



The World's Largest Open Access Agricultural & Applied Economics Digital Library

This document is discoverable and free to researchers across the globe due to the work of AgEcon Search.

Help ensure our sustainability.

Give to AgEcon Search

AgEcon Search

<http://ageconsearch.umn.edu>

aesearch@umn.edu

*Papers downloaded from **AgEcon Search** may be used for non-commercial purposes and personal study only. No other use, including posting to another Internet site, is permitted without permission from the copyright owner (not AgEcon Search), or as allowed under the provisions of Fair Use, U.S. Copyright Act, Title 17 U.S.C.*

No endorsement of AgEcon Search or its fundraising activities by the author(s) of the following work or their employer(s) is intended or implied.

The Stata Journal (2016)
16, Number 3, pp. 632–649

A sparser, speedier reshape

Kenneth L. Simons
Department of Economics
Rensselaer Polytechnic Institute
simonk@rpi.edu

Abstract. A new command, `sreshape`, supports sparse and speedy reshaping of data. Often reshaped data are “sparse” in the sense of containing many missing values that are dropped after reshaping. `sreshape` automates the process of reshaping and dropping such missing information to avoid potential errors and, for both sparse and nonsparse data, yields speed improvements. Using large test datasets, `sreshape` achieves identical results to `reshape` 8 to 31 times faster in wide-to-long reshapes and 2 to 13 times faster in long-to-wide reshapes. Further suggested improvements may allow StataCorp to increase these speed gains in the built-in version.

Keywords: dm0090, `sreshape`, `reshape`, sparse data, missing data, long form, wide form, data management, speed

1 Introduction

Reshaping data between wide and long forms is a fundamental and frequently used transformation. With large datasets, however, reshaping can be frustrating. A reshape could easily take half an hour or more, keeping the user from other work. The frustration seems particularly poignant when the transformed data contain many blank cells, or missing data, which really only need to be dropped after transformation.

The blank-cells scenario occurs frequently. In company data, with one row per company, a few companies existed every year from, say, 1953 to 2013, but most companies formed after 1953 or closed before 2013, so columns (variables) giving company financial data in different years are usually blank. In patent data, with one row (observation) per patent, a few patents had hundreds or more of inventors, but the average number of inventors per patent was below three, so columns naming the inventors and their geographic locations are usually blank.

The obvious solutions do not always help. Keeping the data only in long form is sometimes a solution, but often not. Big datasets may have many attributes, each to be reshaped in a different dimension—such as a patent’s n_1 inventors, n_2 applicants, n_3 technology classes, n_4 claims, n_5 citations, and n_6 words in the abstract, requiring for a given patent $n_1 \times n_2 \times n_3 \times n_4 \times n_5 \times n_6$ rows in the true long form that shows all combinations of information. Keeping the data as a set of files to match-merge when needed is another solution, but this, too, has drawbacks, including speed and file management problems. Dropping variables can help, but there may be a reason the data were wanted in memory. Therefore, reshaping is often needed.

In this article, I introduce a new Stata command called **sreshape** that implements an efficient “sparse” reshaping process for data with a high percentage of blank cells. Such data are sparse in the sense that only a small fraction of cases contain data other than a default value, in this case a missing value code; hence, the data can be represented in a simplified form that allows more efficient reshaping. Because this scenario is common and could lead to errors in manipulation, it makes sense to consider this sparse data issue in the reshaping command.

The **sreshape** command, compared with Stata’s built-in **reshape** command, also turns out to be speedier. A speedup of 8 to 31 times faster, depending on the application, is observed below for wide-to-long reshapes, or 2 to 13 times faster for long-to-wide reshapes, while obtaining identical results. Greater speedups can occur with sparse datasets, yielding another near doubling of speed in the typical highly sparse data tests reported here. A much greater speedup, in a truly proper implementation, requires—for reasons explained later—further implementation by StataCorp.

The **sreshape** command works much like Stata’s built-in **reshape** command. However, for sparse data, some consideration is needed of how the data are organized and treated. The layout of reshaped data may differ in long form from Stata’s format in that observations with missing data are dropped. For example,

```
sreshape long a, i(id) j(t) missing(drop all)
```

accomplishes the same result as

```
reshape long a, i(id) j(t)  
drop if missing(a)
```

However, **sreshape** can do this in cases where use of **reshape** may be impractical.

If all reshaped values are missing for an individual i (for example, a company or patent) in the sample, the above treatment would entirely drop that individual from the dataset. It may instead be important to retain information about the individual: the individual’s identity as given by the value of the i variable, and variables that are constant within every individual i . To retain at least one observation per individual, while dropping all other observations with missing data, one can use the **missing(drop)** option, as in the following command:

```
sreshape long a, i(id) j(t) missing(drop)
```

Keeping at least one observation per individual retains all information but otherwise drops unneeded observations. After this command, the original form of the data is obtained using

```
sreshape wide
```

whereas the full original data could not be obtained after using the **missing(drop all)** option if, as shown in the next section, some individuals are dropped entirely from the sample because they have only missing values in the reshaped variables.

Using **sreshape** without the **missing()** option gives the same result as **reshape**.

sreshape is compatible with Stata 13 and Stata 14.

The command and its potential future implementation are discussed in the following sections. Section 2 illustrates different forms of the data when a sparse dataset is reshaped. Section 3 describes the operation of the command, including options beyond the **reshape** command and traits of how it operates. Section 4 provides examples of usage. Section 5 reports on tests of the program's reliability and performance. Section 6 explains the reasons for performance constraints in **sreshape**, why those performance constraints can only be addressed by StataCorp, and how StataCorp might implement the ideal version of an improved reshaping command.

2 Reshaping with missing data

Example data are shown in tables 1–4 to illustrate the formats and variables involved. Formats are wide and long, including three long forms that range from less concise to more concise. Variables are of types i , j , X_{ij} , and X_i . The wide data in table 1 include columns for variable types i , X_{ij} , and X_i . i variables, in this case the variable named **Patent**, identify individuals; any one individual i is identified by a unique combination of values of one or more i variables. In wide form, there must be exactly one row for each individual so that the combinations of i variables are indeed unique. X_{ij} variables contain values that differ across some j . Here the variables named **Inventor 1**, **Inventor 2**, and **Inventor 3** are all X_{ij} variables, and the inventor numbers 1, 2, and 3 in the variable names are the values of j . The X_i variables contain values that do not change with j . Here there is one X_i variable named **Application Date**, and it is a fixed value for each patent; it does not change with the inventor number. The first patent has three inventors, the second has one inventor, and the third is missing data on its inventor (actually, U.S. patent 223898 was invented by Thomas Alva Edison, but that information is not in the dataset).

Table 1. Wide-form data (example)

Patent	Inventor 1	Inventor 2	Inventor 3	Application Date
1540476	Peter M Hoffman	Charles Doering	Henry H Doering	12-Mar-1923
649621	Nikola Tesla			2-Sep-1897
223898				4-Nov-1879

The long data in table 2 contain the same information as table 1; however, instead of having separate X_{ij} variables for each j , there is one row for each pair of i and j values. This is the form yielded by Stata's **reshape long** command or by **sreshape** without the **missing()** option. All i – j pairs are kept, even though some i – j pairs have no inventor names. The j variable is now an explicit column of data, the variable named **Inventor Number**, instead of the j values being implicit as part of the variable names in wide form. The only X_{ij} variable is **Inventor**, and the only X_i variable is **Application Date**.

Table 2. Example long-form data, keeping all i - j pairs

Patent	Inventor Number	Inventor	Application Date
1540476	1	Peter M Hoffman	12-Mar-1923
1540476	2	Charles Doering	12-Mar-1923
1540476	3	Henry H Doering	12-Mar-1923
649621	1	Nikola Tesla	2-Sep-1897
649621	2		2-Sep-1897
649621	3		2-Sep-1897
223898	1		4-Nov-1879
223898	2		4-Nov-1879
223898	3		4-Nov-1879

The long data in table 3 also contain the same information as table 1, but the presentation is more concise because i - j pairs for which the X_{ij} data are always missing have been dropped if there is no loss of information. This is the form yielded by `sreshape` with the `missing(drop)` option. Specifically, the rows for patents 649621 and 223898 have been dropped for the second and third inventors because these patents have no known second and third inventors. The dropped rows do not cause a loss of information; they could always be reconstructed by creating new rows as needed to obtain all i - j pairs and then copying X_i variables to the new rows from the same i . The row for patent 223898 for the first inventor has been kept: if it were dropped, the data would no longer report the patent number 223898 and its application date.

Table 3. Example long-form data, dropping i - j pairs with redundant information

Patent	Inventor Number	Inventor	Application Date
1540476	1	Peter M Hoffman	12-Mar-1923
1540476	2	Charles Doering	12-Mar-1923
1540476	3	Henry H Doering	12-Mar-1923
649621	1	Nikola Tesla	2-Sep-1897
223898	1		4-Nov-1879

The long data in table 4 are similar to those above, but they exclude all rows in which the X_{ij} data are missing. This form is yielded by `sreshape` with the `missing(drop all)` option. This is useful if the goal is only to analyze entities contained in the X_{ij} data. Here one could count how often each inventor appeared in the dataset and the mean application date of each inventor's patents. However, some information has been lost. In these reshaped data, it is not apparent that patent 223898 was in the sample, and computing the minimum application date would yield a value of 2-Sep-1897 instead of 4-Nov-1879. Given only these data, one cannot reconstruct the wide-form data of table 1; the reconstructed wide-form data would exclude the row for patent 223898.

Table 4. Example long-form data, dropping all i - j pairs with all missing X_{ij} data

Patent	Inventor Number	Inventor	Application Date
1540476	1	Peter M Hoffman	12-Mar-1923
1540476	2	Charles Doering	12-Mar-1923
1540476	3	Henry H Doering	12-Mar-1923
649621	1	Nikola Tesla	2-Sep-1897

3 Syntax and operation

The **sreshape** command is best understood in terms of Stata's existing **reshape** command because most of its operation and options are the same as for **reshape**. All syntaxes of **reshape** can be specified as **sreshape**, except for the advanced usage of **reshape xi**, which requires a special option described below. Therefore, users should consult Stata's help system and [D] **reshape** for basic information about Stata's **reshape** command. Additional options for **sreshape** are described here.

3.1 Options beyond Stata's reshape command

In addition to those options in Stata's **reshape** command, the following options are allowed:

Alternative specification of i variables

newi(newvar) generates a new i variable with the specified name and sets it equal to the observation number in the data. This option may be used instead of **i(varlist)** but only with **sreshape long**.

Sparse data treatment

missing(keep|drop|drop all) specifies how to treat observations whose reshaped data are all missing when converting the data to long form. The default specification **keep** specifies to keep all observations, even if they contain only missing data in the reshaped variables. Alternatively, **drop** causes observations to be dropped if all reshaped X_{ij} data are missing, except that at least one observation (for one j value) is kept for each i . To drop all observations that contain only missing X_{ij} data in the reshaped variables, even if that means dropping values of X_i variables and individuals i , use **drop all**. See section 2 for examples. The **missing()** option has an effect only with **sreshape long**.

widevars(all|nonmissing) specifies that either all X_{ij} variables be created in wide form (the default) or only X_{ij} variables containing one or more nonmissing values be created. The **widevars()** option has an effect only with **sreshape wide**.

Rarely used options

`nocompress` specifies that X_{ij} wide variables may not have their data types compressed to save space. The default is to compress the variables when possible to achieve the most efficient data format. The **`nocompress`** option has an effect only with **`sreshape wide`**.

`norecastasdouble` specifies that when long-integer variables and float variables share the same stubname and are being reshaped from wide to long to have their values contained within a single variable, the resulting variable should always be of type float instead of double. The default, as in Stata's **`reshape long`** command, is to use type double wherever necessary to ensure that large-integer values are still exactly represented after reshaping (specifically, when the long variable contains numbers whose absolute value exceeds 16,777,216). The **`norecastasdouble`** option has an effect only with **`sreshape long`**.

`atwl(string)` specifies how to treat stubnames that contain a special character, “@”. In the command, **`sreshape long`** or **`sreshape wide`** is followed by a list of stubnames, that is, the beginnings of the X_{ij} variable names, for variables reshaped between wide and long forms. Stata's **`reshape`** command and **`sreshape`** allow “@” in the stubnames to indicate where the j values should appear in variable names, instead of the default behavior of putting the j values at the end. Stata's advanced **`reshape`** command also allows replacement of “@” in long-form X_{ij} variable names using the **`atwl()`** advanced method. **`sreshape`** honors advanced-form declarations of **`atwl()`**, as with **`reshape`**, and the **`atwl()`** setting is carried forward from the last usage of **`reshape`** or **`sreshape`** (this is awkward but maintains compatibility with **`reshape`**). The **`atwl(string)`** option declares a new string, or **`atwl(" ")`** replaces the string with null (note that **`atwl("")`** does not change the string because it means no change; **`atwl(" ")`** must be used instead).

`favorspace(0|1|2|3|4)` requests that **`sreshape`** use slower methods that require less memory. The default, **`favorspace(0)`**, is usually fastest. The exception is when the computer's available memory is exceeded. Methods 1 or greater preferentially use plugin programs that are often more space efficient. A Java plugin is used if the computer has an installed copy of the Java Runtime Environment.¹ Methods 2, 3, and 4 reduce memory usage when plugins cannot be used. Then the larger-numbered methods may yield progressively greater memory savings at the cost of more intensive computation. Specifically, methods 2, 3, and 4 make a difference only if reshaped variables are of type **`byte`** with method 2, **`byte`** or **`int`** with 3, or **`byte`**, **`int`**, or **`long`** with 4.² The option causes the values of the reshaped variables

1. Java may already be installed on your computer for Internet browser use, but if not, it is a free download. Alternatively, a compiled C plugin can also be used and is preferred if available because it is slightly faster than the Java plugin. However, you need to compile the relevant C code for your computer and run a test to ensure it functions properly (place the compiled plugins in your ado-directory and uncomment the three lines in the ado-file following the text “* Define C plugin programs”).

2. Squeezing **`byte`**, **`int`**, and **`long`** variables also helps when a C plugin but no Java plugin is available and the number of observations exceeds 2,147,483,646 (because Stata's C plugin interface does not work for observation numbers exceeding this value).

when stored in Mata to be compressed into strings instead of being stored in arrays of `doubles`, resulting in an 87.5% saving of space for `bytes`, 75% saving for `ints`, or 50% saving for `longs`.³

`nopreserve` is a programmer's option. It prevents the initial data from being preserved, saving a little time, but causing the data to be left disarranged if `sreshape` is terminated with the *Break* key or by an internal error. This option may be useful if `sreshape` is used inside a command that preserves the data before calling `sreshape`.

Denigrated option

`keeponlyxi(varlist)` should probably not be used. It specifies that only the specified X_i variables (variables constant within i but possibly varying across i , other than the list in `i(varlist)` or `newi(newvar)`) be kept; all other X_i variables are dropped. This option may be useful to maintain compatibility with prior work in which the advanced-format command `reshape xi` was used. Specify the same list of variables here as was used with `reshape xi`. Otherwise, prior settings of the list of X_i variables to keep, including advanced format settings with `reshape xi`, cause an error message, warning that instead the user should explicitly drop undesired variables.

3.2 Other traits of the command

Interchangeability with Stata's reshape command

The `sreshape` command can be used interchangeably with Stata's `reshape` command. Behind the scenes, information needed to reshape between wide and long form is stored in the form used by `reshape`. Therefore, one can, for example, reshape from wide to long form using `sreshape` and then go back to wide form using `reshape`. However, sparse treatment of the data requires the use of `sreshape`.

If there is an error when reshaping the data, `sreshape` stores information needed to allow diagnosis of the error. The `sreshape error` or `reshape error` command, when used after a problem, will report on any apparent problems with the dataset and with how it was to be reshaped.

Advanced form of the command

Stata's `reshape` command supports an advanced usage, in which the user first declares i and j variables, plus the list of stubnames for the X_{ij} variables to be reshaped and any other options, and then simply types the `reshape long` or `reshape wide` command. For full compatibility, `sreshape` honors such information declared in the advanced format with `reshape` or `sreshape` with one exception. The `reshape xi` advanced usage is not supported, because it could cause unintentional dropping of variables (however, see

3. This data compression comes at a substantial speed penalty, which is greater for `int` variables than for `byte` variables and is still greater for long variables.

the denigrated `keeponlyxi(varlist)` option if this is really needed). In contrast, the new options to `sreshape` must be typed as options on the same line with the `sreshape long` or `sreshape wide` command (no “advanced” usage is supported for the new options).

Formats and labels preserved

The `sreshape` command preserves display formats, value labels, and variable labels when reshaping from long to wide form and also when reshaping from wide to long form as long as the formats or labels are consistent across the set of variables combined to make a long-form variable. An exception applies for stubnames that contain the character “@” when reshaping to wide form; as in `reshape`, the variable label then takes the form “<j-value> <long-form-variable-name>”. Notes and characteristics, which might lose their meanings when the data are transformed, are never preserved for X_{ij} variables.

Stored results

The `sreshape` command stores the same results as `reshape`. It also stores the following scalars. `r(k)` is the resulting number of variables, and `r(N)` is the resulting number of observations. `r(changed)` is 1 if the dataset has been changed (reshaped) successfully, or 0 if not. `r(width)` is the number of bytes occupied in the dataset for each observation.

Official Stata commands for equivalent treatment of missing data

`sreshape`’s missing data treatments could also be achieved using official Stata commands. This is illustrated here for the case of a single long-form X_{ij} variable.

As the introduction noted, the equivalent is simple in the case of `missing(drop all)`. The command

```
sreshape long a, i(id) j(t) missing(drop all)
```

accomplishes the same result as

```
reshape long a, i(id) j(t)
drop if missing(a)
```

With multiple long-form X_{ij} variables, an observation should be dropped only if all its X_{ij} variables have missing values.

The case of `missing(drop)` is more complicated, requiring several steps. The command

```
sreshape long a, i(id) j(t) missing(drop)
```

accomplishes the same result as

```
reshape long a, i(id) j(t)
```

```

qbys id (t): egen kept = min(cond(!missing(a),t,.))
qbys id (t): replace kept = t[1] if kept>=.
keep if (!missing(a)) | t==kept
drop kept

```

With multiple long-form X_{ij} variables, the second and fourth lines above need to be modified to check whether all long-form X_{ij} variables have missing values. Because of the complications and the potential for mistakes and inadvertent loss of data, it is preferable to carry out this process using the single **sreshape** command.

4 Examples

The **sreshape** command may be used just like **reshape**; simply insert an “s” before the **reshape** command in the Stata documentation. For example, the following replicates a basic example, starting with wide-form variables **inc80**, **inc81**, **inc82**, **ue80**, **ue81**, and **ue82** and converting them to long-form variables **inc** and **ue** (with observations for each individual in each year 80, 81, and 82) and then back again:

```

webuse reshape1
list
sreshape long inc ue, i(id) j(year)
list, sepby(id)
sreshape wide
list

```

If the original variable names have numbers in the middle of text, use an @ sign in place of the number:

```

webuse reshape3, clear
sreshape long inc@r ue, i(id) j(year)

```

If the j variable values are strings, you must specifically allow strings, as in the following example that transforms wide variables **incm** and **incf** to long variable **inc**, with a new **sex** variable equal to “f” or “m” for a female or a male, respectively:

```

webuse reshape4, clear
sreshape long inc, i(id) j(sex) string

```

Strings in i , X_{ij} , and X_i are handled automatically, regardless of the **string** option; only use **string** if the j variable contains strings.

The above shows typical usages similar to **reshape**, but now consider new options.

With sparse data, you may wish to keep only those long-form observations that have nonmissing X_{ij} data. If missing data exist for the second individual in year 80, the first example above yields one fewer observation:

```

webuse reshape1
replace inc80 = . in 2
replace ue80 = . in 2
sreshape long inc ue, i(id) j(year) missing(drop)

```

If only one of `inc80` and `ue80` is replaced with a missing value, then the observation is not dropped because it still contains information about an X_{ij} variable.

If the X_{ij} data are missing for all j values, then one observation will still be kept for each individual. To continue the above example, type

```
webuse reshape1
replace inc80 = . in 2
replace inc81 = . in 2
replace inc82 = . in 2
replace ue80 = . in 2
replace ue81 = . in 2
replace ue82 = . in 2
sreshape long inc ue, i(id) j(year) missing(drop)
list, sepby(id)
```

When you retain one observation for the second individual, the information that sex equals 1 for this individual is not lost. If the values of such X_i variables are not needed unless there is X_{ij} data, then all observations for the second individual could have been dropped using the following:

```
sreshape long inc ue, i(id) j(year) missing(drop all)
```

In the latter case, after you type `sreshape wide` to convert the data back to wide form, the second individual will no longer exist in the wide dataset.

When converting data from long form to wide form, you can avoid creation of X_{ij} wide variables that would be filled entirely with missing values:

```
sreshape wide inc ue, i(id) j(year) widevars(nonmissing)
```

If an i variable does not previously exist in a wide-form dataset, you might wish to use the observation number to create an i variable. As a convenience, you can create the variable and reshape the data to long form in one step,

```
sreshape long inc ue, newi(inum) j(year)
```

where `inum` is a new variable.

The resulting X_{ij} variables may have new data types because `sreshape wide` compresses X_{ij} variables when possible to save space without loss of information. This behavior can be turned off:

```
sreshape wide inc ue, newi(inum) j(year) nocompress
```

As discussed below, the ideal implementation of `sreshape` requires some implementation internal to StataCorp. In the meantime, users might not achieve speedups with datasets that take up a large fraction of the computer's memory. This is because the data must be duplicated temporarily in the computer's memory. During the data duplication, Stata may not free the memory previously used for the dataset, and variables may not be stored as efficiently as in the Stata dataset.

5 Reliability and performance

To assess quality, programmatic tests measured reliability and performance. Reliability tests evaluated both the subroutines and plugins used to store and retrieve data outside the Stata dataset and overall results of the **sreshape** command. Subroutine evaluations considered all different types of variables that can be stored in Stata, missing and extended missing values, numbers of observations currently only possible in Stata/MP 14, and methods of data storage and retrieval that depend on Mata (Stata's matrix programming language), Stata's Java plugin interface, and Stata's C plugin interface. The subroutine evaluations revealed problems of the current Java and C plugin interfaces, requiring that the program be designed to prevent the use of the plugins in certain circumstances or for certain data types, and they revealed relative speeds that influenced program design.

Tests of the overall **sreshape** command were carried out for small and large datasets with a wide range of data types. Results were computed using both **reshape** and **sreshape**, and the outcomes compared to ensure exactly identical results. In analyses with sparse data treatments, the **reshape** command was followed by dropping observations as appropriate to allow valid tests. In either case, checksums (using Stata's **checksum** command) were computed and compared to ensure identical values. In all cases, the tests indicate identical behavior.

Speed tests likewise were carried out on two large datasets.⁴ The tests address a range of variable types likely to be in typical data. A first dataset analyzed consisted of entirely numeric variables of all numeric data types (with 285,874 observations, 200 j values, and the average observation having nonmissing X_{ij} data for 14.2 j values). A second dataset consisted of a mix of numeric and string variables with string variables occupying almost all the dataset memory (with 285,874 observations, 26 j values, and the average observation having nonmissing X_{ij} data for 2.3 j values). Both datasets were transformed from wide to long form 1) using **reshape** and 2) using **sreshape** a) without and b) with the **missing(drop)** option. After transformation to wide form with the **missing(drop)** option, both datasets were transformed back to long form 3) using **reshape** and 4) using **sreshape**.

Results of the speed tests are summarized in table 5 for the two datasets on several types of computers. Test results varied across the computers used for testing, two Windows and two Mac computers, although speedups occurred on all machines. The wide-to-long transformations were sped up the most by using **sreshape**. For the numeric dataset, **sreshape** was 14.7 to 31.0 times faster than **reshape** when computing identical results, and **sreshape** was 25.3 to 53.6 times faster when it dropped observations with missing X_{ij} data using the **missing(drop)** option. For the string dataset, **sreshape**

4. To allow easy distribution of the datasets without violating intellectual property ownership, I used artificial datasets that were generated using a do-file. See the do-file of reliability and performance tests distributed with the program.

was 7.7 to 15.4 times faster when computing identical results and 11.4 to 23.0 times faster when using the `missing(drop)` option.⁵

Table 5. Run times of `sreshape` and `reshape` with alternative datasets, in seconds

	PowerMac 2008 ^a	Windows on Dell Optiplex 990 2012 ^b	Windows on Dell Precision T7500 2012 ^c	MacBook Pro 2014 ^d
Wide to long				
Numeric dataset				
1. <code>reshape</code>	4408.8	1178.1	1802.7	1126.8
2a. <code>sreshape</code>	142.3	80.0	77.1	75.0
2b. <code>sreshape, drop</code>	82.3	35.5	51.7	44.6
String dataset				
1. <code>reshape</code>	819.8	157.1	199.5	135.4
2a. <code>sreshape</code>	53.4	17.0	22.7	17.6
2b. <code>sreshape, drop</code>	35.7	9.8	13.7	11.9
Long to wide				
Numeric dataset				
3. <code>reshape</code>	1130.8	139.0	222.3	152.0
4. <code>sreshape</code>	141.5	57.6	86.0	65.0
String dataset				
3. <code>reshape</code>	589.0	52.1	71.6	54.8
4. <code>sreshape</code>	45.9	20.3	31.7	20.1
Sample size	80	440	60	60

Notes: Times reported are means for each task, after an initial burn-in computation whose time is thrown out to reduce error. All runs used Stata/MP 14.0 with the number of processors supported equal to the number of cores, except that on the third machine the number of processors used was restricted to one using command `set processors 1`.

^a8-core 2.8 GHz, spinning disk, 32 GB

^b4-core 3.4 GHz, spinning disk, 8 GB

^c12-core 2.66 GHz, spinning disk, 192 GB

^d4-core 2.2 GHz, flash drive, 16 GB

5. The standard error for the number of times faster is 0.2 for the number 23.0 (wide-to-long transformations with string dataset using `missing(drop)`) and 0.1 or less for all other numbers. Standard errors were computed by the delta method (using Stata's `nlcom` command) after estimating the means in table 5.

The long-to-wide transformations saw more modest speed improvements. For the numeric dataset, **sreshape** was 2.3 to 8.0 times faster than **reshape** when computing identical results. For the string dataset, **sreshape** was 2.3 to 12.8 times faster when computing identical results.

Speedups are similar when one uses entirely nonmissing data, comparing **reshape** and **sreshape** without the **missing(drop)** option.

By building on the code in **sreshape**, StataCorp can quite simply achieve substantially greater speed improvements. These improvements are not currently possible using ado-files, Mata, or plugins, but instead require access to the internal computer code for Stata.

6 Possible improvements by StataCorp

Stata's **reshape** command has a long history and deserves credit as a core tool of the program. However, it was designed at a time when the methods used here were not practical, so it is no surprise that an improved version is now feasible. The **sreshape** command can easily be improved further by StataCorp to yield substantially greater speed and lower memory usage.

The improvements require programming within Stata's base code. To understand this, let's consider the alternatives. **sreshape** moves data temporarily out of the Stata dataset and deletes variables or observations; then it creates new observations or variables and puts the data back into the dataset. One might consider shuffling the data within the dataset instead; a within-dataset approach is described in the *Appendix*, but the within-dataset approach turns out to have disadvantages that make **sreshape**'s approach preferable. **sreshape** uses Mata or plugins as places to store data, yielding performance gains that reach approximately the limits of what is now possible in Stata with these approaches.

6.1 Plugins and Mata

Stata provides plugin interfaces for Java and C programmers. The currently available interfaces (as of Stata 14.0) have limitations that affect when performance gains are possible. The Java interface alters some string values, making it unsuitable for variables of type **str#** or **strL**, and the Java interface depends on a user having installed Adobe's Java software, which is free but has been implicated with security concerns. The C interface does not allow more than 2,147,483,646 observations (one fewer than most flavors of Stata allow), whereas Stata/MP 14 allows many more observations, the C interface precludes access to variables of type **strL**, and compiling C plugins to serve multiple computer types can be extremely challenging. The currently available interfaces also impose a speed penalty on copying data, particularly if the data are to be represented efficiently in memory (for which detection and representation of missing-value codes creates special challenges).

Using Mata is fast but can require more memory. Mata stores all numbers as doubles (taking up 8 bytes per number), so storing **byte** (1 byte per number), **int** (2 bytes), **long** (4 bytes), or **float** (4 bytes) data in Mata requires 8, 4, 2, or 2 times as much memory as needed. The **byte**, **int**, and **long** data types can be compressed into strings and stored with efficient use of space with the **favourspace()** option, but this requires an order of magnitude longer to convert byte data to string characters and store them, and the speed penalty is worse for **int** and especially long data.⁶ Storing strings in Mata may use memory less efficiently than Stata if there are strings of constant length (a pointer to a string may take up more memory than the string itself), or particularly for **strL** data with many repeated (especially very long) values.⁷

6.2 Why further implementation should be internal to Stata

Speeding up Stata's **reshape** command requires a means to copy data efficiently between places, using the computer's fast internal (random access) memory. Stata's primary built-in data storage mechanism, the dataset, is quite efficient and allows copying between locations, but (as the *Appendix* addresses) it is not ideal for efficient reshaping. This leaves a need to use data storage locations other than variables in the dataset. The obvious possibilities are plugins and Mata, but these have the substantial limitations described above.

6.3 Potential further implementation by StataCorp

A substantial speedup of **sreshape** requires only minor programming by StataCorp. Storing and retrieving data take the majority of execution time for **sreshape** with big datasets: this involves simply copying variables, in all or selected observations, to a temporary storage location, then copying them back when needed. New code can replace **sreshape**'s subroutines that store and retrieve data with built-in code that copies data as a block. (**strL** data might require special treatment, depending on how **strL** values are recorded when in a single long variable versus many wide variables.) The changes could alternately be implemented through Mata methods to store and retrieve data in efficient arrays (**byte**, **int**, **long**, **float**, **strL**, and perhaps fixed-length-string **str#**).

These simple changes should yield major improvements in speed and memory efficiency and a further—albeit more complicated—improvement in memory efficiency is also possible. Stata's total memory usage during a **reshape** could nearly double because Stata's dataset storage space does not contract for some time, even though variables or observations are dropped, while additional memory is used to store copies of parts

6. To store **byte**, **int**, or **long** numbers in strings in Mata, one can convert each number, respectively, to 1, 2, or 4 integers each in the range from 0 to 255 and convert the resulting numbers to characters. Missing value codes must be detected and given numeric values, and arithmetic computations are required. This is slow. If Mata had functions to efficiently store arrays of the same data types used in Stata datasets, these speed penalties could be mitigated, and float numbers could also be stored efficiently (a possible alternative is specialized functions to convert Stata data, including missing-value codes, to and from strings of characters).

7. A Mata method to store an array of **strL** values in their native format may be helpful.

of the data. In fact, however, memory usage need not expand much. Regions of memory allocated to the Stata dataset but temporarily unused can be commandeered for a short period to store copied data. During this time, the dataset space can be temporarily prevented from contracting. The stored copies can be placed strategically so that information needed soon, when deleted, makes room for expansion of the real data.

These improvements would not be the first time a user-suggested improvement to **reshape** was incorporated into Stata: previous improvements by Weesie (1997) are now part of Stata's base implementation. The **sreshape** command is fully compatible with these improvements as implemented in Stata. The command also preserves variable labels, as does a pair of add-on commands, **reshape8** and **reshape7**, by Rising (2003, 2002).

7 Acknowledgments

This work began during efforts to analyze patent data on light-emitting diodes and LED lighting and hence was partly funded by two sources. This work was supported primarily by the Engineering Research Centers Program (ERC) of the National Science Foundation under NSF Cooperative Agreement No. EEC-0812056 and in part by New York State under NYSTAR contract C090145. I thank an anonymous reviewer for comments.

8 References

- Rising, B. 2002. **reshape7**: Stata module to provide improved reshape. Statistical Software Components S427501, Department of Economics, Boston College. <http://econpapers.repec.org/software/bocbocode/s427501.htm>.
- . 2003. **reshape8**: Stata module to reshape while preserving variable labels. Statistical Software Components S436202, Department of Economics, Boston College. <https://ideas.repec.org/c/boc/bocode/s436202.html>.
- Weesie, J. 1997. dm48: An enhancement of **reshape**. *Stata Technical Bulletin* 38: 2–4. Reprinted in *Stata Technical Bulletin Reprints*, vol. 7, pp. 40–43. College Station, TX: Stata Press.

About the author

Kenneth L. Simons is an associate professor of economics at Rensselaer Polytechnic Institute.

Appendix 1 Reshaping through successive iterations

The **sreshape** command temporarily stores data outside the Stata dataset while reshaping. Data could instead be reshaped solely within the dataset to allow programming in a do-file without storing data elsewhere. However, this introduces major programming complications if wide data of different types (such as **byte** and **int**) are being reshaped

to create a single long variable or if the programming is to take advantage of the fact that some observations can be dropped to reduce the size of the reshaping job. Also, it requires extra steps to copy data from one place to another, to shuffle data between places as the dataset changes form. Thus I used an outside-the-dataset approach in creating **sreshape**; moreover, the approach used in **sreshape** can readily be taken further by StataCorp to yield an approach that consumes relatively little more memory than the dataset requires. To allow users to understand why the within-dataset approach is not ideal, I describe such an approach to reshaping, termed “reshaping through successive iterations”, that is deliberately not used in **sreshape**.

A.1 Basic approach with wasted space

The “reshaping through successive iterations” method, which again is not used in **sreshape**, rearranges the data between wide and long shapes through a series of intermediate stages. A set of stages between wide and long shapes is depicted in figure 1 for the case with no observations dropped and three values of j . The shapes have N observations in wide form, $2N$ observations in intermediate form, and $3N$ observations in long form, indicated by the numbers 0, N , $2N$, and $3N$ on the left side. The data in each observation include the i , j , and X_i variables, plus the X_{ij} variables, as noted along the top. In wide form, the data fit within the solid rectangle at the top of the figure. The wide form initially does not have a j variable, but j is created when the reshaping is about to begin; hence, it is included in the depiction of initial memory usage. The wide-form data thus consist of the i , j , and X_i variables in the left column, plus three sets of X_{ij} variables for the three values of j . The data’s memory usage per observation is the width of these variables, that is, the number of bytes that contain their numeric and string data, and multiplying by the number of observations N gives the data’s total memory usage. Hence, memory usage is proportional to the area of the solid rectangle.

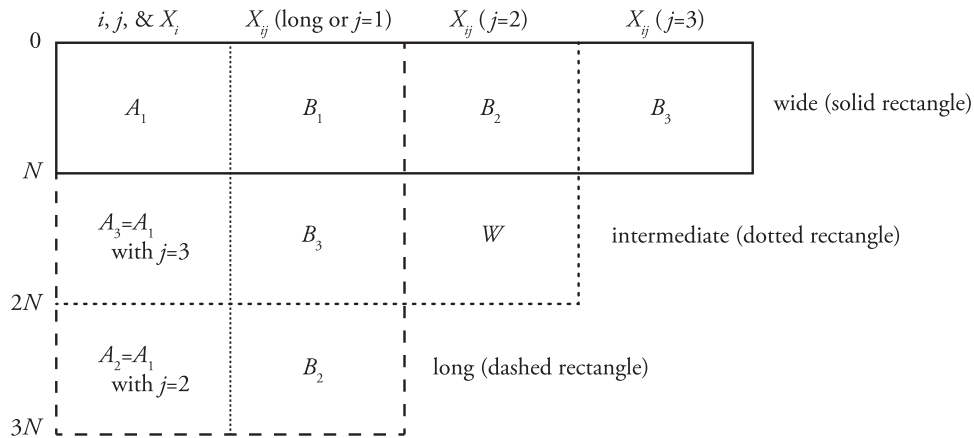


Figure 1. Stages from wide shape (solid rectangle) to long shape (dashed rectangle)

A simple intermediate step toward converting the data to long form is to 1) add N more observations; 2) copy the i and X_i data to the same variables in the new observations and set $j = 3$ in the new observations so that the data A_3 equal the data A_1 but with $j = 3$; 3) copy the X_{ij} data for $j = 3$ from observations 1 to N to the variables with X_{ij} data for $j = 1$ in observations $N + 1$ to $2N$ so that B_3 is copied from the upper right to the region of data below B_1 ; and 4) drop the variables with X_{ij} data for $j = 3$ at the right. This yields the intermediate shape of data, outlined by the dotted rectangle, with memory usage proportional to the area of the rectangle. The region W is wasted space, which could be put to better use when there are more values of j as discussed below.

Following the intermediate step, a last transformation puts the data into long form. The last step adds N more observations, copies the i and X_i data to the new observations and sets $j = 2$ in these observations, copies the X_{ij} data for $j = 2$ in observations 1 to N to the variables with X_{ij} data for $j = 1$ in the new observations, and drops the X_{ij} variables for $j = 2$. This yields the long shape of the data, outlined by the dashed rectangle, and with memory usage proportional to the area of the rectangle. After some sorting and renaming of variables, this is the long-form data familiar to Stata users.

A.2 Memory usage with wasted space

The idea extends straightforwardly to cases where the number of j values, J , is any integer greater than one. However, as J increases, the peak memory usage would become excessive without better utilizing the wasted space regions. Memory usage without using the waste regions can be computed for each step s from long to wide form (so $s = 1$ is long form and $s = J$ is wide form) as

$$M(s) = (w_A + sw_B)N(J - s + 1)$$

where w_A is the width of variables i , j , and X_i combined and w_B is the width of the X_{ij} variables for a single value of j . Peak memory use occurs when $s = 1$ if $w_B \geq w_A/(J-1)$ but otherwise can be estimated by treating s as continuous, yielding peak usage

$$M(s^*) = \frac{(w_A + w_B + w_B J)^2 N}{4w_B}$$

where s^* is the value of s that maximizes $M(s)$. Comparing $M(s^*)$ with the memory usage in long form, which is $M(1) > M(J)$, one finds that as J becomes large, holding w_A and w_B constant,

$$\frac{M(s^*)}{M(1)} \approx \frac{w_B}{4(w_A + w_B)}(J + 2)$$

This reveals that in some intermediate stage, for large J , the memory usage would reach many times that needed to store the long-form data; if J were multiplied by 10, the long-form data would take 10 times as much memory, but the reshaping would take about 100 times as much memory. In practice, memory can be conserved by dropping observations and variables with missing data and compressing variables with small values; nonetheless, it is sometimes crucial to avoid wasted space.

A.3 Eliminating wasted space creates disadvantages

The wasted-space problem can be greatly reduced by filling that space with X_{ij} data, allowing much more compact intermediate forms of data. When one goes from wide to long form, whenever N new observations are created and column $s + 1$ of X_{ij} variables are about to be dropped, there are $s - 1$ columns of X_{ij} variables for $j = 2, \dots, s$ that would be new wasted space in those N new observations. This space could be filled with X_{ij} data from columns $j = s, \dots, 2$ of the first N observations. Various algorithms to transform the data in (for large problems) much fewer than $J - 1$ steps could be designed.

This method would be somewhat straightforward with no sparse data, if the types of the X_{ij} variables remain the same across all j values. However, departures from these assumptions introduce major complications because straightforward copying between the blocks of X_{ij} data illustrated in figure 1 requires blocks with a sufficient number of observations and the right variable types. For example, if a wide dataset includes variables `inc1`, `...`, `incJ`, which are to be combined into a single variable, it may be that `inc1` is of type `byte`, `inc2` is of type `float`, `inc3` is of type `long`, and so forth, preventing data from readily being copied from one column to another. Recasting numeric variables to double and string variables to the largest string type in the data would solve the problem but brings back the potential for heavy memory usage. Ordering the variables so that information is copied from bigger- to smaller-format variables is possible. However, a different order might be needed from the standpoint of having a sufficient number of observations to copy into and may require a different ordering for different X_{ij} long variables, which again increases memory requirements when going from long to wide forms: observations cannot be dropped until information on all X_{ij} variables has been copied out of them. Moreover, even if these complications did not arise, this method would require more copying of information to-and-from than (and require at least as much storage space as) the approach described in the text, giving it inherently less potential speed gains—and much greater complexity—than `sreshape`'s approach.