The Stata Journal

The *Stata Journal*, electronic version (ISSN 1536-8734) is a publication of Stata Press, and Stata is a registered trademark of StataCorp LP.

The Stata Journal (2005)
5, Number 3, pp. 421–441

Mata matters: Translating Fortran*

William Gould
StataCorp

Abstract. Mata is Stata's matrix language. In the *Mata matters* column, we show how Mata can be used interactively to solve problems and as a programming language to add new features to Stata. In this column, we demonstrate how Fortran programs can be translated and incorporated into Stata.

Keywords: pr0017, Mata, Fortran

In this column, we are going to translate a program from Fortran to Mata. There is a large legacy of Fortran routines for performing statistics, and much of it is available over the web. It is usually easier to translate such programs into Mata rather than to compile them and write the necessary interface functions to include them as a Stata plugin.

For instance, Fortran algorithm AS 89 (Best and Roberts 1975) is frequently used to calculate the upper-tail probabilities of Spearman's rank-order correlation coefficient. This routine, known as AS 89—AS because it came from the journal *Applied Statistics* published by the Royal Statistical Society—is readily available over the web. The copy below was obtained from Statlib (*http://lib.stat.cmu.edu*), a library of statistical software, data, and information maintained at Carnegie–Mellon University. AS 89 can be found at *http://lib.stat.cmu.edu/apstat/89*:

```
      double precision function prho(n, is, ifault)
c
c        Algorithm AS 89   Appl. Statist. (1975) Vol.24, No. 3, P377.
c
c        To evaluate the probability of obtaining a value greater than or
c        equal to is, where is=(n**3-n)*(1-r)/6, r=Spearman's rho and n
c        must be greater than 1
c
c     Auxiliary function required: ALNORM = algorithm AS66
c
      dimension l(6)
      double precision zero, one, two, b, x, y, z, u, six,
     $  c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12
      data zero, one, two, six /0.0d0, 1.0d0, 2.0d0, 6.0d0/
      data        c1,     c2,     c3,     c4,     c5,     c6,
     $          c7,     c8,     c9,    c10,    c11,    c12/
     $  0.2274d0, 0.2531d0, 0.1745d0, 0.0758d0, 0.1033d0, 0.3932d0,
     $  0.0879d0, 0.0151d0, 0.0072d0, 0.0831d0, 0.0131d0, 0.00046d0/
```

*Fortran in mixed case is now the official name for what was born as FORTRAN in uppercase. Bo Einarsson reports at *http://www.nsc.liu.se/ boein/f77to90/a7.html* that the first Fortran to be spelled in mixed case was Fortran 90.


```
c
c       Test admissibility of arguments and initialize
c
      prho = one
      ifault = 1
      if (n .le. 1) return
      ifault = 0
      if (is .le. 0) return
      prho = zero
      if (is .gt. n * (n * n -1) / 3) return
      js = is
      if (js .ne. 2 * (js / 2)) js = js + 1
      if (n .gt. 6) goto 6
c
c       Exact evaluation of probability
c
      nfac = 1
      do 1 i = 1, n
        nfac = nfac * i
        l(i) = i
    1 continue
      prho = one / dble(nfac)
      if (js .eq. n * (n * n -1) / 3) return
      ifr = 0
      do 5 m = 1,nfac
        ise = 0
        do 2 i = 1, n
          ise = ise + (i - l[i]) ** 2
    2   continue
        if (js .le. ise) ifr = ifr + 1
        n1 = n
    3   mt = l(1)
        nn = n1 - 1
        do 4 i = 1, nn
          l(i) = l(i + 1)
    4   continue
        l(n1) = mt
        if (l(n1) .ne. n1 .or. n1 .eq. 2) goto 5
        n1 = n1 - 1
        if (m .ne. nfac) goto 3
    5 continue
      prho = dble(ifr) / dble(nfac)
      return
c
c       Evaluation by Edgeworth series expansion
c
    6 b = one / dble(n)
      x = (six * (dble(js) - one) * b / (one / (b * b) -one) -
     $  one) * sqrt(one / b - one)
      y = x * x
      u = x * b * (c1 + b * (c2 + c3 * b) + y * (-c4
     $  + b * (c5 + c6 * b) - y * b * (c7 + c8 * b
     $  - y * (c9 - c10 * b + y * b * (c11 - c12 * y)))))
```

```
c
c      Call to algorithm AS 66
c
      prho = u / exp(y / two) + alnorm(x, .true.)
      if (prho .lt. zero) prho = zero
      if (prho .gt. one) prho = one
      return
      end
```

An index of other popular routines can be found at *http://lib.stat.cmu.edu/apstat*.

Step 1: Copy the file

Downloading the above routine resulted in file `prho.f` appearing on my disk. I then copied file `prho.f` to `prho.do`, giving me two copies. My plan will be to convert the Fortran code in file `prho.do` to Mata, and I will keep the original untouched in case I need to review it.

Step 2: Review the copy

At this stage, we do not need to understand the logic of AS 89 and, as a matter of fact, we will never need to understand it. The translation we will perform will be mechanistic, requiring us to understand that original Fortran line

```
ise = ise + (i - l(i)) ** 2
```

means to add a value to `ise`, which value is the square of `(i-l(i))`—meaning `i` minus the `i`th value of vector `l`—but why the original authors choose to increment `ise` in this way we will never need to understand.

Look through the source code to see if you spot anything you mechanically do not understand or if there are any other oddities.

I looked through and did not see anything I did not understand, but I did spot one thing. Near the end are lines that read

```
c
c      Call to algorithm AS 66
c
      prho = u / exp(y / two) + alnorm(x, .true.)
```

`alnorm()` is not a built-in, standard Fortran function. `alnorm()` is, in fact, another AS routine, so either I am going to have to obtain and to translate it, too, or I am going to have to find out what it does and substitute another, already existing Mata function for it.

Back to the web site I went. I found the code for AS 66 and the top read,

```
c This file includes the Applied Statistics algorithm AS 66 for calculating
c the tail area under the normal curve, and two alternative routines which
c give higher accuracy.   The latter have been contributed by Alan Miller of
c CSIRO Division of Mathematics & Statistics, Clayton, Victoria.   Notice
c that each function or routine has different call arguments.
c
c
      double precision function alnorm(x,upper)
c
c         Algorithm AS66 Applied Statistics (1973) vol22 no.3
c
c       Evaluates the tail area of the standardised normal curve
c       from x to infinity if upper is .true. or
c       from minus infinity to x if upper is .false.
```

The call to `alnorm()` that appears in AS 89 reads `alnorm(x, .true.)`. From the documentation of `alnorm()`, we now know that it calculates the normal curve from `x` to infinity. The Mata function to calculate normal areas is `normal(x)`, and it calculates the area from $-\infty$ to `x`. We will want $1 -$ `normal(x)` or, better, `normal(-x)`, because that avoids the subtraction, so it will be more accurate. Thus I made a note to myself that later I would translate

`alnorm(x, .true.)` $\iff$ `normal(-x)`

Looking through the program again, I also noted the line

```
data zero, one, two, six /0.0d0, 1.0d0, 2.0d0, 6.0d0/
```

That is AS style; rather than using constants `0`, `1`, `2`, and `6` in the program, they used `zero`, `one`, `two`, and `six`. In Mata, there is no reason to do this and adding the extra variables just makes the code more difficult to read. So using my editor, I ran through file `prho.do` and changed `zero` to `0`, `one` to `1`, and so on, and finally I deleted the line

```
data zero, one, two, six /0.0d0, 1.0d0, 2.0d0, 6.0d0/
```

There was another data line below that,

```
data        c1,     c2,     c3,     c4,     c5,     c6,
$          c7,     c8,     c9,    c10,    c11,    c12/
$  0.2274d0, 0.2531d0, 0.1745d0, 0.0758d0, 0.1033d0, 0.3932d0,
$  0.0879d0, 0.0151d0, 0.0072d0, 0.0831d0, 0.0131d0, 0.00046d0/
```

but that one I left because I did not think the resulting code would be more readable if I substituted numbers for the variables. I knew that later I could translate the line to read

```
c1 = .2274
c2 = .2531
...
c12 = .00046
```

Step 3: Translate

Step 3.1 The opening

The most difficult portion of a Fortran program to translate is the opening

```
      double precision function prho(n, is, ifault)
      (comments omitted)
      dimension l(6)
      double precision zero, one, two, b, x, y, z, u, six,
     $  c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12
```

This becomes

```
real scalar prho(real scalar n, real scalar is, real scalar ifault)
{
        real scalar     prho
        real vector     l
        real scalar     b, x, y, z, u
        real scalar     c1, c2, c3, c4, c5, c6, c7, c8, c9, c10,
                        c11, c12
        l = J(1, 6, .)
```

The first thing to notice is that I added

```
        real scalar     prho
```

even though nothing similar appeared in the original Fortran code. I did that because I know something about Fortran. Fortran functions return the value that happens to be in a variable of the same name as the function. That variable is undeclared in Fortran.

The other declarations are translated to `real scalar` if they are scalars or to `real vector` if they are vectors or to `real matrix` if they are matrices. In the case of vectors and matrices, you do not specify the extent of their dimensions in Mata. If, in Fortran, `mymat` is declared to be 3×2, you just translate that as `real matrix mymat`. Then later, after all the declarations, you fill in `mymat` with a 3×2 matrix, the values of which do not matter. The easy way to do that is via Mata's `J()` function, such as `mymat = J(3,2,.)`.

In AS 89, `l` was declared to be a vector of length 6: `dimension l(6)`. In translation, that resulted in the declaration `real vector l` and then, after all the declarations, I added the line,

```
        l = J(1, 6, .)
```

I filled `l` in with missing values, but that was irrelevant. I could just as well have coded

```
        l = J(1, 6, 0)
```

Step 3.2: Initializations

Fortran programs can include statements of the form

```
      data        c1,     c2,     c3,     c4,     c5,     c6,
     $          c7,     c8,     c9,    c10,    c11,    c12/
     $  0.2274d0, 0.2531d0, 0.1745d0, 0.0758d0, 0.1033d0, 0.3932d0,
     $  0.0879d0, 0.0151d0, 0.0072d0, 0.0831d0, 0.0131d0, 0.00046d0/
```

which preloads `c1`, `c2`, ..., with the specified values. Mata has no equivalent. AS 89 has that line. Translate this as

```
c1  = 0.2274d0
c2  = 0.2531d0
c3  = 0.1745d0
c4  = 0.0758d0
c5  = 0.1033d0
c6  = 0.3932d0
c7  = 0.0879d0
c8  = 0.0151d0
c9  = 0.0072d0
c10 = 0.0831d0
c11 = 0.0131d0
c12 = 0.00046d0
```

The line for `c1` could just as well read

```
c1  = 0.2274e0
```

or

```
c1  = 0.2274
```

Mata understands Fortran's `d0` suffix, so you do not have to change or remove it.

Step 3.3: The body

The first lines of the body of our Fortran program read

```
c
c        Test admissibility of arguments and initialize
c
      prho = one
      ifault = 1
      if (n .le. 1) return
      ifault = 0
      if (is .le. 0) return
      prho = zero
      if (is .gt. n * (n * n -1) / 3) return
      js = is
      if (js .ne. 2 * (js / 2)) js = js + 1
      if (n .gt. 6) goto 6
```

and these become

```
//
//      Test admissibility of arguments and initialize
//
        my_is = trunc(is)
        prho = 1
        ifault = 1
        if (n < 1) return(prho)
        ifault = 0
        if (my_is <= 0) return(prho)
        prho = 0
        if (my_is >  n * (n * n -1) / 3) return(prho)
        js = my_is
        if (js != 2 * (js / 2)) js = js + 1
        if (n > 6) goto L6
```

First, I changed all the Fortran `RETURN`s to `return(prho)`. As previously mentioned, Fortran functions return the value that is in the variable of the same name. Mata requires that you specify the value to be returned. I was momentarily tempted to make a more thorough translation

```
//
//      Test admissibility of arguments and initialize
//
        my_is = trunc(is)
        ifault = 1
        if (n < 1) return(1)
        ifault = 0
        if (my_is <= 0) return(1)
        if (my_is >  n * (n * n -1) / 3) return(0)
        js = my_is
        if (js != 2 * (js / 2)) js = js + 1
        if (n > 6) goto L6
```

but I decided against that. In the above, I got rid of `prho` and made the `return()` lines clearly specify exactly what was being returned. The code looks better and is easier to understand, but if I am going to make such a thorough translation, I am going to need to be cautious. At some later point in the code, it might be assumed the variable `prho` contains 1, or contains 0, and I may not notice that hidden assumption. It is safer to translate line by line, so we will return to the original translation:

```
//
//      Test admissibility of arguments and initialize
//
        my_is = trunc(is)
        prho = 1
        ifault = 1
        if (n < 1) return(prho)
        ifault = 0
        if (my_is <= 0) return(prho)
        prho = 0
        if (my_is >  n * (n * n -1) / 3) return(prho)
        js = my_is
        if (js != 2 * (js / 2)) js = js + 1
        if (n > 6) goto L6
```

There is one more translation I made that is deserving of comment. I added the line

```
my_is = trunc(is)
```

at the top of the block, and then, throughout all the rest of the Fortran code, I changed `is` to `my_is`. It turned out that there were only three such places, and all are in the above block.

Fortran variables that begin with the letters `i` through `n` are assumed to be integers if they are not explicitly declared otherwise. Thus, variable `is` is a Fortran `INTEGER` variable. In our translation, however, variable `is` is a real scalar because Mata does not have an integer type. Variable `is` will be an integer if the caller happens to pass an integer value, but otherwise, it is unconstrained.

My solution to this was to create a new variable `my_is` and to guarantee that it is an integer. I did not really need a new variable; I could have just added

```
is = trunc(is)
```

Doing that, however, is not recommended because that would change the caller's value of the argument as well.

Presumably, I should do the same thing with variable `n`, but I decided that it was unlikely the caller (who will be me) will call the program with a noninteger number of observations, and so ignored the issue.

The opening block is now translated; let's translate the next block

```
c
c        Exact evaluation of probability
c
      nfac = 1
      do 1 i = 1, n
        nfac = nfac * i
        l(i) = i
    1 continue
```

This becomes

```
nfac = 1
for (i=1; i<=n; i++) {
        nfac = nfac * i
        l[i] = i
}
```

although it just as well could have become

```
nfac = 1
for (i=1; i<=n; i++) {
        nfac = nfac * i
        l[i] = i
L1:  }                          (notice the tag L1)
```

The `1 continue` statement in the original Fortran source was included only to close the `DO` loop. I looked ahead and verified that there was no `GOTO 1` in the rest of the Fortran

program, so I do not need to label the line, although I could have and it would not have mattered. We will talk a little more about labels later.

Do loops are always translated the same way:

```
      DO # var = a, b
      ...
    # CONTINUE
```

translate as

```
for (var=a; var<=b; var++) {
        ...
}
```

I also had to remember to change the subscripting parentheses in the Fortran line

```
l(i) = i
```

to square brackets, which Mata requires:

```
l[i] = i
```

The next bit of Fortran code reads

```
      prho = one / dble(nfac)
      if (js .eq. n * (n * n -1) / 3) return
      ifr = 0
      do 5 m = 1,nfac
        ise = 0
        do 2 i = 1, n
          ise = ise + (i - l(i)) ** 2
2     continue
      if (js .le. ise) ifr = ifr + 1
      n1 = n
3     mt = l(1)
      nn = n1 - 1
      do 4 i = 1, nn
        l(i) = l(i + 1)
4     continue
      l(n1) = mt
      if (l(n1) .ne. n1 .or. n1 .eq. 2) goto 5
      n1 = n1 - 1
      if (m .ne. nfac) goto 3
5   continue
      prho = dble(ifr) / dble(nfac)
      return
```

(*Continued on next page*)

and this becomes

```
prho = 1 / nfac
if (js ==  n * (n * n -1) / 3) return(prho)
ifr = 0
for (m=1; m<=nfac; m++) {
        ise = 0
        for (i=1; i<=n; i++) {
                ise = ise + (i - l[i]) ^ 2
        }
        if (js <= ise) ifr = ifr + 1
        n1 = n
   L3:  mt = l[1]
        nn = n1 - 1
        for (i=1; i<=nn; i++) {
                l[i] = l[i + 1]
        }
        l[n1] = mt
        if (l[n1] != n1 | n1 == 2) goto L5
        n1 = n1 - 1
        if (m != nfac) goto L3
   L5:
}
prho = ifr / nfac
return(prho)
```

Note that in translating the next-to-last line,

```
prho = dble(ifr) / dble(nfac)
```

I omitted Fortran's `DBLE()` function:

```
prho = ifr / nfac
```

`DBLE()` is unnecessary in Mata because all variables are double. On the other hand, if you ever see a Fortran statement, such as

```
K = N / 2
```

remember to translate it as

```
k = trunc(n/2)
```

Mata's `trunc()` function is equivalent to how Fortran does integer arithmetic.

The block of code we just translated had many line numbers, such as

```
3   mt = l(1)
```

and

```
if (m .ne. nfac) goto 3
```

which translated as

```
   L3:  mt = l[1]
```

and

```
            if (m != nfac) goto L3
```

Line numbers in Fortran are just that, numbers. Line numbers in Mata must begin with a letter or underscore and need not be numeric at all. When translating Fortran line numbers, I add an uppercase `L` to the original number.

The final block of Fortran code reads

```
c
c        Evaluation by Edgeworth series expansion
c
    6 b = one / dble(n)
      x = (six * (dble(js) - one) * b / (one / (b * b) -one) -
     $  one) * sqrt(one / b - one)
      y = x * x
      u = x * b * (c1 + b * (c2 + c3 * b) + y * (-c4
     $  + b * (c5 + c6 * b) - y * b * (c7 + c8 * b
     $  - y * (c9 - c10 * b + y * b * (c11 - c12 * y)))))
c
c      Call to algorithm AS 66
c
      prho = u / exp(y / two) + alnorm(x, .true.)
      if (prho .lt. zero) prho = zero
      if (prho .gt. one) prho = one
      return
      end
```

and this becomes

```
        //
        //       Evaluation by Edgeworth series expansion
        //
            L6: b = 1 / n
                x = (6 * (js - 1) * b / (1 / (b * b) -1) - 1) * sqrt(1 / b - 1)
                y = x * x
                u = x * b * (c1 + b * (c2 + c3 * b) + y * (-c4
                        + b * (c5 + c6 * b) - y * b * (c7 + c8 * b
                        - y * (c9 - c10 * b + y * b * (c11 - c12 * y)))))
                prho = u / exp(y / 2) + normal(-x)
                if (prho < 0) prho = 0
                if (prho > 1) prho = 1
                return(prho)
        }
```

The last bit of translation was pretty simple. Notice, however, the original line

```
      u = x * b * (c1 + b * (c2 + c3 * b) + y * (-c4
     $  + b * (c5 + c6 * b) - y * b * (c7 + c8 * b
     $  - y * (c9 - c10 * b + y * b * (c11 - c12 * y)))))
```

which became

```
                u = x * b * (c1 + b * (c2 + c3 * b) + y * (-c4
                        + b * (c5 + c6 * b) - y * b * (c7 + c8 * b
                        - y * (c9 - c10 * b + y * b * (c11 - c12 * y)))))
```

In Fortran, a character in column 6 indicates a continuation line. The authors of AS 89 used `$`. In Mata, a line may continue across physical lines, and there is nothing special you need to do except to ensure that wherever the line is broken, it is obvious that it is incomplete. It was obvious in this case because of the pending close parentheses.

At this point, we have the following completed program

```
real scalar prho(real scalar n, real scalar is, real scalar ifault)
{
        real scalar     prho
        real vector     l
        real scalar     b, x, y, z, u
        real scalar     c1, c2, c3, c4, c5, c6, c7, c8, c9, c10,
                        c11, c12

        l = J(1, 6, .)

        c1  = 0.2274d0
        c2  = 0.2531d0
        c3  = 0.1745d0
        c4  = 0.0758d0
        c5  = 0.1033d0
        c6  = 0.3932d0
        c7  = 0.0879d0
        c8  = 0.0151d0
        c9  = 0.0072d0
        c10 = 0.0831d0
        c11 = 0.0131d0
        c12 = 0.00046d0

//
//      Test admissibility of arguments and initialize
//
        my_is = trunc(is)
        prho = 1
        ifault = 1
        if (n < 1) return(prho)
        ifault = 0
        if (my_is <= 0) return(prho)
        prho = 0
        if (my_is >  n * (n * n -1) / 3) return(prho)
        js = my_is
        if (js != 2 * (js / 2)) js = js + 1
        if (n > 6) goto L6

        nfac = 1
        for (i=1; i<=n; i++) {
                nfac = nfac * i
                l[i] = i
        }
```

```
            prho = 1 / nfac
            if (js == n * (n * n -1) / 3) return(prho)
            ifr = 0
            for (m=1; m<=nfac; m++) {
                    ise = 0
                    for (i=1; i<=n; i++) {
                            ise = ise + (i - l[i]) ^ 2
                    }
                    if (js <= ise) ifr = ifr + 1
                    n1 = n
                  L3:  mt = l[1]
                    nn = n1 - 1
                    for (i=1; i<=nn; i++) {
                            l[i] = l[i + 1]
                    }
                    l[n1] = mt
                    if (l[n1] != n1 | n1 == 2) goto L5
                    n1 = n1 - 1
                    if (m != nfac) goto L3
               L5:
            }
            prho = ifr / nfac
            return(prho)
    //
    //       Evaluation by Edgeworth series expansion
    //
        L6: b = 1 / n
            x = (6 * (js - 1) * b / (1 / (b * b) -1) - 1) * sqrt(1 / b - 1)
            y = x * x
            u = x * b * (c1 + b * (c2 + c3 * b) + y * (-c4
                    + b * (c5 + c6 * b) - y * b * (c7 + c8 * b
                    - y * (c9 - c10 * b + y * b * (c11 - c12 * y)))))

            prho = u / exp(y / 2) + normal(-x)
            if (prho < 0) prho = 0
            if (prho > 1) prho = 1
            return(prho)
    }
```

I have neatly indented the translation to reveal nesting level, but that was not necessary.

Step 4. First compilations

We now modify the `prho.do` file so that Stata can execute it:

top: prho.do

```
clear
mata:

real scalar prho(real scalar n, real scalar is, real scalar ifault)
{
        (rest of program appears here)
}
end
```

end: prho.do

I do not yet bother adding lines to the do-file to test the program. Just passing the program though Mata will cause Mata to compile it, and in the process, I expect the compiler to reveal problems I will need to solve first.

The result of executing the do-file is

```
. do prho
  (output omitted)
note:  variable z unused
  (output omitted)

end of do-file
```

Mata noted that a variable in our program was unused. What that means is that we declared the variable early in our program, in particular in the line

```
real scalar     b, x, y, z, u
```

and, after that, we never used `z` again. Mata does not consider that an error, but it considers the fact worthy of mention.

I went back to the original, untranslated, untouched Fortran code in file `prho.f`, and I discovered that, indeed, variable `z` was declared but never used by the original authors. I removed `z` from the declaration line in the translation,

```
real scalar     b, x, y, u
```

and ran the do-file again. That got rid of the warning message.

Then I went back to the `prho.do` file and added a line `mata set matastrict on`:

top: prho.do

```
clear
mata:
mata set matastrict on              (new)
real scalar prho(real scalar n, real scalar is, real scalar ifault)
{
        (rest of program appears here)
}
end
```

end: prho.do

`mata set matastrict on` makes Mata far more demanding in terms of program construction. I am looking for translation errors and using the compiler to help me. I ran the do-file again, and this time, I got lots of error messages:

```
. do prho
  (output omitted)
variable my_is undeclared
variable js undeclared
variable nfac undeclared
variable i undeclared
variable ifr undeclared
variable m undeclared
variable ise undeclared
variable n1 undeclared
variable mt undeclared
variable nn undeclared
r(3000);

end of do-file
r(3000);
```

I was not much concerned because I know that Fortran programmers often leave variables undeclared. They are especially likely to do this with integer variables, such as `i`, `j`, ..., `n`. My concern is that I mistyped a variable and this inadvertently introduced a new one. To guard against that, I will attempt to explain to myself each of these undeclared variables.

Variable `my_is` is the new variable I introduced, and I should have declared it as a real scalar.

To check the remaining variables, I verified that each appeared in the Fortran stored in file `prho.f`. Each was. I then added the line

```
real scalar        my_is, js, nfac, i, ifr, m, ise, n1, mt, nn
```

to the other declarations in the translation. This time, when I ran file `prho.do`, there were no errors, even with `matastrict` set on.

Step 5. First executions

Function `prho(`n`,` i_s`,` *ifault*`)`, to paraphrase the comment at the top of the original Fortran code, returns the probability of obtaining a value greater than or equal to i_s, where $i_s = (n^3 - n)(1 - r)/6$ and where r is Spearman's rho and n the number of observations. Said differently, it returns $P(I_s \geq i_s \,|\, \rho_s = 0)$.

I want to use `prho()` to calculate the significance of r, that is, the probability of observing a correlation more extreme than r, which is to say,

$$p = P(R_s \leq -\text{abs}(r) \,|\, \rho_s = 0) \;+\; P(R_s \geq \text{abs}(r) \,|\, \rho_s = 0)$$

The probability of $R_s \leq -\text{abs}(r)$ is `prho(`$n, (n^3 - n)(1 + \text{abs}(r))/6,$ *ifault*`)`.

The probability of $R_s \geq \text{abs}(r)$ is $1 -$ `prho(`$n, (n^3 - n)(1 - \text{abs}(r))/6,$ *ifault*`)`.

The term *ifault* in `prho()` is how AS routines flag errors. If all goes well, *ifault* is returned containing 0. If there are problems, *ifault* is set to 1.

Thus before trying `prho()`, I will write another function called `sigrho()`:

```
real scalar sigrho(real scalar n, real scalar r)
{
        real scalar     res1, res2, ifault
        res1 = prho(n, (n^3-n)*(1+abs(r))/6, ifault)
        if (ifault) return(.)
        res2 = 1 - prho(n, (n^3-n)*(1-abs(r))/6, ifault)
        if (ifault) return(.)
        return(res1+res2)
}
```

My `prho.do` file now looks like this:

top: prho.do

```
clear
mata:
mata set matastrict on
real scalar prho(real scalar n, real scalar is, real scalar ifault)
{
        (rest of program appears here)
}
real scalar sigrho(real scalar n, real scalar r)
{
        (rest of program appears here)
}
end
```

end: prho.do

It does not matter that I put `sigrho()` last; I could have put it first.

Now I can try my program:

```
. spearman weight rep78

 Number of obs =      69
Spearman's rho =      -0.4138

Test of Ho: weight and rep78 are independent
    Prob > |t| =       0.0004

. mata: sigrho(69, -.4138)
  .0004569269
```

The significance reported by Stata's `spearman` command is based on an approximation formula. The significance calculated by `sigrho()` is exact.

Step 6. Packaging

Really, I am done. I have translated the Fortran program, and it appears to work. If I just needed a few results from it, I would stop.

It would be more useful in the future, however, if I had a new command, say, `spearman2`, that followed the syntax of `spearman` and also reported the results of exact calculation. That would not be a difficult Stata program to write, because I could use existing Stata command `spearman` to calculate rho and then use my new Mata function `sigrho()` to calculate its significance:

```
program spearman2
        version 9
        syntax varlist(min=2 max=2) [if] [in]
        spearman `varlist' `if' `in'
        mata: my_sigrho(`r(N)', `r(rho)')
        display as txt "      New test = " as res %12.4f r(p2)
end
```

In the above Stata program, I assume that new function `my_sigrho()` is virtually identical to `sigrho()`, except that it returns the result in `r(p2)`, where my Stata program can easily access it.

Mata function `my_sigrho()` is easy to write,

```
void my_sigrho(real scalar n, real scalar r)
{
        st_numscalar("r(p2)", sigrho(n, r))
}
```

Finally, I can put all of this together as an ado-file:

top: spearman2.ado

```
*! version 1.0.0  16aug2005
program spearman2
        version 9
        syntax varlist(min=2 max=2) [if] [in]
        spearman `varlist' `if' `in'
        mata: my_sigrho(`r(N)', `r(rho)')
        display as txt "      New test = " as res %12.4f r(p2)
end
version 9
mata:
mata set matastrict on
void my_sigrho(real scalar n, real scalar r)
{
        st_numscalar("r(p2)", sigrho(n, r))
}
real scalar prho(real scalar n, real scalar is, real scalar ifault)
{
        (rest of program appears here)
}
real scalar sigrho(real scalar n, real scalar r)
{
        (rest of program appears here)
}
end
```

end: spearman2.ado

With this new ado-file, I can type

```
. sysuse auto, clear
(1978 Automobile Data)
. spearman2 weight rep78
 Number of obs =      69
Spearman's rho =      -0.4138
Test of Ho: weight and rep78 are independent
    Prob > |t| =       0.0004
      New test =       0.0005
```

Final result

File `spearman2.ado` reads

top: spearman2.ado

```
*! version 1.0.0  16aug2005
program spearman2
        version 9
        syntax varlist(min=2 max=2) [if] [in]
        spearman `varlist' `if' `in'
        mata: my_sigrho(`r(N)', `r(rho)')
        display as txt "      New test = " as res %12.4f r(p2)
end
version 9
mata:
mata set matastrict on
void my_sigrho(real scalar n, real scalar r)
{
        st_numscalar("r(p2)", sigrho(n, r))
}
real scalar prho(real scalar n, real scalar is, real scalar ifault)
{
        real scalar     prho
        real vector     l
        real scalar     b, x, y, u
        real scalar     c1, c2, c3, c4, c5, c6, c7, c8, c9, c10,
                        c11, c12
        real scalar     my_is, js, nfac, i, ifr, m, ise, n1, mt, nn
        l = J(1, 6, .)
        c1  = 0.2274d0
        c2  = 0.2531d0
        c3  = 0.1745d0
        c4  = 0.0758d0
        c5  = 0.1033d0
        c6  = 0.3932d0
        c7  = 0.0879d0
        c8  = 0.0151d0
        c9  = 0.0072d0
        c10 = 0.0831d0
        c11 = 0.0131d0
        c12 = 0.00046d0
```

```
//
//      Test admissibility of arguments and initialize
//
        my_is = trunc(is)
        prho = 1
        ifault = 1
        if (n < 1) return(prho)
        ifault = 0
        if (my_is <= 0) return(prho)
        prho = 0
        if (my_is >  n * (n * n -1) / 3) return(prho)
        js = my_is
        if (js != 2 * (js / 2)) js = js + 1
        if (n > 6) goto L6
        nfac = 1
        for (i=1; i<=n; i++) {
                nfac = nfac * i
                l[i] = i
        }
        prho = 1 / nfac
        if (js == n * (n * n -1) / 3) return(prho)
        ifr = 0
        for (m=1; m<=nfac; m++) {
                ise = 0
                for (i=1; i<=n; i++) {
                        ise = ise + (i - l[i]) ^ 2
                }
                if (js <= ise) ifr = ifr + 1
                n1 = n
           L3:  mt = l[1]
                nn = n1 - 1
                for (i=1; i<=nn; i++) {
                        l[i] = l[i + 1]
                }
                l[n1] = mt
                if (l[n1] != n1 | n1 == 2) goto L5
                n1 = n1 - 1
                if (m != nfac) goto L3
           L5:
        }
        prho = ifr / nfac
        return(prho)
//
//       Evaluation by Edgeworth series expansion
//
    L6: b = 1 / n
        x = (6 * (js - 1) * b / (1 / (b * b) -1) - 1) * sqrt(1 / b - 1)
        y = x * x
        u = x * b * (c1 + b * (c2 + c3 * b) + y * (-c4
                + b * (c5 + c6 * b) - y * b * (c7 + c8 * b
                - y * (c9 - c10 * b + y * b * (c11 - c12 * y)))))

        prho = u / exp(y / 2) + normal(-x)
        if (prho < 0) prho = 0
        if (prho > 1) prho = 1
        return(prho)
}
```

```
real scalar sigrho(real scalar n, real scalar r)
{
        real scalar     res1, res2, ifault

        res1 = prho(n, (n^3-n)*(1+abs(r))/6, ifault)
        if (ifault) return(.)
        res2 = 1 - prho(n, (n^3-n)*(1-abs(r))/6, ifault)
        if (ifault) return(.)
        return(res1+res2)
}
end
```

end: spearman2.ado

Statistical commentary

`spearman2`—AS 89—performs an exact calculation only when $n \leq 6$. For $n > 6$, it uses an approximation formula. I ran simulations under the null hypothesis $\rho_s = 0$ to check coverage for 5% hypotheses tests:

n	replications	seed	fraction
50	10,000	651201	.0490
7	10,000	109399	.0499
6	10,000	487388	.0583
5	10,000	297777	.0166
4	10,000	121987	.0808
3	10,000	893351	.0000

The last column reports the fraction of tests for which `spearman2` returned `r(p2)` $\leq .05$. Coverage is correct when fraction is .05. The simulations were performed by the do-file

top: simul.do

```
clear
program mysim
args n
drop _all
set obs `n'
gen u1 = uniform()
gen u2 = uniform()
spearman2 u1 u2
end
set seed 893351
simulate r(p2), reps(10000) nodots: mysim 6
count if _sim_1<=.05
count if _sim_1<.
```

end: simul.do

where the number following `set seed` and the number following `mysim` on the `simulate` command were substituted from the table above.

1 References

Best, D. J. and D. E. Roberts. 1975. Algorithm AS 89: The upper tail probabilities of Spearman's rho. *Applied Statistics* 24(3): 377–379.

About the Author

William Gould is President of StataCorp, chief developer of Stata, and principal architect of Mata.