+\psi_1 & \beta_{12}\beta_{22}\phi_{22} \\
\beta_{11}\phi_{11} & \beta_{11}\phi_{11} & \beta_{12}\phi_{22} & \beta_{11}^2\phi_{11}+\beta_{12}^2\phi_{22}+\psi_1 & \beta_{11}^2\phi_{11}+\beta_{12}^2\phi_{22}+\omega_5+\psi_1 & \beta_{12}\beta_{22}\phi_{22} \\
0 & 0 & \beta_{22}\phi_{22} & \beta_{12}\beta_{22}\phi_{22} & \beta_{12}\beta_{22}\phi_{22} & \beta_{22}^2\phi_{22}+\omega_6+\psi_2
\end{array}\right)$$

Yes! As long as $\beta_{12} \neq 0$,

$$\frac{\sigma_{46}}{\sigma_{34}} = \frac{\beta_{12}\beta_{22}\phi_{22}}{\beta_{12}\phi_{22}} = \beta_{22}$$

Actually, this way of identifying β_{22} works even when $\phi_{12} \neq 0$. We could scroll up and look at the orignal $\boldsymbol{\Sigma}$ in terms of the parameters. Or, the `sem` package's `pad` function can be used to add a row and column of zeros to a matrix, making it more convenient to refer to the elements.

```
# Does sigma46/sigma34 work without phi12=0?
padSigma = Pad(Sigma)
show(padSigma[4,6]); show(padSigma[3,4])
```

evaluate

$$(\beta_{11}\phi_{12}+\beta_{12}\phi_{22})\beta_{22}$$
$$\beta_{11}\phi_{12}+\beta_{12}\phi_{22}$$

The identifying solution $\beta_{22} = \frac{\sigma_{46}}{\sigma_{34}}$ is superior to the solution $\beta_{22} = \frac{\sigma_{16}}{\sigma_{13}}$ on page 258, because it only fails when *both* $\beta_{12} = 0$ and ($\beta_{11} = 0$ or $\phi_{12} = 0$). Some ways of solving the covariance structure equations are better than others, in the sense that they reveal more clearly where in the parameter space the parameters are identifiable. Sage's `solve` function will not necessarily locate the most informative solution, and neither will you if you do it by hand.

The solution $\beta_{22} = \frac{\sigma_{46}}{\sigma_{34}}$ does not apply when both $\phi_{12} = 0$, and $\beta_{12} = 0$, but it is a good idea to examine $\boldsymbol{\Sigma}$ under these conditions to see if yet another solution appears.

```
# What if both phi12 and beta12 equal zero?
factor(Sigma(phi12=0,beta12=0))
```

evaluate

$$\left(\begin{array}{rrrrrr}
\omega_1+\phi_{11} & \phi_{11} & 0 & \beta_{11}\phi_{11} & \beta_{11}\phi_{11} & 0 \\
\phi_{11} & \omega_2+\phi_{11} & 0 & \beta_{11}\phi_{11} & \beta_{11}\phi_{11} & 0 \\
0 & 0 & \omega_3+\phi_{22} & 0 & 0 & \beta_{22}\phi_{22} \\
\beta_{11}\phi_{11} & \beta_{11}\phi_{11} & 0 & \beta_{11}^2\phi_{11}+\omega_4+\psi_1 & \beta_{11}^2\phi_{11}+\psi_1 & 0 \\
\beta_{11}\phi_{11} & \beta_{11}\phi_{11} & 0 & \beta_{11}^2\phi_{11}+\psi_1 & \beta_{11}^2\phi_{11}+\omega_5+\psi_1 & 0 \\
0 & 0 & \beta_{22}\phi_{22} & 0 & 0 & \beta_{22}^2\phi_{22}+\omega_6+\psi_2
\end{array}\right)$$

It seems that β_{22} is not identifiable when both ϕ_{12}, and β_{12} equal zero. The only way to get at it is through ϕ_{22}, which is not accessible at all. The conclusion is that β_{22} is identifiable if either $\beta_{12} \neq 0$, or if both β_{11} and ϕ_{12} are non-zero.

It is worth noting that the sufficient condition $\beta_{12} \neq 0$ was concealed until we actually set $\phi_{12} = 0$ and took another look at the covariance matrix. The general principle is that when the solution for a parameter in terms of σ_{ij} quantities is a fraction, the parameter is identifiable at points in the parameter space where the denominator is non-zero. While it is tempting to think that identifiability fails where the denominator *is* zero, this need not be the case. If the model imposes equality constraints on the covariance matrix, there may be other ways to recover the parameter.

In our examination of identifiability for β_{22}, it was easy (with Sage) to re-calculate the covariance matrix with $\phi_{12} = 0$ to see if it was possible to solve for β_{22} in that part of the parameter space. Doing this by hand would have been possible though tedious. For β_{11} and β_{12}, hand calculation is almost out of the question because the denominators are so complicated; it's quite easy with Sage and the `sem` package.

```
# Look at beta11 and beta12.
show(beta11 == sol[beta11]);  show(beta12 == sol[beta12])
```

evaluate

$$\beta_{11} = \frac{\sigma_{16}\sigma_{34}-\sigma_{14}\sigma_{36}}{\sigma_{13}\sigma_{16}-\sigma_{12}\sigma_{36}}$$

$$\beta_{12} = \frac{\sigma_{13}\sigma_{14}\sigma_{16}-\sigma_{12}\sigma_{16}\sigma_{34}}{\sigma_{13}^2\sigma_{16}-\sigma_{12}\sigma_{13}\sigma_{36}}$$

To see where in the parameter space the denominators equal zero, we need to take the formulas for the σ_{ij}s in terms of the parameters, and substitute them into the denominators (just the denominators, of course). The `SigmaOfTheta` function of the `sem` package is designed to make this task easy. Given a covariance matrix that is a function of model parameters, `SigmaOfTheta` makes a dictionary that will allow any function of the σ_{ij} variances and covariances to be evaluated at the model parameters. In the following, `SigmaOfTheta` is used to create a dictionary called `theta`, and the denominator of the solution for β_{11} is put into `d1`. Then, `d1(theta)` gives `d1` as a function of the model parameters. The notation is simple and natural, partly because `theta` is a very good name for the dictionary. The `Simplify` function first expands an expression (multiplies it out), and then factors the result. I find it more helpful than Sage's built-in `simplify` function, which is already applied to everything automatically anyway.

```
# Now examine denominators of the solutions to see exactly where in
# the parameter space they equal zero.
theta = SigmaOfTheta(Sigma)
d1 = denominator(sol[beta11])
Simplify(d1(theta))
```

evaluate

$$(\phi_{12}^2 - \phi_{11}\phi_{22})\beta_{22}$$

See how nice that was? The denominator is just $-|\mathbf{\Phi}_x|\beta_{22}$. Since $\mathbf{\Phi}_x$ is positive definite, the denominator will be zero if and only if $\beta_{22} = 0$.

```
# What if beta22=0?
Sigma(beta22=0)
```

evaluate

$$\left(\begin{array}{rrrrr}
\omega_1 + \phi_{11} & \phi_{11} & \phi_{12} & \beta_{11}\phi_{11} + \beta_{12}\phi_{12} & \\
\phi_{11} & \omega_2 + \phi_{11} & \phi_{12} & \beta_{11}\phi_{11} + \beta_{12}\phi_{12} & \\
\phi_{12} & \phi_{12} & \omega_3 + \phi_{22} & \beta_{11}\phi_{12} + \beta_{12}\phi_{22} & \\
\beta_{11}\phi_{11} + \beta_{12}\phi_{12} & \beta_{11}\phi_{11} + \beta_{12}\phi_{12} & \beta_{11}\phi_{12} + \beta_{12}\phi_{22} & \beta_{11}^2\phi_{11} + 2\,\beta_{11}\beta_{12}\phi_{12} + \beta_{12}^2\phi_{22} + \omega_4 + \psi_1 & \beta_{11}^2\phi_{11} + 2\,\beta_{11} \\
\beta_{11}\phi_{11} + \beta_{12}\phi_{12} & \beta_{11}\phi_{11} + \beta_{12}\phi_{12} & \beta_{11}\phi_{12} + \beta_{12}\phi_{22} & \beta_{11}^2\phi_{11} + 2\,\beta_{11}\beta_{12}\phi_{12} + \beta_{12}^2\phi_{22} + \psi_1 & \beta_{11}^2\phi_{11} + 2\,\beta_{11}\beta_{12}\phi_{1} \\
0 & 0 & 0 & 0 &
\end{array}\right.$$

The answer got cut off and there is no scrollbar in this document, but you can see that the only useable equations involving β_{11} are variations of

$$\begin{aligned} \sigma_{42} &= \beta_{11}\phi_{11} + \beta_{12}\phi_{12} \qquad (B.7) \\ \sigma_{43} &= \beta_{11}\phi_{12} + \beta_{12}\phi_{22} \end{aligned}$$

The parameters ϕ_{11} and ϕ_{12} are immediately identifiable, but ϕ_{22} is inaccessible when $\beta_{22} = 0$. This means that solving two linear equations in two unknowns won't work. The parameter of interest, β_{11}, can only be recovered if $\phi_{12} = 0$ as well as $\beta_{22} = 0$.

The conclusion is that β_{11} is identifiable provided that $\beta_{22} \neq 0$ or $\beta_{22} = \phi_{12} = 0$.

```
# Study the identifiability of beta12
d2 = denominator(sol[beta12]); Simplify(d2(theta))
```

evaluate

$$(\phi_{12}^2 - \phi_{11}\phi_{22})\beta_{22}\phi_{12}$$

It looks like we need both β_{22} and ϕ_{12} non-zero. Earlier, we calculated the covariance matrix $\mathbf{\Sigma}$ with $\phi_{12} = 0$ but not β_{22}. In that case,

$$\beta_{12} = \frac{\sigma_{46}}{\sigma_{36}} = \frac{\beta_{12}\beta_{22}\phi_{22}}{\beta_{22}\phi_{22}}.$$

If β_{22} but not $\phi_{12} = 0$, we are back to the two linear equations (B.7). We can't solve the two equations for β_{11} and β_{12} because ϕ_{22} isn't identifiable. However, we can recover β_{12} if $\beta_{11} = 0$. Okay, so far we have established that β_{12} is identifiable if

- $\beta_{22} \neq 0$, or
- $\beta_{22} = 0$ and $\phi_{12} \neq 0$ and $\beta_{11} = 0$.

Now let's see what happens if both β_{22} and ϕ_{12} equal zero.

```
# If both beta22 and phi12 equal zero,
Sigma(beta22=0,phi12=0)
```

evaluate

$$\left(\begin{array}{rrrrrr}
\omega_1+\phi_{11} & \phi_{11} & 0 & \beta_{11}\phi_{11} & \beta_{11}\phi_{11} & 0 \\
\phi_{11} & \omega_2+\phi_{11} & 0 & \beta_{11}\phi_{11} & \beta_{11}\phi_{11} & 0 \\
0 & 0 & \omega_3+\phi_{22} & \beta_{12}\phi_{22} & \beta_{12}\phi_{22} & 0 \\
\beta_{11}\phi_{11} & \beta_{11}\phi_{11} & \beta_{12}\phi_{22} & \beta_{11}^2\phi_{11}+\beta_{12}^2\phi_{22}+\omega_4+\psi_1 & \beta_{11}^2\phi_{11}+\beta_{12}^2\phi_{22}+\psi_1 & 0 \\
\beta_{11}\phi_{11} & \beta_{11}\phi_{11} & \beta_{12}\phi_{22} & \beta_{11}^2\phi_{11}+\beta_{12}^2\phi_{22}+\psi_1 & \beta_{11}^2\phi_{11}+\beta_{12}^2\phi_{22}+\omega_5+\psi_1 & 0 \\
0 & 0 & 0 & 0 & 0 & \omega_6+\psi_2
\end{array}\right)$$

The sign of β_{12} can be identified but not the value, because ϕ_{22} can't be recovered.

We now have a detailed picture of the identifiability of the key parameters β_{11}, β_{12} and β_{22}, a picture that would be just too much work to obtain without a symbolic math program like Sage. If at this point you are wishing that you didn't know so much about the identifiability of the β_{ij}, think again. For example, it would be natural to try testing $H_0 : \beta_{11} = \beta_{12} = \beta_{22} = 0$ with a likeihood ratio test, but this would be a disaster because the parameters are not identifiable under the null hypothesis.

Next, we will obtain explicit formulas for the model-induced equality constraints on the variances and covariances of the observable data, by substituting solutions for the parameters into the equations that were set aside. Results without an = sign are polynomials implicitly set to zero.

```
for item in aside: factor(item(sol))
```

evaluate

$$\sigma_{14} - \sigma_{15}$$

$$\sigma_{13} - \sigma_{23}$$

$$\sigma_{14} - \sigma_{24}$$

$$\sigma_{14} - \sigma_{25}$$

$$\sigma_{16} - \sigma_{26}$$

$$\sigma_{34} - \sigma_{35}$$

$$\frac{\sigma_{16}\sigma_{34}}{\sigma_{13}} = \sigma_{56}$$

$$\frac{\sigma_{16}\sigma_{34}}{\sigma_{13}} = \sigma_{46}$$

If the last two polynomials are multiplied through by σ_{13}, we get

$$\sigma_{16}\sigma_{34} = \sigma_{13}\sigma_{56} = \sigma_{13}\sigma_{56},$$

which is a nice way to express the constraints because the statement remains true when the denominator $\sigma_{13} = \phi_{12}$ equals zero. This claim is verified by evaluating the σ_{ij} quantities at the model parameters, as follows.

```
# Are constraints still true when sigma13=0?
equal3 = [sigma16*sigma34, sigma13*sigma56, sigma13*sigma46]
for item in equal3: show(item(theta))
```

evaluate

$$(\beta_{11}\phi_{12} + \beta_{12}\phi_{22})\beta_{22}\phi_{12}$$
$$(\beta_{11}\phi_{12} + \beta_{12}\phi_{22})\beta_{22}\phi_{12}$$
$$(\beta_{11}\phi_{12} + \beta_{12}\phi_{22})\beta_{22}\phi_{12}$$

The equality constraints we have worked so hard to obtain can be quite valuable in data analysis. If the model is re-parameterized by making $\psi_2 + \omega_6$ a single parameter, we have 21 covariance structure equations in 13 unknown parameters. The likelihood ratio chi-squared test for goodness of fit will have $21 - 13 = 8$ degrees of freedom, and the null hypothesis is that exactly those eight equality constraints hold. If the model does not fit, the constraints can be tested individually to track down why the model does not fit, and suggest how it might be fixed up.

Groebner Basis While there is no doubt that Sage can make life easier by reducing the computational burden of studying a model, it's still too bad that so much thinking is required. In particular, in order to prove identifiability by obtaining explicit solutions for the parameters, you need to figure out which equations are redundant so you can give `solve` a system that has a general solution. To do this, you almost have to solve the equations by hand, or at least look at them carefully and decide how you would proceed if you were going to do it by hand.

An alternative that sometimes works (but not always) is to apply Groebner basis methods. If you subtract the σ_{ij} from both sides of the covariance structure equations, you get a set of multivariate polynomials, and the roots of those polynomials are the solutions of the equations. A Groebner basis is a set of polynomials having the same roots as the input set, but they are typically much easier to solve. See the documentation for the `GroebnerBasis` function on page 275 for more details.

Input to the `GroebnerBasis` function is a list of polynomials and a list of variables. The polynomials correspond to the covariance structure equations, and are produced as an option by the `SetupEqns` function. The "variables" are the model parameters *and* the σ_{ij} quantities. Ordering of the list of variables is very important. The σ_{ij} come last. The model parameters go before the σ_{ij} quantities, usually in reverse order of how interesting or important they are.

If the σ_{ij} quantities are last in the input list of variables and there are equality constraints among them, the first set of polynomials in the Groebner basis will involve only σ_{ij}s. Setting these to zero gives you the equality constraints. Then come the model parameters. If the first parameter (the last you mentioned in the list of variables) is identifiable it will appear by itself, accompanied only by covariances. If fortune smiles, the next polynomial will involve two model parameters, and so on.

The Groebner basis algorithm simplifies the input by multiplying polynomials together and then adding multiples of polynomials to other polynomials. Depending on the size and structure of the problem, the number of polynomials can become very large before finally reducing to a small set with a nice simple form. As a mathematical certainty, the target (a Groebner basis) exits and algorithm terminates at the right answer, but in practice this may not happen during your lifetime. As I said, Groebner basis does not always work, but when it works it is beautiful. In the following, we will just hand the whole system of Example B.1.2 to the `GroebnerBasis` function, warts and all.

Notice how the list of parameters is reversed, so that the β_{ij} come last and therefore the solutions for those parameters will emerge first. The σ_{ij} quantities are reversed as well. This makes the output of the `GroebnerBasis` function easier to compare with earlier work. I may as well explain why, because it sheds light on how Groebner basis works in practice, as well as features of some other functions in the `sem` package.

The `GroebnerBasis` function requires σ_{ij} quantities as input, and I do not want to type in the names of the 21 unique elements. `Parameters(SymmetricMatrix(6,'sigma'))` does the trick. When I examine a covariance matrix, my preference is to look at the upper triangle, scanning from left to right. For this reason, the `SymmetricMatrix` function, which produces a matrix containing only unique elements, puts copies of the upper triangle into the lower triangle. So, for example, row 4 column 2 contains σ_{24}. In this example, the `Parameters` function detects that the matrix is symmetric, and returns the main diagonal and the upper triangle, from left to right and top to bottom.

When I was deciding which equations to set aside, I followed my usual practice of looking at the upper triangle left to right an top to bottom. If I discovered an equation that was redundant with the earlier ones, I selected it for deletion. When I did this I was just trying to be systematic and not thinking about Groebner basis, but it was fortunate. Groebner basis works from the end of the list of input variables (parameters and σ_{ij} quantities). When a variable appears in the list of output polynomials for the first time, it will tend to appear with variables closer to the end of the list.

With the σ_{ij} reversed as well as at the end, the list of variables looks like $\ldots \sigma_{13}, \sigma_{12}, \sigma_{11}]$. This means, for example, that if $\sigma_{14} = \sigma_{15}$, other polynomials in the output (and the corresponding solutions for the model parameters) will be in terms of σ_{14} rather than σ_{15}. That's exactly the way I did it. It is only because of this happy coincidence that we have a prayer of checking that the Groebner basis results are consistent with what we did before without doing a lot of work.

```
param2 = copy(param) # Work with a copy to avoid changing the original.
param2.reverse() # Reversed order of interest
sigmaij = Parameters(SymmetricMatrix(6,'sigma'))
sigmaij.reverse() # Reverse the sigma_ij too
param2.extend(sigmaij) # Put sigma_ij values at the end
polynoms = SetupEqns(Sigma,poly=True) # Covariance structure polynomials
# Throw the whole thing at GroebnerBasis.
basis1 = GroebnerBasis(polynoms,param2)
```

evaluate

```
Defining tT1, tT2, tT3, tT4, tT5, tT6, tT7, tT8, tT9, tT10, tT11, tT12,
tT13, tT14, tT15, tT16, tT17, tT18, tT19, tT20, tT21, tT22, tT23, tT24,
tT25, tT26, tT27, tT28, tT29, tT30, tT31, tT32, tT33, tT34, tT35
```

To my surprise, it finished almost immediately. Take a look.

```
for item in basis1: show(item)
```

evaluate

$-\sigma_{14}+\sigma_{15}$
$-\sigma_{13}+\sigma_{23}$
$-\sigma_{14}+\sigma_{24}$
$-\sigma_{14}+\sigma_{25}$
$-\sigma_{16}+\sigma_{26}$
$-\sigma_{34}+\sigma_{35}$
$-\sigma_{16}\sigma_{34}+\sigma_{13}\sigma_{46}$
$-\sigma_{46}+\sigma_{56}$
$-\beta_{11}\sigma_{13}\sigma_{16}+\beta_{11}\sigma_{12}\sigma_{36}+\sigma_{16}\sigma_{34}-\sigma_{14}\sigma_{36}$
$\beta_{11}\sigma_{12}+\beta_{12}\sigma_{13}-\sigma_{14}$
$\beta_{12}\sigma_{16}\sigma_{34}+\beta_{11}\sigma_{12}\sigma_{46}-\sigma_{14}\sigma_{46}$
$\beta_{11}\sigma_{16}+\beta_{12}\sigma_{36}-\sigma_{46}$
$\beta_{22}\sigma_{13}-\sigma_{16}$
$\beta_{22}\sigma_{34}-\sigma_{46}$
$\beta_{11}\beta_{22}\sigma_{12}-\beta_{22}\sigma_{14}+\beta_{12}\sigma_{16}$
$\phi_{11}-\sigma_{12}$
$\phi_{12}-\sigma_{13}$
$\phi_{22}\sigma_{16}-\sigma_{13}\sigma_{36}$
$-\sigma_{34}\sigma_{36}+\phi_{22}\sigma_{46}$
$\beta_{11}\phi_{22}\sigma_{12}-\beta_{11}\sigma_{13}^{2}-\phi_{22}\sigma_{14}+\sigma_{13}\sigma_{34}$
$\beta_{12}\phi_{22}+\beta_{11}\sigma_{13}-\sigma_{34}$
$\beta_{22}\phi_{22}-\sigma_{36}$
$\beta_{11}\sigma_{14}+\beta_{12}\sigma_{34}+\psi_{1}-\sigma_{45}$
$\omega_{1}-\sigma_{11}+\sigma_{12}$
$\omega_{2}+\sigma_{12}-\sigma_{22}$
$\omega_{3}+\phi_{22}-\sigma_{33}$
$\omega_{4}-\sigma_{44}+\sigma_{45}$
$\omega_{5}+\sigma_{45}-\sigma_{55}$
$\beta_{22}\sigma_{36}+\omega_{6}+\psi_{2}-\sigma_{66}$

These polynomials have the same roots as the input set. There are more polynomials than in the input set, but not hundreds — something that can easily happen. The first eight polynomials in the Groebner basis involve only σ_{ij} quantities. Comparing them to the constraints we obtained earlier (see page 263), we see that they are exactly the same, except just a little better. The first six constraints are even in the same order. For the last two, the Groebner basis is better because $\frac{\sigma_{16}\sigma_{34}}{\sigma_{13}}=\sigma_{56}$ and $\frac{\sigma_{16}\sigma_{34}}{\sigma_{13}}=\sigma_{46}$ do imply $\sigma_{46}=\sigma_{56}$. A simple equality between covariances is preferable to a product set equal to another product.

The next polynomial involves β_{11}, which appears first because it is the last parameter on the input list. Setting it equal to zero and solving yields the solution on page 258. Next comes not one but three polynomials involving β_{11} and β_{12}. If the solution for β_{11} in terms of σ_{ij} is substituted into the first polynomial yields the solution for β_{12} on page 258. The other two yield alternative solutions for β_{12}; these solutions are also correct. Setting any two of them equal yields a complicated equality constraint on the σ_{ij} — a constraint that causes `solve` to think the whole system has no general solution. There is nothing new though, because these constraints are implied by the constraints located earlier.

The next two polynomials involve β_{22} and σ_{ij} quantities. Setting the first one equal to zero yields the solution for β_{22} on page 258. Setting the second one equal to zero yields the "superior" solution on page 259.

This is the way it goes. There may be multiple ways of solving for a particular parameter in terms of σ_{ij} quantities and parameters that have come before. Because of the way the variables are ordered in this example, the first polynomial involving a particular parameter always corresponds to one of the solutions given on page 258. When more than one way of solving for a parameter is indicated by the Gorebner basis, sometimes one of them is preferable because it's simpler or applies in more of the parameter space; sometimes not. Almost always, the polynomials are simple enough that one can verify the existence of a solution by inspection without actually calculating it.

The last polynomial in the set is $\beta_{22}\sigma_{36}+\omega_6+\psi_2-\sigma_{66}$. This is the first time either ψ_2 or ω_6 appears, and the fact that they appear together tells you they are not identifiable. They come last not because they are non-identifiable, but because one of them, ω_6, is first in the list of variables. The way they appear together as a sum reflects the way they are non-identifiable.

When Groebner basis works, it is hard to exaggerate how excellent it is. Equality constraints involving the σ_{ij} quantities appear immediately without all the hard work, and identifiability or lack of identifiability can usually be verified by inspection. It is really wonderful that the equality constraints implied by models whose parameters are non-identifiable can be so easy to obtain, because it makes these models testable (falsifiable) without finding a way to re-parameterize them in a way that preserves the equality constraints.

But as I have mentioned several times, the Groebner basis approach does not always work. When it fails, it usually fails by not finishing. I have had most trouble with unrestricted factor analysis, and multi-stage models of the *a* influences *b* influences *c* variety — the kind for which identifiablity would be established by the Acyclic Rule. It seems likely that this case could be resolved by ordering the variables better.

B.2.2 Function Documentation

To use the `sem` functions, you must load them once per session.

```
sem = 'http://www.utstat.toronto.edu/~brunner/openSEM/sage/sem.sage'
load(sem)
# load('~/sem.sage') # To load a local version in your home directory
```

evaluate

After the package is loaded, `Contents()` will display a list of the available functions. For help on a particular function, type the function name followed by a question mark, like "`PathVar?`"

1. Matrix Creation
   - 1a) `DiagonalMatrix(size,symbol='psi')`
   - 1b) `GeneralMatrix(nrows,ncols,symbol)`
   - 1c) `IdentityMatrix(size)`
   - 1d) `SymmetricMatrix(size,symbol,corr=False)`
   - 1e) `ZeroMatrix(nrows,ncols)`
2. Covariance Matrix Calculation
   - 2a) `EqsVar(beta,gamma,Phi,oblist,simple=True)`
   - 2b) `FactorAnalysisVar(Lambda,Phi,Omega)`
   - 2c) `NoGammaVar(Beta,Psi)`
   - 2d) `PathVar(Phi,Beta,Gamma,Psi,simple=True)`
   - 2e) `RegressionVar(Phi,Gamma,Psi,simple=True)`
3. Manipulation
   - 3a) `GroebnerBasis(polynomials,variables)`
   - 3b) `LSTarget(M,x,y)`
   - 3c) `Parameters(M)`
   - 3d) `SigmaOfTheta(M,symbol='sigma')`
   - 3e) `Simplify(x)`
4. Utility
   - 4a) `BetaCheck(Beta)`
   - 4b) `Contents()`
   - 4c) `CovCheck(Psi)`
   - 4d) `MultCheck(Beta,Psi)`
   - 4e) `Pad(M)`

For each function, explanation is followed by the function definition (without the documentation string).

1. **Matrix Creation**

   (a) `DiagonalMatrix(size,symbol='psi',double=False)`
   This function creates a diagonal symbolic matrix (size by size) with Greek-letter symbols (default ψ), and single subscripts. Double subscripts are optional. The arguments of the function are

- `size`: Number of rows, equal to number of columns
- `symbol`: A string containing the root. It is usually a Greek letter, but does not have to be. Notice the single quotes in the examples below.
- `double`: Should diagonal elements be doubly subscripted? Default is no, use single subscripts.

Examples:

```
DiagonalMatrix(4) # Will have psi1 to psi4 on main diagonal
DiagonalMatrix(4,double=True) # Will have psi11 to psi44 on main diagon
DiagonalMatrix(2,'phi')
DiagonalMatrix(2,'phi',True)
DiagonalMatrix(size=2,symbol='phi')
```

```
DiagonalMatrix(3,'omega')
```

evaluate

$$\begin{pmatrix} \omega_1 & 0 & 0 \\ 0 & \omega_2 & 0 \\ 0 & 0 & \omega_3 \end{pmatrix}$$

Here is the function definition without the documentation string.

```
def DiagonalMatrix(size,symbol='psi',double=False):
    M = identity_matrix(SR,size) # SR stands for Symbolic Ring
    for i in interval(1,size):
        subscr = str(i)
        if double: subscr = subscr+str(i)
        M[i-1,i-1] = var(symbol+subscr)
    return M
```

(b) `GeneralMatrix(nrows,ncols,symbol)`

This function returns a general symbolic matrix containing symbols with specified root, usually a Greek letter. In each cell of the matrix are the root symbol and subscript(s). The argumments are

- `nrows`: Number of rows
- `ncols`: Number of columns
- `symbol`: A string containing the root. It is usually a Greek letter, but does not have to be. Notice the single quotes in the examples below.

Because it is difficult (impossible?) to get good doubly subscripted variables with the two subscripts separated by a comma, there is potential ambiguity when either `nrows` or `ncols` gets into double figures. What is γ_{111}? Is it $\gamma_{1,11}$ or $\gamma_{11,1}$? For this reason, if either the number of rows or the number of columns exceeds 9, the contents of the matrix returned by this function are singly subscripted.

Examples:

```
GeneralMatrix(6,2,'lambda')
GeneralMatrix(11,3,'L')
GeneralMatrix(3,4,'gamma')
GeneralMatrix(nrows=3,ncols=4,symbol='gamma')
```

```
Gamma = GeneralMatrix(nrows=3,ncols=5,symbol='gamma')
Gamma
```

evaluate

$$\left(\begin{array}{ccccc} \gamma_{11} & \gamma_{12} & \gamma_{13} & \gamma_{14} & \gamma_{15} \\ \gamma_{21} & \gamma_{22} & \gamma_{23} & \gamma_{24} & \gamma_{25} \\ \gamma_{31} & \gamma_{32} & \gamma_{33} & \gamma_{34} & \gamma_{35} \end{array}\right)$$

Here is the function definition without the documentation string.

```
def GeneralMatrix(nrows,ncols,symbol):
    M = matrix(SR,nrows,ncols) # SR is the Symbolic Ring
    if nrows < 10 and ncols < 10:
        for i in interval(1,nrows):
            for j in interval(1,ncols):
                M[i-1,j-1] = var(symbol+str(i)+str(j))
    else:
        index=1
        for i in interval(1,nrows):
            for j in interval(1,ncols):
                M[i-1,j-1] = var(symbol+str(index))
                index = index+1
    return M
```

(c) `IdentityMatrix(size)`
This function returns a symbolic identity matrix of specified size. It's the same as `identity_matrix(SR,size)`.
Example: `IdentityMatrix(3)`
Here is the function definition without the documentation string.

```
def IdentityMatrix(size):
    M = identity_matrix(SR,size) # SR is the Symbolic Ring
    return M
```

(d) `SymmetricMatrix(size,symbol,corr=False)`
This function returns a square symmetric matrix of the symbolic type, containing symbols with a specified root, usually a Greek letter. In each cell of the

matrix is the root symbol with subscript(s). The matrix contains only unique elements; the lower triangle contains symbols from the upper triangle, so that the element in row 5 and column 2 is something like σ_{25}.

The arguments of the function are

- `size`: Number of rows, equal to number of columns
- `symbol`: A string containing the root. It is usually a Greek letter, but does not have to be. Notice the single quotes in the examples below.
- `corr`: A logical variable (True or False) specifying whether it's a correlation matrix. If True, there are ones on the main diagonal. This argument is optional, with a default of False.

Examples:

```
SymmetricMatrix(6,'phi')
SymmetricMatrix(11,'psi')
SymmetricMatrix(4,'rho',True)
SymmetricMatrix(size=4,symbol='rho',corr=True)
```

Because it is difficult or maybe even impossible with Sage to get good doubly subscripted variables with the two subscripts separated by a comma, there is potential ambiguity when either `nrows` or `ncols` gets into double figures. What is σ_{111}? Is it $\sigma_{1,11}$ or $\sigma_{11,1}$? For this reason, if either the number of rows or the number of columns exceeds 9, the contents of the matrix returned by this function are singly subscripted. In this case the diagonal elements are numbered last, which is usually what you want once you get used to it.

```
Phi = SymmetricMatrix(5,'phi'); Phi
```

evaluate

$$\begin{pmatrix} \phi_{11} & \phi_{12} & \phi_{13} & \phi_{14} & \phi_{15} \\ \phi_{12} & \phi_{22} & \phi_{23} & \phi_{24} & \phi_{25} \\ \phi_{13} & \phi_{23} & \phi_{33} & \phi_{34} & \phi_{35} \\ \phi_{14} & \phi_{24} & \phi_{34} & \phi_{44} & \phi_{45} \\ \phi_{15} & \phi_{25} & \phi_{35} & \phi_{45} & \phi_{55} \end{pmatrix}$$

Here is the function definition without the documentation string.

```
def SymmetricMatrix(size,symbol,corr=False):
    M = identity_matrix(SR,size) # SR is the Symbolic Ring
    if size < 10:
        for i in interval(1,size):
            for j in interval(i+1,size):
                M[i-1,j-1] = var(symbol+str(i)+str(j))
                M[j-1,i-1] = M[i-1,j-1]
        if not corr:
            for i in interval(1,size):
                M[i-1,i-1] = var(symbol+str(i)+str(i))
    else:
        index=1
        for i in interval(1,size):
            for j in interval(i+1,size):
                M[i-1,j-1] = var(symbol+str(index))
                M[j-1,i-1] = M[i-1,j-1]
                index = index+1
        if not corr:
            for i in interval(1,size):
                M[i-1,i-1] = var(symbol+str(index))
                index = index+1
    return M
```

(e) `ZeroMatrix(nrows,ncols)`

This function returns a symbolic matrix with specified number of rows and nummber of columns, full of zeros. It's the same as the sage function `matrix(SR,size)`. The `ZeroMatrix` function is particularly useful for setting up parameter matrices that consist mostly of zeros.

Example: `ZeroMatrix(4,4)`

Here is the function definition without the documentation string.

```
def ZeroMatrix(nrowz,ncolz):
    M = matrix(SR,nrowz,ncolz) # SR is the Symbolic Ring
    return M
```

2. **Covariance Matrix Calculation**

(a) `EqsVar(beta,gamma,Phi,oblist,simple=True)`

The `EqsVar` function is alphabetically first in the category of covariance matrix calculation, but it is among the less frequently used. It calculates the covariance matrix of an obvservable data vector for the EQS model of Bentler and Weeks (1980). The EQS model makes no distinction between error terms and other exogenous variables, and there is no notational difference between latent and observable variables. Instead, the covariance matrix of all variables in the model is calculated, and then the rows end columns corresponding to the observable variables are selected to form $\mathbf{\Sigma}$, the common covariance matrix of the n observable data vectors.

The model equations are

$$\boldsymbol{\eta}_i = \boldsymbol{\beta}\boldsymbol{\eta}_i + \boldsymbol{\gamma}\boldsymbol{\xi}_i,$$

with $cov(\boldsymbol{\xi}_i) = \mathbf{\Phi}$.

The exogenous variables (including error terms) are in the vector $\boldsymbol{\xi}_i$, which is spelled "xi" and pronounced more or less like the letter "c." The endogenous variables are in $\boldsymbol{\eta}_i$, which is spelled "eta" and pronounced like "I can't believea I atea the whole thing." Because $\boldsymbol{\xi}_i$ includes error terms as well as ordinary exogenous variables, the `EqsVar` function is useful for calculating the covariance matrix for pathological but disturbingly realistic models in which exogenous variables are correlated with error terms, or measurement errors are correlated with errors in the latent variable model. Other functions in the `sem` package are based on standard models which do not admit this possibility.

In the `EqsVar` function, $\mathbf{V} = cov\begin{pmatrix}\boldsymbol{\eta}_i \\ \boldsymbol{\xi}_i\end{pmatrix}$ is first calculated, and then the covariance matrix $\boldsymbol{\Sigma}$ is formed by selecting rows and columns corresponding to the observable variables.

The indices of the observable variables are given in the function argument `oblist`. The indices start with one, not zero. Following EQS conventions, *the endogenous variables come first in the list of variables* $(\boldsymbol{\eta}_i, \boldsymbol{\xi}_i)$.

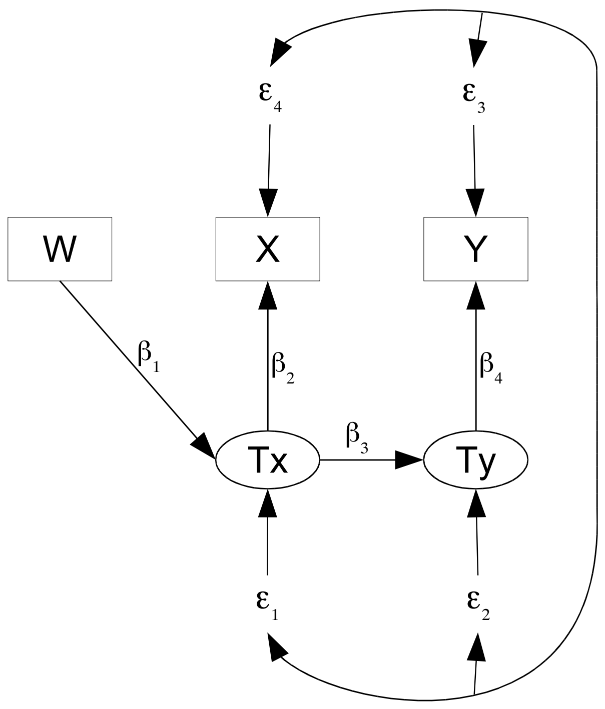
The arguments of the function are

- `beta`: A square matrix containing the coefficients from each element of eta to each other element. Number of rows equals number of columns equals number of exogenous variables, including error terms. Diagonal elements of Beta should be zeros.
- `gamma`: A matrix of regression coefficients linking each exogenous (ξ) variable to each endogenous (η) variable. There is one row for each eta variable and one column for each ξ variable. Thus, the number of rows in `gamma` must equal the number of rows (and columns) in `beta`, and the number of columns in `gamma` must equal the number of rows (and columns) in `Phi`.
- `Phi`: The variance-covariance matrix of the exogenous variables $\boldsymbol{\eta}_i$.
- `oblist`: List of indices of observable variables. First index is one, not zero. May be in any order. Following EQS conventions, the endogenous variables come first in the list of variables $(\boldsymbol{\eta}_i, \boldsymbol{\xi}_i)$. So the variable with index one is the first *endogenous* variable.
- `simple`: Should the covariance matrix be simplified? Simplification consists of expanding and then factoring all the elements of $\boldsymbol{\Sigma}$. This is time consuming, but usually worth it. The default for this optional argument is True.

Example: `EqsVar(beeta,gammma,fee,pickout)`

The following more detailed example is an extension of Example 0.6.1 on page 30, which was about the connection between income and credit card debt among real estate agents. In the path diagram of Figure B.3, X is reported income (Tx measured with error), Y is reported credit card debt (Ty measured with error), and W is local average selling price of a resale home (real estate agents typically get a percentage of the the selling price). Because of numerous omitted variables, the error terms are all correlated with one another.

Figure B.3: Massively correlated error terms



In the notation of the EQS model,

$$\boldsymbol{\xi}_i = \left(\begin{array}{c} \epsilon_{i,1} \\ \epsilon_{2,1} \\ \epsilon_{i,3} \\ \epsilon_{i,4} \\ W_i \end{array} \right) \quad \text{and} \quad \boldsymbol{\eta}_i = \left(\begin{array}{c} Tx_i \\ Ty_i \\ X_i \\ Y_i \end{array} \right)$$

```
# In EsqVar, eta = beta eta + gamma xi,   with cov(xi) = Phi
# eta' = (Tx,Ty,X,Y) and xi' = (epsilon1,epsilon2,epsilon3,epsilon4,W)
B = ZeroMatrix(4,4) # beta
B[1,0] = var('beta3') ; B[2,0] = var('beta2') ; B[3,1] = var('beta4')
G[0,0] = 1; G[0,4] = var('beta1'); G[1,1] = 1; G[2,3] = 1; G[3,2] = 1
P = SymmetricMatrix(5,'psi'); P[4,4]=var('phi') # This is the Phi matrix
# No correlations between W and the errors
for j in interval(0,3):
    P[j,4] = 0
    P[4,j] = 0
P
```

evaluate

$$\begin{pmatrix} \psi_{11} & \psi_{12} & \psi_{13} & \psi_{14} & 0 \\ \psi_{12} & \psi_{22} & \psi_{23} & \psi_{24} & 0 \\ \psi_{13} & \psi_{23} & \psi_{33} & \psi_{34} & 0 \\ \psi_{14} & \psi_{24} & \psi_{34} & \psi_{44} & 0 \\ 0 & 0 & 0 & 0 & \phi \end{pmatrix}$$

```
pickout = 9,3,4 # Indices of observable variables, order eta, xi
Sigma = EqsVar(B,G,P,pickout); Sigma
```

evaluate

$$\left(\begin{array}{rrr} \phi & \beta_1\beta_2\phi & \\ \beta_1\beta_2\phi & \beta_1^2\beta_2^2\phi + \beta_2^2\psi_{11} + 2\,\beta_2\psi_{14} + \psi_{44} & \beta_1^2\beta_2\beta_3\beta_4\phi + \\ \beta_1\beta_3\beta_4\phi & \beta_1^2\beta_2\beta_3\beta_4\phi + \beta_2\beta_3\beta_4\psi_{11} + \beta_2\beta_4\psi_{12} + \beta_3\beta_4\psi_{14} + \beta_2\psi_{13} + \beta_4\psi_{24} + \psi_{34} & \beta_1^2\beta_3^2\beta_4^2\phi + \beta_3^2 \end{array} \right.$$

(b) `FactorAnalysisVar(Lambda,Phi,Omega)`

(c) `NoGammaVar(Beta,Psi)`

(d) `PathVar(Phi,Beta,Gamma,Psi,simple=True)`

(e) `RegressionVar(Phi,Gamma,Psi,simple=True)`

3. **Manipulation**

(a) `GroebnerBasis(polynomials,variables)`

(b) `LSTarget(M,x,y)`

(c) `Parameters(M)`

(d) `SigmaOfTheta(M,symbol='sigma')`

(e) `Simplify(x)`

4. **Utility**

(a) `BetaCheck(Beta)`

(b) `Contents()`

(c) `CovCheck(Psi)`

(d) `MultCheck(Beta,Psi)`

(e) `Pad(M)`

Indices of arrays and vectors in Sage start with zero, which can be a minor irritant to those of us who are used to counting on our fingers. This function returns a "padded" version of a matrix by inserting a row zero and a column zero consisting entirely of zeros. This makes it more convenient to refer to elements of the matrix.

Here is the function definition without the documentation string.

```
def Pad(M):

    """
    Pad by making first row and first col all zeros, so it is

    Argument: A matrix that needs padding
    Result: A padded matrix, with one additional row and one additional
            column.

    Example

    SIGMA = Pad(Sigma)




    """
    nrowz = M.nrows(); nrowz = nrowz+1 # Strange work-around
    ncolz = M.ncols(); ncolz = ncolz+1
    padM = matrix(SR,nrowz,ncolz)
    for i in interval(1,M.nrows()):
        for j in interval(1,M.ncols()):
            padM[i,j] = M[i-1,j-1]
    return padM
```

```
Phi = SymmetricMatrix(5,'phi')
PadPhi = Pad(Phi); PadPhi
```

evaluate

$$\left(\begin{array}{cccccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \phi_{11} & \phi_{12} & \phi_{13} & \phi_{14} & \phi_{15} \\ 0 & \phi_{12} & \phi_{22} & \phi_{23} & \phi_{24} & \phi_{25} \\ 0 & \phi_{13} & \phi_{23} & \phi_{33} & \phi_{34} & \phi_{35} \\ 0 & \phi_{14} & \phi_{24} & \phi_{34} & \phi_{44} & \phi_{45} \\ 0 & \phi_{15} & \phi_{25} & \phi_{35} & \phi_{45} & \phi_{55} \end{array}\right)$$

Here is the function definition without the documentation string.

B.3 Using Sage on your Computer

Sage has a browser interface, which means you interact with it through an ordinary Web browser[11]. This means that the actual Sage software can reside either on your computer or a remote server. In practice, there are three possibilities:

1. You may use Sage free of charge on computers maintained by the Sage development group. To do it this way, go to http://sagenb.com, set up a free account, and start using Sage. This is the easiest way to get started, but be aware that many people may be trying to use the service at the same time. My experience is that performance is sometimes quick and pleasant (for example, during the summer), and sometimes very slow. So this is an excellent way to give Sage a try and it's very handy for occasional use, but depending on it to do homework assignments is a bit risky.

2. You can connect to Sage on a server at your university or organization, provided that someone has gone to the trouble to set it up. If you can use Sage this way, you are fortunate, and you only have some minor font issues to take care of. These are discussed below.

3. You can download and install Sage on your own computer. You still use a Web browser, but the Web server is your own machine, and it serves only you. It's pretty straightforward, but the details depend on your operating system. Some of these details may change, because the Sage developers are constantly working (without payment) to improve the package. They also are responding to the actions of various companies like Apple, Google and Microsoft.

Mac OS and Linux There are two steps. First, go to http://www.sagemath.org, download the software, and install it as usual. As of March 2013, there was almost[12] nothing out of the ordinary for Mac OS, and this appeared to be the case for `linux` as well.

The second step is probably needed if you do *not* already have LaTeX installed, which will be the case for many students. Even if you do have LaTeX installed, the following is very helpful if you plan to use Sage on the servers at http://sagenb.com, even occasionally. Go to

http://www.math.union.edu/ dpvc/jsMath/download/jsMath-fonts.html,

download the `jsMath` fonts, and install them. You should only download one set of fonts. To install, Mac users can open the System folder, open the library sub-folder, and then drag the fonts to the Fonts sub-sub folder. You may need to click "Authenticate" and type your password. A re-start will be required before the new fonts are available.

[11]The Sage website says Mozilla Firefox and Google Chrome are recommended, and if you are a Windows user, you should believe it. In a Mac environment, I have had no trouble with Safari.

[12]Under Mac OS, the "App" version of the software is recommended. It works like any other Mac application. The first time you start it, you might have to wait

Microsoft Windows As mentioned earlier, `Sage` incorporates a number of other open source math programs, and makes them work together using a common interface. This marvelous feat, which is accomplished mostly with Python scripts, depends heavily on features that are part of the `linux` and `unix` operating systems, but are missing from Microsoft Windows. This makes it difficult or perhaps actually impossible to construct a native version of `Sage` for Windows. The current (and possibly final) solution is to run Sage in a *virtual machine* – a set of software instructions that act like a separate computer within Windows. The virtual machine uses the `linux` operating system, and has `Sage` preinstalled. The `http://www.sagemath.org` website calls it the "Sage appliance."

The software that allows the virtual machine to function under Windows is Oracle Corporation's free open-source VirtualBox, and you need to install that first. Start at http://wiki.sagemath.org/SageApplianceInstallation, and follow the directons. You will see that the first step is to download VirtualBox.

Then, go to `http://wiki.sagemath.org/SageAppliance`, and follow the directions there. It is *highly* recommended that you set up a folder for sharing files between Windows and the Sage appliance, because a good way of printing your `Sage` output depends on it. Follow *all* the directions, including the part about resetting the virtual machine.

Now you are ready to use `Sage` and see your output on screen. Printing under Windows is a separate issue, but it's easy once you know how.

Printing Under Windows The virtual machine provided by VirtualBox is incomplete by design; it lacks USB support[13]. So, most printers don't work easily. I know of four ways to print, and I have gotten the first three to work. The fourth way is speculation only and I don't intend to try it. The methods are ordered in terms of my personal preference.

1. In the `Sage` appliance, click on the printer icon or press the right mouse button and choose Print from the resulting menu. The default will be to Save as PDF. To choose the location to save the file, click on File System, then media, then the name of the shared folder[14]. Click Save. In Windows, go to the shared folder and print the pdf file[15]. An advantage of this method is that you don't need to install any fonts, because the `jsMath` fonts are already installed in the `linux` system of the Sage Applicance.

2. For this method, you do need to install the `jsMath` fonts under Windows. Go to

[13]Presumably this is a strategic decision by Oracle Corporation. As of this writing, USB support is available from Oracle as a separate free add-on. It's free to individual users for their personal use, meaning nobody can legally re-sell a virtual machine that includes it without paying Oracle a royalty. Sagemath would give it away and not sell it, but the developers strongly prefer to keep `Sage` fully free under the GNU public license.

[14]You set up the shared folder when you installed the Sage applicance.

[15]When working with Sage in a Windows environment, it may be helpful to keep the shared folder open in Windows Explorer. As soon as you save the file you want to print, you will see it appear in Windows Explorer.

http://www.math.union.edu/ dpvc/jsMath/download/jsMath-fonts.html,

download the jsMath fonts, and install them; A darkness level of 25 is good. To install under Windows 7, I needed to double-click on each font individually and click install. More experienced Windows users may be able to install the fonts some other way. A re-start is required.

Now once the jsMath fonts are installed, note that you can reach the Sage runnning in your virtual machine from Windows. Minimize the browser in the virtual machine, and open Firefox or Chrome under Windows. Go to https://localhost:8000. Now you can do whatever calculations you wish and print as usual. When you are done, you need to close the browser in the Sage appliance as well as Windows, and sent the shutdown signal before closing Virtualbox.

3. When you choose Print from within the Sage appliance, the default is Save as PDF. But because the Web browser in the Sage appliance is Google Chrome, Google Cloud Print is also an option. You can connect your printer to Google Cloud Print provided that Google Chrome is installed under Windows, and you have a Google (gmail) account. Using Chrome, go to http://www.google.com/cloudprint/learn and locate the instructions to set up your printer. If the printer is physically connected to the computer (not wireless), it's called a "classic" printer. Once your printer is connected, you can print to it from the Sage appliance through Google's servers, provided you are connected to the Internet and signed in to your Google account under Windows at the time. There is no need to install any fonts; they are already installed on the virtual linux machine.

4. Finally, in principle one should be able to install the appropriate printer driver (if one exists) in the virtual linux machine and print directly from the Sage appliance. Under Windows, you can access the linux command line using the free open source PuTTy SSH client, which can be obtained from www.putty.org. Once the Sage appliance is running, connect using Host Name localhost through port 2222. The user name is sage and the password is also sage. There may be better ways to reach the linux shell, but this works. You can ignore all the warnings.

   A package containing USB support for VirtualBox is available at https://www.virtualbox.org. Once it's installed, you can start looking for a linux driver for your printer. This printing method is appropriate only for those with linux experience who feel like playing around.