The Stata Journal (2023)
23, Number 3, pp. 884–896 DOI: 10.1177/1536867X231196519

Speaking Stata: Replacing missing values: The easiest problems

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

Abstract. Missing values are common in real datasets, and what to do about them is a large and challenging question. This column focuses on the easiest problems in which a researcher is clear, or at least highly confident, about what missing values should be instead, implying a deterministic replacement. The main tricks are copying values from observation to observation and using the `ipolate` command. Both may often be extended simply to panel or longitudinal datasets or to other datasets with a group structure, such as data on individuals within families or households. This column includes how to satisfy constraints that interpolation is confined to filling gaps between values known to be equal or to observations moderately close to a known value in time or in some other sequence or position variable.

Keywords: dm0113, missing values, interpolation, panel data, longitudinal data. reversing time, ipolate, by: prefix

1 Introduction

Missing values are common in real datasets, especially those with many variables or observations. There are just so many reasons why a value may not be on hand. An entry in a data cell of "not available" would often be a euphemism of some kind for (say) person refused to answer, patient too ill, true conditions in that country are a matter of guesswork, firm ceased trading, machine broke, gauge overtopped by flood, observer forgot to write it down, or whatever. What to do about missing values is thus a difficult and challenging question, with answers varying in complexity up to multiple imputation. This column focuses on the easiest problems, in which a researcher knows, or at least is highly confident about, what the answer should be as a nonmissing value. The operations needed all have deterministic flavor, without a whiff of probabilistic variation.

There are two main devices at this easy end to fill in missing values: copying down or forward from observation to observation and using interpolation in its simplest form, namely, linear interpolation as supported by `ipolate` (see [D] **ipolate**). Even brief introductions to interpolation within texts on numerical analysis go much further (for example, Morton [1964]; Wilkes [1966]; Hamming [1973]; Lancaster and Šalkauskas [1986]; and Press et al. [2007]). If the topic of interpolation interests you, then you should enjoy a splendidly detailed historical review by Meijering (2002).

These problems arise commonly from the very start of anybody's research with data yet are rarely discussed in statistics texts or courses. One reason for that could be a cultural divide between statistics and numerical analysis making it so that interpolation as a deterministic problem is seen as beyond statistics. Early texts on combination or calculus of observations (for example, Whittaker and Robinson [1924]) and a few statistical texts (for example, Bowley [1901]; Yule [1911]; Davis [1973]; and Pollard [1977]) do bridge the gap in various ways. Another reason could be that details about data quality are somewhere between mildly irritating and highly embarrassing but either way not of central interest. Yet another reason could be that the technique needed seems too obvious or trivial to deserve discussion—once it is understood!

Three simple general points deserve emphasis right now.

Think about infilling graphically. Plotting the data and the results of replacement can give an idea of what to do and of whether what you did makes sense.

Copy original data and work on a clone. When doing this for real with serious datasets, work with copies of the original variables. `clonevar` (see [D] **clonevar**) is convenient here. It copies information on variable and value labels, too, among other details. Other way round, overwriting existing variables is dangerous because you might mess them up and because it is always prudent to keep the original data as they came. Some awkward supervisor or reviewer might insist on a comparison using those original data. To keep code examples simple, we will not always do this in this column, but good practice is a case of "do as we say, not as we do".

Fix any problem with absent values first. The methods here do not themselves fix implicit gaps in the data, a problem better described as one of absent values. Concretely, suppose you have annual values for 2016 and 2020 and wish to infill values for 2017, 2018, and 2019. Then first use a suitable command such as `expand`, `fillin` (see [D] **expand**, [D] **fillin**) (Cox 2005), or `tsfill` (see [TS] **tsfill**) to add the extra observations with missing values, which can then be `replace`d.

Related columns include Cox (2002, 2011) and Cox and Schechter (2019).

2 Copying within blocks

A good place to start is when observations occur in blocks but some variable, often an identifier or category variable, is given only for the first observation in each block. Here "observation" in Stata, as usual, means what in other software or from other points of view is regarded as a record, row, or case. In, say, a spreadsheet, it can be tacit that a value applies to rows below until the next such value. In Stata, and in statistical software generally, such values need to be present explicitly.

Working backward, the technique to be explained can be exploited in advance, especially if you or someone working for you is typing in data.

Let's assume that we are typing in data for some females and some males in blocks of each. We have decided in advance that we will code the difference as 1 for female and 0 for male and will use `female` as the variable name. See Cox and Schechter (2019) for a recycling of long-standing advice to name an indicator or dummy variable for the category coded with 1. We are committed to define value labels in due course, but set that aside. So we need to enter 1 and 0 just once each, and that gets us to a layout like that produced as follows:

```
. set obs 10
Number of observations (_N) was 0, now 10.

. generate female = 1 in 1
(9 missing values generated)

. replace female = 0 in 6
(1 real change made)

. list female

     ┌────────┐
     │ female │
     ├────────┤
  1. │      1 │
  2. │      . │
  3. │      . │
  4. │      . │
  5. │      . │
     ├────────┤
  6. │      0 │
  7. │      . │
  8. │      . │
  9. │      . │
 10. │      . │
     └────────┘
```

In short, each nonmissing value is to be copied to replace the missing values below, stopping at the next nonmissing value or at the end of the dataset. Here is a good answer for this specific problem:

```
. replace female = 1 in 2/5
(4 real changes made)

. replace female = 0 in 6/L
(4 real changes made)
```

Note that `L` is worth remembering as a way to specify the last observation. Indeed, if you thought of that, it would be fair to point out that something like

```
. generate female = 1 in 1/5
(5 missing values generated)

. replace female = 0 in 6/L
(5 real changes made)
```

would have been a better way to start. This method, however, does not generalize well. The problem can be seen by imagining any more complicated setup with several such blocks, so the same remedy would mean typing several such commands. That could be error prone, tedious, and repetitive.

Now and henceforth: If your taste runs to abbreviating command names that can be abbreviated, then you do not have to type out the full command name, `generate`.

Curiously or otherwise, on the evidence I have seen, most experienced Stata users type `gen`, evidently finding the minimal abbreviation `g` a little too cryptic, at least in code intended to be easily readable by others. One well-known Stata user customarily writes `gene`, but he did start out as a biologist.

There is a better and more general solution:

```
. replace female = female[_n - 1] if missing(female)
```

In this kind of statement, `missing(female)` would be replaced by `female == .` if you needed that more restrictive condition. The difference is that `missing(female)` includes any of the extended numeric missing values `.a` through `.z` as well as `.`, the so-called system missing value. For more on missing values, start with `help missing`.

An advantage of the `missing()` syntax is that it carries over to missing string values, namely, empty strings `""`.

Let's back up. How does that command work? In the data segment just given, the first missing value is in observation 2, so `_n - 1` (observation number minus 1) is there evaluated as 1, so the command is equivalent to

```
. replace female = female[1] in 2
```

So `female[2]` becomes a copy of `female[1]` with the same value, namely, 1. The next observation with a missing value is in observation 3, so the command becomes equivalent to

```
. replace female = female[2] in 3
```

But we just changed the value of `female[2]` to 1. So the value 1, just copied from observation 1 to observation 2, is copied once more to observation 3. In short, because Stata evaluates commands in observation order (Newson 2004), this sequence will continue within the first block of observations. What is more, the same kind of sequence is repeated in the next block of observations. Copying is a cascading operation in this sense, and there is automatically a loop over observations.

3 Panel data or other group data

The structure seen in the previous section is quite common, as is some kind of simple twist on it in which observations are for a set of panels or other groups. Panel or longitudinal datasets often have structures defined jointly by an identifier variable and a time or sequence variable. Similar problems often arise with spatial data with one spatial dimension—going up into the atmosphere, down below the surface, or along a profile or transect. Other kinds of grouped data often have identifiers at two or more distinct levels, say, family and individual.

Our structure might resemble that from

```
. clear
. set obs 20
Number of observations (_N) was 0, now 20.
. egen id = seq(), block(10)
. egen time = seq(), to(10)
. generate value = _n if time == 1
(18 missing values generated)
```

If the problem is to copy the first value forward within each group, we need something more like

```
. bysort id (time): replace value = value[1]
(18 real changes made)
```

or indeed

```
. bysort id (time): replace value = value[_n-1] if missing(value)
```

The tool of choice here is the `by:` prefix command. If this is unfamiliar, check out the manual entry [D] **by** and its references, including Cox (2002).

What may look to be a different problem—copying one particular value to the rest of a group—turns out to have a very similar solution. We just need to sort that value to a known position, say, so that it becomes the first or last observation in each group. Then we have an easier problem. Suppose that we have family data and want to copy the mother's age to all observations for a family; we might want to find out how much older or younger everyone else is. A good step forward is to have or get an indicator variable for being a mother.

To make this concrete, consider this toy dataset with family identifier `fam_id`, individual identifier `ind_id`, `age`, and `role`, where for the last variable 1 means father, 2 means mother, and 3 means child. If you have ever worked with such data or just thought about families you know, you will realize that real data can be much more complicated, but we need to start somewhere. If you wish to experiment yourself, the dataset is bundled with the media for this column.

```
. use family_example, clear
. list

     +-------------------------------+
     | fam_id   ind_id   age   role  |
     |-------------------------------|
  1. |      1        1    45      1  |
  2. |      1        2    42      2  |
  3. |      1        3    18      3  |
  4. |      1        4    15      3  |
  5. |      1        5    13      3  |
     |-------------------------------|
  6. |      2        1    29      2  |
  7. |      2        2    26      1  |
  8. |      2        3     5      3  |
     +-------------------------------+
```

First, get an indicator with values 0 and 1:

```
. generate is_mother = role == 2
```

If we `sort` on that indicator variable within families, each mother is sorted to last, and we have a way to refer to such observations.

```
. bysort fam_id (is_mother) : generate mother_age = age[_N] if is_mother[_N]
```

Look at the results:

```
. list

     ┌───────────────────────────────────────────────────────┐
     │ fam_id   ind_id   age   role   is_mot~r   mother~e    │
     ├───────────────────────────────────────────────────────┤
  1. │      1        3    18      3          0         42    │
  2. │      1        5    13      3          0         42    │
  3. │      1        4    15      3          0         42    │
  4. │      1        1    45      1          0         42    │
  5. │      1        2    42      2          1         42    │
     ├───────────────────────────────────────────────────────┤
  6. │      2        2    26      1          0         29    │
  7. │      2        3     5      3          0         29    │
  8. │      2        1    29      2          1         29    │
     └───────────────────────────────────────────────────────┘
```

The code gestures toward more complicated setups by adding `if is_mother[_N]`, equivalent to `if is_mother[_N] == 1` in practice. If no mother is present in a family, the indicator value will be 0 for all observations in that family.

If you are puzzled here by the use of `_N`, the principle is that when using `by:`, `_N` refers to the number of observations in each block and thus to the last observation in each block. If there are five observations in a block, the last is the fifth observation.

A trick you may prefer is to have an indicator with values −1 for true and 0 for false. Then, on sorting, the "true" observations go first. Alternatively, you might like to think about using `gsort` to sort the observation for each mother to the top of each block.

Naturally, there are yet other ways to solve the problem. From Cox (2011), you can see that

```
. egen mother_age = max(cond(is_mother, age, .)), by(fam_id)
```

gets you there in one. Other way round, we stop short here of what to do if there are two or more mothers in each family.

4 Reversing time or other order: Negate the key variable

Occasionally, the reverse problem arises—wanting to carry the last value back in time or backward in some other order. At first sight, this goal is frustrated by Stata's default of implementing commands in observation order—until you realize that reversing time just needs a negated time or sequence variable used temporarily in sorting the data, as in

```
. generate rev_time = - time
```

If this seems a little puzzling, we will see an example shortly. The device is obvious once you know but is worth the flag of a section title.

5 Adding caution

The spirit of this column is that a researcher knows, or at least is highly confident, what a missing value should be. Confidence is often tempered with caution, as when you want to fill in values if (and only if) known values on either side of a gap are equal or only within a relatively short interval after the last known value. Let's take these variations in turn.

5.1 Interpolating only between equal known values

Wanting to build a bridge only between equal values implies that a sequence 1 . . . 1 should be filled in with values of 1, while 1 . . . 2 should be left as is. Here dots or periods once more represent missing values. The example best fits, but is not quite limited to, variables that encode categories. The sequence 1 . . . 2 implies—or leads us to guess—that at some point between the first and the last value, the state jumped from 1 to 2, but we cannot say certainly when that happened.

Here you need contextual information on what is possible, or what is likely, to guide a decision on whether what you do is defensible. If someone was reported as a university graduate in 2016 and in 2020, it is safe to assume that such a person was a graduate in between. We rule out some bizarre scenario in which someone was first stripped of a degree but then got another one anyway. If someone was reported as employed, or employed by a particular company, in 2016 and 2020, but values are missing in between, filling in the gaps is a riskier change. The same would apply to, say, owning a car in between.

Suppose we decided to fill in anyway, so long as values are the same on either side of a gap. One way to do this is to interpolate forward and backward and use the interpolated values only if methods agree. That is, 1 . . . 1 is interpolated as 1 1 1 1 1 either way, but 1 . . . 2 is mapped to 1 1 1 1 2 by forward copying and to 1 2 2 2 2 by backward copying, so methods disagree. Another way to do it is to use linear interpolation as well as forward copying and again to accept interpolated values if and only if methods agree.

Let's make that concrete. Like much in school mathematics, these problems characteristically appear trivial when solved but tricky before you have gotten there. If you wish to experiment yourself, the dataset is bundled with the media for this column.

```
. use gap_example, clear

. list

     ┌───────────────────┐
     │ id   year   value │
     ├───────────────────┤
  1. │  1   2016       1 │
  2. │  1   2017       . │
  3. │  1   2018       . │
  4. │  1   2019       . │
  5. │  1   2020       1 │
     ├───────────────────┤
  6. │  2   2016       1 │
  7. │  2   2017       . │
  8. │  2   2018       . │
  9. │  2   2019       . │
 10. │  2   2020       2 │
     └───────────────────┘

. generate negyear = - year

. clonevar value_f = value
(6 missing values generated)

. bysort id (year) : replace value_f = value_f[_n-1] if missing(value_f)
(6 real changes made)

. clonevar value_b = value
(6 missing values generated)

. bysort id (negyear) : replace value_b = value_b[_n-1] if missing(value_b)
(6 real changes made)

. by id: ipolate value year, gen(value_l)

. sort id year

. list

     ┌───────────────────────────────────────────────────────────────┐
     │ id   year   value   negyear   value_f   value_b   value_l │
     ├───────────────────────────────────────────────────────────────┤
  1. │  1   2016       1     -2016         1         1         1 │
  2. │  1   2017       .     -2017         1         1         1 │
  3. │  1   2018       .     -2018         1         1         1 │
  4. │  1   2019       .     -2019         1         1         1 │
  5. │  1   2020       1     -2020         1         1         1 │
     ├───────────────────────────────────────────────────────────────┤
  6. │  2   2016       1     -2016         1         1         1 │
  7. │  2   2017       .     -2017         1         2      1.25 │
  8. │  2   2018       .     -2018         1         2       1.5 │
  9. │  2   2019       .     -2019         1         2      1.75 │
 10. │  2   2020       2     -2020         2         2         2 │
     └───────────────────────────────────────────────────────────────┘
```

This worked example raises some crucial points.

`ipolate` is Stata's long-standing command for linear interpolation. If the term is new to you, this is just what you did as a child, either frivolously with puzzle pictures or more seriously with line graphs of data. You joined points or dots in a two-dimensional space (perhaps on a sheet of paper) with straight line segments. As a secondary school student in the 1960s, I used interpolation to get an extra decimal place out of printed tables for logarithmic or other functions, a minor skill that quickly became redundant with the advent of inexpensive electronic calculators. More generally, reading "between the lines" of published tables was a major motivator for interpolation methods.

Most pertinently here, `ipolate`'s linear segments will be flat or constant in the interpolated variable if and only if values are equal on either side of a gap, which is what we want here.

Note that needing to interpolate separately within groups or blocks of observations is only a little more challenging than if the dataset is in effect one group or block. Both `replace` and `ipolate` work with a `by:` prefix. In the case of `replace`, observations need to be in a good `sort` order because otherwise you can easily copy the wrong values. For `ipolate`, the command interpolates with respect to the second variable named, regardless of initial sort order.

A historical aside: `ipolate` allows use of a `by()` option instead of a `by:` prefix. This was the original syntax, still supported but now undocumented, and so you may see examples of its use.

More importantly, you will see that `ipolate` takes the variable to be interpolated literally, which is to say numerically. The command has no notion of whether answers with fractional parts are sensible or acceptable; this decision is up to the user, who should know.

In sum, checking that interpolation is acceptable only when it yields constants can be done by checking copying forward, either with copying backward or with linear interpolation. Either way, if the results are the same, you are good, as with `id` 1 in this example; otherwise, you are not, as with `id` 2. So we could finish with something like

```
. generate wanted = value_f if value_f == value_b
(3 missing values generated)
```

or

```
. generate wanted = value_f if value_f == value_l
```

5.2 Interpolating only for shorter periods

Caution often exercised is to fill in missing values only if the last known value was close in time or close as measured some other way. In homely terms, the notion is that one might stretch evidence but only so far. Quite what the cutoff or threshold should be is up to the researcher and can be anything from a wild guess to something reflecting knowledge or understanding of a process. Let's revisit our previous example and invoke a rule that we will copy for at most two years. That is, we have to tell Stata when to stop the cascade of replacements.

```
. use gap_example, clear
. list

     ┌────────────────────┐
     │ id   year   value │
     ├────────────────────┤
  1. │  1   2016       1 │
  2. │  1   2017       . │
  3. │  1   2018       . │
  4. │  1   2019       . │
  5. │  1   2020       1 │
     ├────────────────────┤
  6. │  2   2016       1 │
  7. │  2   2017       . │
  8. │  2   2018       . │
  9. │  2   2019       . │
 10. │  2   2020       2 │
     └────────────────────┘
```

A solution grows out of keeping track of *when* we last knew a value. Again, this can be calculated using our trick of copying forward.

```
. generate last_known = year if !missing(value)
(6 missing values generated)
. bysort id (year): replace last_known = last_known[_n-1] if missing(last_known)
(6 real changes made)
. list

     ┌───────────────────────────────┐
     │ id   year   value   last_k~n │
     ├───────────────────────────────┤
  1. │  1   2016       1       2016 │
  2. │  1   2017       .       2016 │
  3. │  1   2018       .       2016 │
  4. │  1   2019       .       2016 │
  5. │  1   2020       1       2020 │
     ├───────────────────────────────┤
  6. │  2   2016       1       2016 │
  7. │  2   2017       .       2016 │
  8. │  2   2018       .       2016 │
  9. │  2   2019       .       2016 │
 10. │  2   2020       2       2020 │
     └───────────────────────────────┘
```

Now we copy downward within our cautious limit:

```
. clonevar copy = value
(6 missing values generated)
. bysort id (year) : replace copy = copy[_n-1] if missing(copy) &
> (year - last_known) <= 2
(4 real changes made)
```

```
. list

     ┌──────────────────────────────────────┐
     │ id   year   value   last_k~n   copy │
     ├──────────────────────────────────────┤
  1. │  1   2016       1       2016      1 │
  2. │  1   2017       .       2016      1 │
  3. │  1   2018       .       2016      1 │
  4. │  1   2019       .       2016      . │
  5. │  1   2020       1       2020      1 │
     ├──────────────────────────────────────┤
  6. │  2   2016       1       2016      1 │
  7. │  2   2017       .       2016      1 │
  8. │  2   2018       .       2016      1 │
  9. │  2   2019       .       2016      . │
 10. │  2   2020       2       2020      2 │
     └──────────────────────────────────────┘
```

A little thought shows that this method also works with an irregularly spaced time or other position or sequence variable, as would be typical of much data on patients seen at various times and indeed in many other examples.

Modification to copy backward once more just requires a reversing of time or of the relevant variable.

6 Increasing or decreasing linear sequences

A different case in which we can be clear what the missing value should be is when a variable should increase linearly. If we wanted to type in a sequence of years, we could type

```
. generate year = 2001 in 1
```

and follow with

```
. replace year = year[_n-1] + 1 if missing(year)
```

That will be recognized as just a variation on the now familiar trick of copying downward. Adding 1 is what to do if you need it; adding 10 could make sense for a series of census years given the common practice of holding a census once a decade.

This solution does not exclude others, naturally, such as

```
. generate year = 2000 + _n
```

or use of the `range` (see [D] **range**) command or of the `seq()` function within the `egen` command.

All of these devices extend naturally to datasets with group structure, such as panel or longitudinal data.

A related problem has to do with people's ages, which unsurprisingly (should) increase by 1 each year. Hence, missing values on age can be interpolated with reference to a yearly date variable. There can be small stuff to reckon with, such as people forgetting or being less than candid on how old they are or survey waves being implemented at differing times of year and so before or after people's birthdays in each year. For these and other reasons, plotting age against year is often a good idea, although doing that for many different people can be a challenge.

The section title mentions decreasing sequences, just a variation in principle, if less common in practice.

7 Conclusion

Missing values are common in real datasets. What to do about them is a large and challenging question. This column has surveyed the easiest problems in which a researcher is clear, or at least highly confident, about what missing values should be instead, implying a deterministic replacement.

Major tricks and tips include

- copying values from observation to observation, exploiting the fact that `replace` works in observation order;
- using a `by:` prefix and controlling `sort` order to `replace` within groups of observations (panel or longitudinal data, family or household data, and so forth);
- being willing to negate time or some other position or sequence values to copy backward in a dataset;
- satisfying constraints that interpolation is confined to filling gaps between known equal values or to observations within a certain distance between known values in time or some other sequence or position variable; and
- adapting these ideas to increasing or decreasing sequences to be used in infill.

8 Acknowledgments

Many questions on Statalist and Stack Overflow helped to identify common problems and to indicate that something like this might be helpful.

9 References

Bowley, A. L. 1901. *Elements of Statistics*. London: P. S. King.

Cox, N. J. 2002. Speaking Stata: How to move step by: step. *Stata Journal* 2: 86–102. https://doi.org/10.1177/1536867X0200200106.

———. 2005. Stata tip 17: Filling in the gaps. *Stata Journal* 5: 135–136. https://doi.org/10.1177/1536867X0500500117.

———. 2011. Speaking Stata: Compared with .... *Stata Journal* 11: 305–314. https://doi.org/10.1177/1536867X1101100210.

Cox, N. J., and C. B. Schechter. 2019. Speaking Stata: How best to generate indicator or dummy variables. *Stata Journal* 19: 246–259. https://doi.org/10.1177/1536867X19830921.

Davis, J. C. 1973. *Statistics and Data Analysis in Geology*. New York: Wiley.

Hamming, R. W. 1973. *Numerical Methods for Scientists and Engineers*. New York: McGraw–Hill.

Lancaster, P., and K. Šalkauskas. 1986. *Curve and Surface Fitting: An Introduction*. London: Academic Press.

Meijering, E. 2002. A chronology of interpolation: From ancient astronomy to modern signal and image processing. *Proceedings of the IEEE* 90: 319–342. https://doi.org/10.1109/5.993400.

Morton, B. R. 1964. *Numerical Approximation*. London: Routledge and Kegan Paul.

Newson, R. B. 2004. Stata tip 13: generate and replace use the current sort order. *Stata Journal* 4: 484–485. https://doi.org/10.1177/1536867X0400400411.

Pollard, J. H. 1977. *A Handbook of Numerical and Statistical Techniques: With Examples Mainly from the Life Sciences*. Cambridge: Cambridge University Press. https://doi.org/10.1017/CBO9780511569692.

Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. 2007. *Numerical Recipes: The Art of Scientific Computing*. 3rd ed. Cambridge: Cambridge University Press.

Whittaker, E. T., and G. Robinson. 1924. *The Calculus of Observations: A Treatise on Numerical Mathematics*. London: Blackie and Son.

Wilkes, M. V. 1966. *A Short Introduction to Numerical Analysis*. London: Cambridge University Press. https://doi.org/10.1017/CBO9780511666346.

Yule, G. U. 1911. *An Introduction to the Theory of Statistics*. London: Griffin.

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 16 commands in official Stata. He was an author of several inserts in the *Stata Technical Bulletin* and is Editor-at-Large of the *Stata Journal*. His "Speaking Stata" articles on graphics from 2004 to 2013 have been collected as *Speaking Stata Graphics* (2014, College Station, TX: Stata Press).