The Stata Journal (2009)
9, Number 1, pp. 137–157

Speaking Stata: Rowwise

Nicholas J. Cox
Department of Geography
Durham University
Durham City, UK
n.j.cox@durham.ac.uk

Abstract. Stata's main data model treats observations in rows and variables in columns quite differently, but rowwise problems also arise that require working against the grain. This column shows how to exploit existing functions and `egen` functions when they exist and apply to such problems. It offers advice on how to build your own loops, `egen` functions, or programs when needed. Mata provides especially convenient tools for constructing many such functions and programs, centered on putting selected data into matrices and then processing each observation as a separate vector. Two programs, `rowsort` and `rowranks`, are formally published with this column.

Keywords: pr0046, rowsort, rowranks, rows, functions, loops, egen, Mata, minimum, maximum, median, any, all, distinct values, sorting, ranking

1 Introduction

Stata's main data model is asymmetric. Your datasets consist of tables of data, but Stata regards the rows (observations) and columns (variables) of those tables differently. There is a command to `summarize` variables, for example, but no exact equivalent to summarize observations. Most of the time, this will not bother you in the slightest. You do not want to average a patient's age and systolic blood pressure, or the number of employees and the income of various firms in a given year. But you might well want to summarize across variables that contain systolic and diastolic blood pressures, or sales in different quarters. In these and similar problems, the impulse is to work against the grain, or rowwise.

This column takes working rowwise as a theme, building upward from ideas that should already be familiar.

Sometimes, the best way to work rowwise is direct, even though it may take a little writing of code or learning of unfamiliar parts of Stata to get where you want to be. That is the main focus of this column.

At other times, the best way to work is indirect: the easiest way to proceed can be to work with a different data structure, at least temporarily. The answer then is (very occasionally) to transpose with `xpose` or (much more commonly) to reshape the data with `reshape`. See the help files or manual entries for more information. A different data structure is especially likely to be a good idea if you have panel or longitudinal data in what Stata calls a wide structure. Stata has a marked preference for handling

such data in a long structure. Data restructuring is often easy, but it may raise a variety of questions that lie beyond the story here.

Operations for data that are essentially matrices in which rows and columns are of similar kind (counts, similarities, flows, interactions, and so forth) also lie beyond the scope of this column.

2 Operators and functions

Some rowwise procedures are so easy that you will not even think of them as working against the grain. You can combine variables easily as sums, differences, products, ratios, and more complicated expressions by using appropriate operators and functions. Getting the difference between or the average of systolic and diastolic blood pressure is child's play:

```
. generate diffbp = systolic - diastolic
. generate avebp = (systolic + diastolic)/2
```

Four important functions that can be used to work rowwise are `max()`, `min()`, `inlist()`, and `missing()`. (The last has a synonym, `mi()`.) If you feed a list of variable names (or other appropriate arguments) separated by commas to `max()`, `min()`, `inlist()`, or `missing()`, the result is determined rowwise, that is, separately for each observation.

2.1 Observation minimums and maximums

Consider a simple statistical problem in which we will find good use for the functions `min()` and `max()`. We will use simulation to look at the variation of sample extremes drawn from a normal, or Gaussian, distribution. For reproducibility, we will set the seed explicitly before firing up the random-normal generator.

```
. clear
. set seed 2803
. set obs 100
. forvalues j = 1/100 {
  2. generate x`j´ = rnormal()
  3. }
```

The loop enclosed by `forvalues` and its braces generates 100 new variables, named `x1` to `x100`. Using a loop clearly beats typing out 100 separate `generate` commands. If you want to know more about using `forvalues` for looping, check out [P] **forvalues** or the tutorial in Cox (2002a). `j` is called a local macro; I will say more about that in a moment. If this is the first time you have heard of a macro, take it for now as part of the machinery that controls the loop, here a loop over all the integers from 1 to 100.

Prompts like `2.` and `3.` above are inserted by Stata in interactive sessions. Do not include such prompts in your programs or do-files.

The function `rnormal()` was introduced in Stata 10.1. If you are using an older version of Stata, `invnormal(uniform())` will work instead.

The example is deliberately as symmetrical as possible. We have 100 samples all of size 100. We could think of each observation (row) as a separate sample or of each variable (column) as a separate sample. Each viewpoint is perfectly valid statistically. The only issue is what kind of further procedures are easiest in Stata for each viewpoint. So we could follow the grain and type

```
. summarize x*
```

to see the minimums and maximums. That is easy, but we now have a long table that is awkward to work with. For example, it is not especially obvious how to get the extremes out of the table into new variables, although you can probably think of ways to do that, at least crudely. At the very worst, you could just type the extremes into a new variable. Clearly, that is a poor method and limited to moderate sample sizes. We need something much better.

It would be better in many ways to work rowwise. We have already seen that `min()` and `max()` are functions that will work rowwise. On sight of the syntax, however, the heart sinks, at least slightly. The syntax—which can be seen directly from `help min()` or `help max()`—requires that the arguments to each function be separated by commas. Do we really need to type out 100 names separated by 99 commas? As you might hope, the answer is no. There are higher-level ways to work with that many arguments. These ways can be especially useful when you are doing something like this in a program or do-file where automation is essential.

Here is one way to do it. The `unab` command ([P] **unab**) has one role, to "unabbreviate" a variable list. We can agree that the word is ugly, but the role is useful.

```
. unab xvars: x*
```

puts the unabbreviated list from `x1` to `x100` in a local macro, as we can see for ourselves by displaying it.

```
. display "`xvars´"
x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 x16 x17 x18 x19 x20 x21 x22
> x23 x24 x25 x26 x27 x28 x29 x30 x31 x32 x33 x34 x35 x36 x37 x38 x39 x40 x41
> x42 x43 x44 x45 x46 x47 x48 x49 x50 x51 x52 x53 x54 x55 x56 x57 x58 x59 x60
> x61 x62 x63 x64 x65 x66 x67 x68 x69 x70 x71 x72 x73 x74 x75 x76 x77 x78 x79
> x80 x81 x82 x83 x84 x85 x86 x87 x88 x89 x90 x91 x92 x93 x94 x95 x96 x97 x98
> x99 x100
```

The local macro `xvars` is just a place where we can hold a piece of text. To define it, we use a `local` command. To use it, we refer to it using single quotes, ‘ ’, to include the name. As the example makes clear, that text can include numeric as well as alphabetic characters. If you want to know more about macros, one place to start is the tutorial just mentioned (Cox 2002a). Note that local macros in `forvalues` and `foreach` loops are special: they are defined tacitly and they automatically disappear at the end of their loop.

A subtlety worth underlining here is that the double quotes, `" "`, are essential in the last `display` statement to ensure that the macro is shown as literal text. Otherwise, `display` will try to interpret the variable names and show their values. The best way it can do that is to show the values in the first observation.

The `unab` command does not require that the variable list fed to it is of simple form, in our case `x*`. It just needs to be a variable list. It can be as messy in form as you like so long as it is a variable list. `aardvark-zebra wwg j? foo*` is fine so long as that is a variable list.

The problem of getting input suitable for `min()` or `max()` is half solved already. We just need to replace those spaces in the local macro `xvars` with commas. One virtue of `unab` is that you can be sure that the resulting list of names is separated by single spaces and that there are no leading or trailing spaces. `min()` or `max()` would choke on arguments that began or ended with `,` or contained `,,`. So would `missing()` or `inlist()` for that matter.

```
. local xarg : subinstr local xvars " " ",", all
```

is the device we need. A local named `xarg` is produced by substituting single spaces with commas into the local macro called `xvars`, and that is done in all possible instances. You may want to look at the documentation, most conveniently at the help file for extended macro functions, `help extended_fcn`.

Now rowwise extremes can be put into new variables:

```
. generate xmin = min(`xarg´)
. generate xmax = max(`xarg´)
```

That is, you type the macro name, but the `generate` statement will see the macro contents.

Those new variables could now be used in further analysis. The same devices of producing an unabbreviated and comma-separated variable list could be used with `missing()` or `inlist()`.

2.2 A note on inlist()

`inlist()` deserves a separate comment. The discussion and examples given in Cox (2006b) fall short of explaining rowwise uses of this function. Thus, for example, `if inlist(1, a, b, c, d, e, f)` is a neat alternative to `if a == 1 | b == 1 | c == 1 | d == 1 | e == 1 | f == 1`. Similarly, `if inlist(y, x1, x2, x3, x4, x5)` is a neat alternative to `if y == x1 | y == x2 | y == x3 | y == x4 | y == x5`. Yet more complicated examples are possible.

3 Loops

We have already used a `forvalues` loop to loop rowwise when generating a bunch of new variables. Let's look at some further examples showing how useful loops can be.

3.1 Observation minimums and maximums again

Suppose, for example, that you did not know the tricks for getting comma-separated variable lists used in the previous section. You could make use of a simple looping algorithm for getting rowwise extremes. We will first illustrate with minimums for the same 100 variables.

1. Initialize the minimum as `x1`.

2. Loop across `x2` to `x100`. If any variable is less than the minimum so far, it becomes the new minimum.

That translates easily into code:

```
. generate xmin2 = x1
. forvalues j = 2/100 {
  2. replace xmin2 = min(xmin2, x`j´)
  3. }
```

Code often looks obvious when it is correct, but it may well be worth spelling out some mistakes that beginners often make and that experts occasionally make too. Why is the code not as follows?

```
. forvalues j = 1/100 {
  2. generate xmin2 = min(xmin2, x`j´)
  3. }
```

There are two separate reasons why not.

The first time around the loop, for `j` equal to 1, Stata sees on the right-hand side `min(xmin2, x1)`. But `xmin2` does not exist yet. Stata cannot proceed any further, so the loop will fail.

Even if you fixed that, say, by an initialization before the loop to missing (`.`),

```
. generate xmin2 = .
```

the loop would fail the second time around. When `j` is 2, Stata sees on the left-hand side `generate xmin2`. But `xmin2` already exists. As the original code shows, you need a `replace` statement instead of a `generate` statement. Initializing with a `generate` statement followed by a loop over `replace` statements is a very common pattern.

A useful refinement to the code is to insist that Stata does all this `quietly` (which means, here, suppressing messages about changed values). The loop could be

```
. generate xmin2 = x1
. quietly forvalues j = 2/100 {
  2. replace xmin2 = min(xmin2, x`j´)
  3. }
```

or it could be

```
. generate xmin2 = x1
. forvalues j = 2/100 {
  2. quietly replace xmin2 = min(xmin2, x`j´)
  3. }
```

What is the difference? In this example, none in terms of the results you see when the code is executed. But I tend to put `quietly` on the loop as a whole, as in the first of these two segments, for two reasons. First, in more complicated loops, there could be several potentially noisy commands that would all need attention. Typing `quietly` just once is less work. Second, commands within the loop are more likely to be longer, so putting `quietly` where there is more space has the edge.

Separately from such details, `quietly` tends to be added at a late stage when you are happier that the code is good. The noisier output can make any bugs more obvious.

By the way, you might wonder if there is any objection to

```
. generate xmin2 = .
. quietly forvalues j = 1/100 {
  2. replace xmin2 = min(xmin2, x`j´)
  3. }
```

In terms of substance or style, I think there is very little in it. Indeed, some people might prefer this last segment as a little clearer than the first segment. Stata is obliged to do a bit more work, but we could usually live with that.

We should perhaps spell out what should be familiar. In all these loops, there is another tacit loop over observations. Stata ensures that the `replace` takes place separately in each observation.

In other problems, you may not find the variables to be neatly numbered. The most general approach to looping over variables would use `foreach` rather than `forvalues`. `forvalues` has an edge in speed whenever both constructs are possible.

3.2 Is initialization safe?

A crucial detail in writing loops is ensuring that the initialization is safe, meaning that it cannot prejudice the result. Carelessness on this point is a frequent source of bugs.

For calculating a minimum using the algorithm above, an initialization of 0 would not be safe if values were always positive, because the initial value would never be revised. (When each new value is encountered, the question would be whether it was less than 0, and the answer would always be "No".) Even if values were always positive, an initialization of `.` would still be better style. When there is a choice, it is better to base

code on something that is always true—as that `min(.,` *anything*`)` is *anything*—rather than on something that may be true. The latter code is more difficult to understand without a context and all too likely to be copied to a problem where the contingent assumption is no longer satisfied.

For calculating a sum, an initialization of . would never be safe, because it would be unaffected by anything added to it and Stata would exit from the loop with a sum that was missing, a case of termination with extreme prejudice. (Strictly, the sum should be missing if it were the case that all values encountered were missing, but that would still be getting the right answer for the wrong reason.) Conversely, an initialization of 0 is safe for a sum, as is 1 for a multiplication.

3.3 Maximums and missing values

We could use an equivalent algorithm for calculating maximums:

```
. generate xmax2 = .
. quietly forvalues j = 1/100 {
  2. replace xmax2 = max(xmax2, x`j´)
  3. }
```

A key detail here, and one that may occasionally surprise, is that in Stata `max(.,` *anything*`)` is always *anything*, even though it is also true elsewhere in Stata that missing is treated as higher than any nonmissing value. Thus the maximum of 42 and missing,

```
. display max(42, .)
42
```

is 42, not missing. In essence, this is a deliberate inconsistency, designed into Stata's code on the grounds that the answer is more likely to be what you want.

The question deserves careful attention. Set Stata aside for a moment and consider the following: which is the maximum of 0, 1, 2.71828, 3.14159, 42, and missing? There are two defensible answers: 42; and we cannot say, because one value is missing. The latter is, as said, a defensible answer, but if we followed the same attitude elsewhere, we would end up shrugging our shoulders in many real statistical problems because missing values are so common. Giving the most definite answer we can is usually a good idea.

Now reintroduce Stata to the conversation. A third answer now is that the largest value is missing, because of Stata's rule that missing is to be considered as larger than any nonmissing value. However, notice two things. This third answer is not the same as the second answer: the reasoning is quite different. The rule is one that Stata needs because it needs to decide what to do in certain problems. For example, if you `sort` values, where do numeric missings go? Stata's rule implies that they must be sorted above all numeric nonmissings, but to anyone outside Stata, the rule is no more than a convention. Indeed, a convention of treating missings as arbitrarily low is equally defensible and can be encountered in other software. So this reasoning would be quite arbitrary to anyone outside Stata.

With all this in mind, let's consider what to do if for some reason we really did want a row maximum to be returned as missing whenever any value in an observation were missing. More precisely, say we want system missing, `.`, to be returned if any missing values exist in that observation, whether `.` or the extended missing values `.a`, ..., `.z`.

```
. generate xmax3 = x1
```

can be our initialization. Within the loop, we can follow the same device of comparing each variable and the maximum previously encountered, but we will have to be more delicate in making the comparison, given the way that `max()` behaves.

The `cond(`*x*`,`*a*`,`*b*`)` function is suitable machinery. Its main idea is to test a condition for truth or falsity, and assign results according to each possibility. Complicated branching can be handled by nesting two or more function calls. For a tutorial, see Kantor and Cox (2005).

```
. quietly forvalues j = 2/100 {
  2. replace xmax3 = cond(missing(x`j´, xmax3), ., max(x`j´, xmax3))
  3. }
```

The branching inside `cond()` can be put into words like this:

1. If either this variable or the maximum so far is missing, we want the new maximum to be missing. We need to peel off this case first and not feed values to `max()`, because `max()` will always choose a nonmissing value if one is present. The "or" here is inclusive and includes the case where both values are missing.

2. Otherwise, we want the new maximum to be the maximum of this variable and the maximum so far. `max()` is safe for this case because if neither value is missing, then both values must be nonmissing.

Alternatively, if we have extended missing values (any of `.a`, ..., `.z`), then we may simply want to see the largest of those. That is even easier:

```
. generate xmax4 = x1
. quietly forvalues j = 2/100 {
  2.  replace xmax4 = x`j´ if x`j´ > xmax4
  3. }
```

The inequality operators do treat missing values as larger than nonmissings and extended missing values as larger than system missing (`.z` is largest of all).

3.4 Any, all, and counting

`max()` and `min()` can be useful tools even when the problem is not calculating extremes. There is a simple correspondence between `max()` and `min()` and what may be called *any* and *all* problems. Consider a bunch of numeric variables (`a`, `b`, `c`, `d`, `e`, and `f`) and an interest in which of those variables is positive. For the moment, set the issue of whether missing values are present aside. Precisely, we can ask

1. Are any of the variables positive in each row? (answer 1 if true, 0 if false)
2. Are all of the variables positive in each row? (answer 1 if true, 0 if false)

While we are answering those questions, we can throw in a third:

3. How many of the variables are positive in each row?

Two simple algorithms for the first two questions are

any.1 Assume no variables are positive in each row.

any.2 Looping across variables, change your mind about that row if you meet a positive value for that variable in that row.

all.1 Assume all variables are positive in each row.

all.2 Looping across variables, change your mind about that row if you meet a zero or negative value for that variable in that row.

In code, we first initialize, also adding an initialization for a count:

```
. generate byte any = 0
. generate byte all = 1
. generate byte count = 0
```

Our loop should use `foreach`:

```
. foreach v of var a b c d e f {
  2. replace any = max(any, `v > 0 & `v´ < .)
  3. replace all = min(all, `v´ > 0 & `v <  .)
  4. replace count = count + (`v > 0 & `v´ < .)
  5. }
```

So the variable `any` is born as 0, but we bump it up to 1 if we meet any variable for which `(‘v’ > 0 & ‘v’ < .)` is true, which numerically evaluates to 1. Conversely, the variable `all` is born as 1, but we bump it down to 0 if we meet any variable for which that condition is false, which numerically evaluates to 0. We cannot use `inrange(‘v’, 0, .)` here, because that includes any zero values; see Cox (2006b) for more details. If you knew that all missings were really positive, or wanted to include them for some reason, then the condition `‘v’ > 0` would suffice.

Declaring the variables to be `byte` is not essential unless memory is very tight but is nevertheless recommended as good style.

The beauty of this approach is that it can be extended to any circumstance that can be stated as a true or false condition, including tests on string variables. We might want to split the code for very complicated problems into more commands. That would not be problematic.

You should note that there is some redundancy in this code. You could produce just the counting variable `count` and then test whether `count > 0` or `count == 6` (or, more generally, equal to the number of variables):

```
. generate byte any2 = count > 0
. generate byte all2 = count == 6
```

Knowing two ways to solve a problem is always preferable to knowing none. Indeed, we will mention yet other solutions in the next section.

4 egen functions

The `egen` command is part of official Stata. It is in essence a little engine that drives various functions written in the Stata language. Despite the use of the same term, these functions are quite distinct from Stata's functions in the strict sense, as documented at [D] **functions**, which are all part of the executable; or indeed distinct from Mata's functions, whether built in or user written. A habit of always carefully referring to `egen` functions as such serves everyone well in communication.

The help file for `egen`, the manual entry [D] **egen**, and the tutorial in Cox (2002b) give overviews or introductions in different styles.

If you already know `egen`, you may be surprised that I did not leap straight into a review of various `egen` functions that are applicable rowwise. One good reason for not doing that is because they are typically built on the principles outlined in previous sections. Another good reason is that users sometimes jump to conclusions on what is possible rowwise from what is evidently and readily available as canned code. That reaction is especially unfortunate. One of the main motives behind this column is a desire to make clear that rowwise operations need not be nearly so difficult as you might imagine.

What may not be immediately obvious from the official Stata documentation in particular is that `egen` is highly extensible, so functions for `egen` can often be written easily and quickly. Frequently, taking an existing function as template and changing just a few lines is quite enough. In fact, writing a new `egen` function is often a very good exercise for anyone interested in learning more about Stata programming, even though it would not necessarily help very much in learning how to write quite different Stata programs.

As it happens, the number of user-written `egen` functions (most of them accessible by typing `findit egen`) much exceeds the number of officially written Stata functions, although, unsurprisingly, the official functions tend to cover most of the more important tasks. So watching out for user-written `egen` functions is good practice, if only because you may identify a function closer to what you want.

The official suite of egen functions includes these functions that are applicable row-wise:

```
anycount()   rowfirst()   rowmean()   rownonmiss()
anymatch()   rowlast()    rowmin()    rowsd()
concat()     rowmax()     rowmiss()   rowtotal()
diff()
```

We will not cover them in detail here because their documentation is easily accessible. If any of these functions looks possibly interesting or useful, but not familiar, you know where to look.

You may now be learning for the first time that rowmin() and rowmax() exist as canned egen functions for observation minimums and maximums. You should still be better off for knowing that they could have been invented or emulated quite easily if they did not exist—and especially for knowing that there are ways of producing variants when their built-in behavior is not what you want.

In the previous section, we looked at any or all problems. If we had a bunch of indicator or dummy variables, we could use these egen functions to find solutions to such problems for those variables. The row minimum of a set of indicator variables will be 1 if all those variables equal 1 in that row. The row maximum of a set of indicator variables will be 1 if any of those variables equals 1 in that row. More indirect and less elegant solutions are also possible by using anycount(), anymatch(), concat(), and rowtotal(). However, note the starting point here of possessing a bunch of indicators. If those indicators did not already exist, it would be better to use the approach from first principles outlined earlier.

You can get a better understanding of how egen works with egen functions by glancing at the code. Know that egen itself is implemented as egen.ado and that any egen function, *foobar*, will be implemented in _g*foobar*.ado. Know further that the command viewsource has the role of finding a file of Stata code and opening the Viewer on that file (Cox 2006a). Thus

```
. viewsource _growmax.ado
```

takes you directly to the official implementation of the egen function rowmax().

5 The problem of row medians

Let's consider a problem that did not make the official Stata list of egen functions: row medians.

5.1 Medians

To recapitulate some basics: If you want a median of a variable, you could sort on that variable and pick out the median yourself. Some care is needed whenever you are

interested in only some of the observations or whenever there are missing values. Many users fire up `summarize, detail`, which takes care of all such matters, and identify the median within the results. Possible alternatives include `centile` and `tabstat`. `egen`'s `median()` function is available to put medians into a new variable, as will often be needed when there is a structure of different groups (e.g., panels).

When you have a bunch of variables of the same kind and you want row medians across those variables, sometimes there is an easy answer. If those variables are really panel or longitudinal data, you should `reshape long` and work with a different data structure. As mentioned in the introduction, life will be much easier that way, not only for row medians, which are now just panel medians, but also for almost all kinds of analysis you might want to do with such data.

If you know that the values of your row variables are in numerical order, so that, for example, `y1` $\leq$ `y2` $\leq$ `y3`, and so forth, then the median can be calculated directly as either one of the variables or the mean of two of the variables, depending on whether the number of variables is odd or even. But that recipe would be messed up by any missing values.

Next come situations when you have few variables (two, three, or four) over which you want to take the row median and again no values are missing. The median of two variables is the same as their mean, so that first case is easy:

```
. generate median = (y1 + y2)/2
```

A less-known trick for three variables also makes solving the problem simple:

```
. generate median = y1 + y2 + y3 - min(y1, y2, y3) - max(y1, y2, y3)
```

In words, work out the row sum, and then subtract the minimum and the maximum. What remains must be the median.

Now a trick for four variables is immediate:

```
. generate median = (y1 + y2 + y3 + y4 - min(y1, y2, y3, y4) -
> max(y1, y2, y3, y4))/2
```

In words, work out the row sum, and then subtract the minimum and the maximum. What remains is the sum of the two inner values, and halving gives the median.

Simple tricks for five or more variables, in general, are not in evidence.

Some thought shows that ties pose no problem to any of these tricks. The more awkward assumption is that no values are missing. There is some scope for salvaging problems with missing values.

With any missing values, the median of two can be salvaged by using `egen`'s `rowmean()` function,

```
. egen median = rowmean(y1 y2)
```

or by using

```
. generate median = (y1 + y2)/2
. replace median = max(y1, y2) if median == .
```

The last two commands exploit the fact, already explained in section 3.3, that `max(y1, y2)` is nonmissing whenever one of the values is. If both values are missing, we do not lose out, because some missings are just overwritten by missings. You could write `min(y1, y2)` if that made you feel more comfortable.

Similarly, the median of three can be salvaged. Consider again

```
. generate median = y1 + y2 + y3 - min(y1, y2, y3) - max(y1, y2, y3)
```

If `y1`, `y2`, and `y3` are all missing, you already have the only possible answer: missing. The situations to fix are that just one variable is missing and that two variables are missing in each observation.

```
. replace median = max(y1, y2, y3)
> if (missing(y1) + missing(y2) + missing(y3)) == 2
```

If two variables are missing, then we can get the other one from `max()`, and it is automatically the median:

```
. replace median = (cond(missing(y1), 0, y1) +
>                   cond(missing(y2), 0, y2) +
>                   cond(missing(y3), 0, y3))/2
>    if (missing(y1) + missing(y2) + missing(y3)) == 1
```

If one variable is missing, then the mean of the other two gives the median. We make sure that missings are ignored in the sum by using 0 instead. You could do the same thing more easily with `egen`'s `rowmean()` function:

```
. egen rowmean = rowmean(y1 y2 y3)
. replace median = rowmean if (missing(y1) + missing(y2) + missing(y3)) == 1
```

5.2 An egen function

The code seen so far in this section has some curiosity value in showing tricks for row medians of two, three, or four variables. These tricks have found application in some special problems. For example, medians of three values feature in some robust nonlinear smoothing methods (e.g., Tukey [1977]). But we still need a program embodying a more general approach.

However, there is no official `egen` function for row medians. The explanation is partly historical. Before Stata 9, the problem was a little difficult to solve well. In essence, two approaches were possible before Stata 9.

The first is to loop over observations, copy values for each observation into a variable, and then get the median. Unfortunately, this approach assumes that the number of variables concerned is no greater than the number of observations. That is usually

but not necessarily true. More importantly, this approach can be slow indeed with interpreted code.

The second is to restructure the dataset on the fly, calculate medians, and then restructure back. Arguably, restructuring a dataset is not something that should be done in the middle of an `egen` function, but in any case this approach could easily fail if the number of observations required exceeded the maximum allowed or if enough memory were not available.

With Stata 9, however, came a more positive opportunity: to use Mata. If you want to know more about Stata's matrix language, Mata, the main sources are the Mata manual and William Gould's Mata matters columns in this journal. Baum (2009) contains an excellent introduction. Those looking at Mata code for the first time and thinking "This is like C!" are right on target.

The `egenmore` package available from the Statistical Software Components (SSC) archive includes a `rowmedian()` function. Use `ssc` ([R] **ssc**) if you wish to install that package. This function is much faster than previous `egen` functions, even though the basic loop is still a loop over observations, and it requires little extra memory. Here is the central part of the code:

```
mata:
void row_median(string scalar varnames,
                string scalar tousename,
                string scalar medianname,
                string scalar type)
{
        real matrix y
        real colvector median, row
        real scalar n
        st_view(y, ., tokens(varnames), tousename)
        median = J(rows(y), 1, .)
        for(i = 1; i <= rows(y); i++) {
                row = y[i,]´
                if (n = colnonmissing(row)) {  // sic
                        _sort(row, 1)
                        median[i] =
                                (row[ceil(n/2)] + row[ceil((n + 1)/2)])/2
                }
         }
         st_addvar(type, medianname)
         st_store(., medianname, tousename, median)
}
end
```

The algorithm should seem unsurprising. The variables in question are seen through a Mata view. A column vector of medians is initialized to missings. Values in each observation (row) are copied and transposed into a column vector called `row`. (That may seem backward, but while the name `row` reflects what the contents are, a row of the data, in Mata it is easier to handle those contents as a column vector.)

Note a small programming trick: the Mata function `colnonmissing()` counts nonmissing values in each column. The test

```
if (n = colnonmissing(row))
```

does two things in quick succession. The assignment

```
n = colnonmissing(row)
```

puts the count in the scalar `n`. Once `n` is calculated, it can be used in the test

```
if (n)
```

That will be true whenever `n` is nonzero, which can only be when `n` is one or more, and false when `n` is zero. There is no point to calculating a median if all the values in a row are missing. Recall that the median was initialized as missing, so we would not change our mind by doing the work.

If there are nonmissing values, then we sort the row on the fly and pick out the median. Clearly, the rule used has to be able to cope with both odd and even numbers of values. Textbooks usually introduce a rule for ordered values: if the sample size n is odd, select the value with position $(n + 1)/2$; if n is even, average the values with positions $n/2$ and $n/2 + 1$. This is equivalent to a rule to average the values with positions `ceil(n/2)` and `ceil((n + 1)/2)`. `ceil()` (think ceiling) yields an integer, rounding up if necessary. For a short tutorial on `ceil()` and its sibling, `floor()`, see Cox (2003).

Some experimenting within Mata will make that clear:

```
: n = 1..10
: ceil(n/2)\ceil((n :+ 1)/2)
        1   2   3   4   5   6   7   8   9   10
  +-------------------------------------------+
1 |  1   1   2   2   3   3   4   4   5   5   |
2 |  1   2   2   3   3   4   4   5   5   6   |
  +-------------------------------------------+
```

So, for n odd, as for $n = 1$, 3, 5, 7, and 9, we average two copies of the middle value, and for n even, as for $n = 2$, 4, 6, 8, and 10, we average the two middle values.

6 Looping over observations using Mata

The `egen` function just discussed is based on a simple idea, but one so useful for rowwise operations that it deserves a small flag.

As we loop across observations, we can focus on values for each observation, which are held in Mata as a row vector, part of a matrix of data. We should set up that matrix to be a view whenever we can (Gould 2005).

Mata has its own bias, inherited from Stata, so that many operations are done on columns, not on rows. You can sort columnwise, but not rowwise, for example. This

problem, however, is trivial: just transpose the row vector to a column vector and then do the work required on that column vector. Transposing the values for one observation is much, much less of a big deal than transposing or reshaping much of or all of the dataset. If the result of some work is a scalar, as it often will be, then we use that scalar result. The row median is a case in point. If the result is another column vector, we can easily transpose that back to a row vector.

We will look at further examples of each kind, scalar results and vector results.

7 Numbers of distinct values

Imagine once more numeric variables: `a`, `b`, `c`, `d`, `e`, and `f`. Now consider how we would compute the number of distinct values in each observation. If an observation contains the values 1, 1, 1, 1, 2, and 2, then it contains two distinct values, 1 and 2. Some people would call those unique values, regardless of the fact that neither value occurs just once. This problem is akin to that of determining distinct observations, discussed in the previous column (Cox and Longton 2008). Although the example refers to numeric variables, the same question can be asked of string variables.

The problem is also an example of counting across rows, earlier considered in section 3.4. But the twist of counting distinct values makes it more difficult with the approach considered there. A loop across variables would have to consider whether any value had been met previously in the loop, which is trickier.

A more congenial approach uses Mata in the way described in the last two sections. The `egenmore` package available from the SSC archive includes two `egen` functions: `rownvals()` for the number of distinct numeric values and `rowsvals()` for the number of distinct string values. Most of the code is identical to that of the `rowmedian()` function discussed in section 5.2. The interesting differences lie at the heart of each function. Focus only on the default cases, which are to ignore missings. For distinct numeric values, we have

```
for(i = 1; i <= rows(y); i++) {
        row = y[i,]´
        nvals[i] = length(uniqrows(select(row, (row :< .))))
}
```

and for distinct string values, we have

```
for(i = 1; i <= rows(y); i++) {
        row = y[i,]´
        svals[i] = length(uniqrows(select(row, (row :!= ""))))
}
```

Just as in elementary algebra, telescoped expressions with nested parentheses are best read from the inside outward. The column vector `row` (that naming was explained in section 5.2) is thinned down by omitting any missing elements. It is then thinned down further by removing any duplicated elements. The `length()` of what remains—equivalently here the number of rows `rows()`—is the number we seek.

The obvious variation on the default is to include any missing values. The expression now would be `length(uniqrows(row))`, for both numeric and string cases.

Naturally, we lean heavily here on the canned Mata function `uniqrows()`, but that is what canned functions are for.

8 Row sorting and ranking

The problem of row medians is often extended to an interest in row sorting or ranking a set of variables of the same kind (or both). Although the idea of changing the data in place is sometimes entertained, it is safer to treat both problems as a mapping of a set of existing variables to a set of new variables.

Thus three numeric variables, `y1`, `y2`, and `y3`, might be sorted to `s1`, `s2`, and `s3`, such that `s1` $\leq$ `s2` $\leq$ `s3`, and ranked in `r1`, `r2`, and `r3`, such that `r1` contains the rank of `y1` in the three variables, and so forth. If the values in one observation were 42, 7, and 56, then the corresponding sorted values are 7, 42, and 56 and the corresponding ranks are 2, 1, and 3, assuming lowest maps first in sort order or ranking. Ascending sorts are the default in Stata, as is generally so in statistical computing.

Interest in row sorting and ranking can include string variables as well as numeric. A problem with dyads requiring the row sorting of two string variables is discussed in Cox (2008), although that does not need the machinery here. Note, however, that Stata's sorting of strings does not match the order used in any standard dictionaries, even for the English or American languages. For example, all uppercase letters A to Z sort before all lowercase letters a to z. On occasion, converting strings to one case or the other by using the functions `upper()` or `lower()` in Stata or `strupper()` and `strlower()` in Mata may be advisable before sorting and ranking.

Now that we seek a result of several new variables, one consequence is that we should leave `egen` behind. It can produce only one new variable at a time, and although that does raise the possibility of calling it in a loop, a more direct approach is to program the production of several variables from one command.

8.1 Complications in sorting and ranking

The complications that can affect sorting and ranking are tied values, missing values, and any preference for descending sorting or ranking rather than ascending.

Ties are unproblematic for sorting. Whether any value occurs twice or more is immaterial, because the sorted sequence is identical regardless of where the tied values came from. Ties are, however, a small headache for ranking, because they can reasonably be treated in different ways.

Perhaps the most common practice in statistical science is to assign to tied values the mean of the ranks that would have been assigned otherwise, a practice likely to be familiar to you from nonparametric statistics. For example, this method for ties is the default of `egen`'s `rank()` function.

Perhaps the most common ranking practice outside statistical science—whenever ties are not broken by injecting other information—is to take the lowest rank possible, so that 2, 3, 3, 5, 5, 5 would be ranked 1, 2, 2, 4, 4, 4. Thus rank is defined precisely as 1 + the number more extreme. Thomson (2001, 139) refers to this rule as schoolmaster's rank, but the term is likely to seem obscure, if not objectionable to schoolmistresses.[1] A convention with opposite flavor takes the highest rank possible, yielding for that example 1, 3, 3, 6, 6, 6. Thus rank is now defined precisely as the number as or more extreme. Let's call these *low* and *high ranks*, respectively. High ranks are likely to be more familiar to you in their guise of cumulative frequencies and may be calculated in Stata by using the `cumul` command. Ranks with ties broken by means are just the means of the low and high ranks.

Sometimes there is a preference for descending rather than ascending ranks. A small point that may be overlooked is that even when ascending ranks do not appear to be supported by a command, they are always obtainable by negating the variable concerned, which clearly reverses the order. The `egen` function `rank()` makes this especially easy, because the argument it takes is an expression: a negated variable name is one such expression.

8.2 rowsort and rowranks

New versions of programs `rowsort` and `rowranks` are published with this column. (Earlier versions were made available via the SSC archive.) The main idea is already familiar: passing the data to Mata, looping over rows, sorting each row, and using each vector of results to produce a row of sorted values or ranks. Because each program comes with its own detailed help file, including remarks not reproduced here, examples are left to readers' own experiments. A formal statement of the programs' syntax follows.

Syntax for rowsort

`rowsort` *varlist* [*if*] [*in*] , `generate(`*newvarlist*`)` [`descending` `highmissing`]

Description

`rowsort` creates new variables containing the row-sorted (-ordered) values in each observation of *varlist*. *varlist* should contain either only numeric variables or only string variables.

By default, the first (second, ...) new variable contains the lowest or first-ordered (second-ordered, ...) value within each observation. The `descending` option may be used to reverse order. With strings, uppercase letters sort before lowercase.

1. "Schoolmaster" is an old-fashioned word for a male teacher in a school, schools here definitely not including colleges or universities.

Options

`generate(`*newvarlist*`)` specifies new variable names for the variables to be generated, one for each variable in *varlist*. *newvarlist* may be specified in hyphenated form, as in `s1-s5`. This option is required.

`descending` specifies that *newvarlist* should contain descending values, so that ordering is from highest, or last, downward.

`highmissing` specifies that missing values should be treated as higher than nonmissing values. This option bites for numeric values only when `descending` is also specified and does not bite for string values if `descending` is also specified. With these two options, `1 . 3 . 5 . 7` would be sorted to `7 5 3 1 . . .` and generated as such. Note also that `1 .c 3 .b 5 .a 7` would be sorted to `7 5 3 1 .c .b .a` and generated as such.

Syntax for rowranks

`rowranks` *varlist* [*if*] [*in*], `generate(`*newvarlist*`)` [`descending` `highmissing` `missing` `method(`*method*`)`]

Description

`rowranks` creates new variables giving the row ranks of values in each observation of *varlist*. *varlist* should contain either only numeric variables or only string variables.

By default, lowest values rank lowest and the ranks created are distinct, with ties being broken by the order of occurrence within *varlist*. For example, given values on five variables,

```
2  3  5  7  9
2  3  3  3  9
```

the ranks created are (for both observations)

```
1  2  3  4  5
1  2  3  4  5
```

The `descending` option may be used to reverse order. With strings, uppercase letters sort before lowercase.

Options

`generate(`*newvarlist*`)` specifies new variable names for the variables to be generated, one for each variable in *varlist*. *newvarlist* may be specified in hyphenated form, as in `r1-r5`. This option is required.

`descending` specifies that ranking should be from highest downward.

`highmissing` specifies that missing values should be treated as higher than nonmissing values. This option bites for numeric values only when `descending` is also specified and does not bite for string values if `descending` is also specified.

`missing` specifies that missing values be included in the ranking. By default, missing values are mapped to missing ranks.

`method(`*method*`)` specifies an alternative to the default. Alternatives are `low`, `high`, or `mean`. Any abbreviation is allowed. `low` specifies the use of low ranks, $1 + (\# <$ this value). `high` specifies the use of high ranks, ($\# \leq$ this value), a.k.a., cumulative frequencies. `mean` specifies the use of mean ranks so that the sum of ranks is preserved under ties.

9 Conclusions

Working rowwise is easier than you might fear.

Several Stata functions can work rowwise, notably, `max()`, `min()`, `missing()`, and `inlist()`. Some simple macro manipulations help in preparing long lists of comma-separated arguments.

You can write your own loops over variables by using `forvalues` or `foreach`. A common pattern is to initialize a variable by using `generate` and then to loop over variables by using `replace`. Note in particular the duality of `max()` and "any" and of `min()` and "all".

The official command `egen` includes several functions that work rowwise. (Not all their names start with `row`.) Other such user-written functions are in the public domain, and yet others can be written using existing code as a template to be modified. Examples of row medians and the number of distinct numeric or string values in each row show that it can be congenial and efficient to pass the tricky central part of each calculation to Mata.

A more challenging class of problems entails working rowwise with a set of variables to produce another set of variables. Row sorting and ranking are good examples. Here, again, a combination of Stata and Mata is a recommended strategy.

10 Acknowledgments

William Gould helped improve the row median code. Jeffrey Arnold's program `sortrows`, available from the SSC archive, helped provoke the versions of `rowsort` and `rowranks` published here. In particular, their `highmissing` option was suggested by an option of `sortrows`.

11 References

Baum, C. F. 2009. *An Introduction to Stata Programming*. College Station, TX: Stata Press.

Cox, N. J. 2002a. Speaking Stata: How to face lists with fortitude. *Stata Journal* 2: 202–222.

———. 2002b. Speaking Stata: On getting functions to do the work. *Stata Journal* 2: 411–427.

———. 2003. Stata tip 2: Building with floors and ceilings. *Stata Journal* 3: 446–447.

———. 2006a. Stata tip 30: May the source be with you. *Stata Journal* 6: 149–150.

———. 2006b. Stata tip 39: In a list or out? In a range or out? *Stata Journal* 6: 593–595.

———. 2008. Stata tip 71: The problem of split identity, or how to group dyads. *Stata Journal* 8: 588–591.

Cox, N. J., and G. M. Longton. 2008. Speaking Stata: Distinct observations. *Stata Journal* 8: 557–568.

Gould, W. 2005. Mata Matters: Using views onto the data. *Stata Journal* 5: 567–573.

Kantor, D., and N. J. Cox. 2005. Depending on conditions: A tutorial on the cond() function. *Stata Journal* 5: 413–420.

Thomson, N. 2001. *J: The Natural Language for Analytic Computing*. Baldock, UK: Research Studies Press.

Tukey, J. W. 1977. *Exploratory Data Analysis*. Reading, MA: Addison–Wesley.

About the author

Nicholas Cox is a statistically minded geographer at Durham University. He contributes talks, postings, FAQs, and programs to the Stata user community. He has also coauthored 15 commands in official Stata. He wrote several inserts in the *Stata Technical Bulletin* and is an editor of the *Stata Journal*.