The Stata Journal (2011)
11, Number 1, pp. 126–142

Speaking Stata: MMXI and all that: Handling Roman numerals within Stata

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

Abstract. The problem of handling Roman numerals in Stata is used to illustrate issues arising in the handling of classification codes in character string form and their numeric equivalents. The solutions include Stata programs and Mata functions for conversion from numeric to string and from string to numeric. Defining acceptable input and trapping and flagging incorrect or unmanageable inputs are key concerns in good practice. Regular expressions are especially valuable for this problem.

Keywords: dm0054, fromroman, toroman, Roman numerals, strings, regular expressions, Mata, data management

1 Introduction

“MMXI” within the title, as you will recognize, is a Roman numeral representing 2011. Although the decline and fall of the Roman Empire is a matter of ancient history, Roman numerals are far from obsolete, even in contemporary science and technology. Recently on Statalist, the distinguished statistician Tony Lachenbruch asked about handling Roman numerals in Stata. Answering his question has proved to be entertaining and enlightening, so here I share my experiences.

The interest of this problem does not depend on how commonly it arises in Stata practice. Rather, how well can Stata meet such a challenge? One hallmark of a statistical environment such as Stata is its extensibility, its scope for adding new functionality that supports new needs. The Roman numeral problem raises several issues typical of conversions between string codes and numeric equivalents. Depending partly on your background, you are likely to be familiar with formal codes for book classification, medical conditions, sectors of economic activity, and so forth. The Roman numeral example is not so trivial that it can be discussed and dismissed in a paragraph, but not so tricky nor so large as to defy careful and complete examination in a column.

As of Stata 11, official Stata provides no specific support for handling Roman numerals, for converting them to Hindu–Arabic decimal numbers, or for representing decimal numbers as Roman numerals using a dedicated display format. Users do not have scope for creating new Stata functions or display formats, so the problem in practice pivots on being able to move back and forth between string representations such as `"MMXI"` and numeric representations such as 2011.

A key principle with any kind of conversion is that conversions either way should be supported if they both make sense. Thus in Stata, string-to-numeric and numeric-to-string functions and commands occur in pairs: `real()` and `string()`, `encode` and `decode`, and `destring` and `tostring`. See an earlier column (Cox 2002) for more on this theme, while noting that `tostring` has become an official command since that column was written. A further motive for implementing both conversions is to provide some consistency checking. Broadly, conversion one way followed by conversion the other way should yield the original. (We will touch later on the possibility of different string encodings of the same numbers.)

With this column are published two new programs, `fromroman` and `toroman`, and two stand-alone Mata functions. We need to discuss not only how to solve the problem but also why it is done that way.

2 Roman numerals

Let us first spell out the rules, or at least one common version of the rules, for Roman numerals and their conversion to decimal numbers.

Character set. The atoms (our term) M, D, C, L, X, V, and I stand for 1000, 500, 100, 50, 10, 5, and 1.

Subtraction rule. The composites (also our term) CM, CD, XC, XL, IX, and IV are used to stand for 900, 400, 90, 40, 9, and 4. Whenever any of these composites occurs, this rule trumps the previous rule.

Order rule. Two or more atoms or composites appearing within a numeral appear in the order implied by their numerical equivalent.

Parsimony rule. The smallest number of elements possible is used to encode any number.

Addition rule. Following conversions under the first two rules, sum the results.

Thus a person knowing these rules would know to parse MMXI as M + M + X + I = 1000 + 1000 + 10 + 1 = 2011 and would also parse MCMLII as M + CM + L + I + I = 1000 + 900 + 50 + 1 + 1 = 1952.

The subtraction rule is the twist that gives this problem its particular spin. Without it, we would just need to count the occurrences of the seven possible atoms, multiply, and then sum—a much simpler problem.

For those seeking further information on Roman numerals, the popular accounts of Gullberg (1997) and Ifrah (1998) provide much interesting and useful detail. Although these works have some scholarly limitations (Allen 1999; Zebrowski 2001; and Dauben 2002), they remain interesting and useful at the level we need here. The classic monographs of Cajori (1928) and Menninger (1969) remain useful surveys. Such sources underline that what we now know as Roman numerals are in fact a limited and late version. Roman symbols and conventions for numbers greater than 1000, for fractions, and for various multiplications have evidently not survived into widespread current use, so no more will be said about them. Conversely, the subtraction principle, whereby (for example) CM is interpreted as C subtracted from M, only became popular in late medieval times.

History aside, it is often said that Roman numerals are not especially attractive mathematically. Addition and subtraction with Roman numerals are awkward, and multiplication and division rather more so, although the difficulties can be exaggerated. The subtractive notation that increases the awkwardness is of late popularity, and the abacus was often used for calculations, anyway.

However, the rules do seem a little arbitrary. Clearly, there would be no practical point to allowing, say, DM, LC, or VX, which would just be longer ways of writing D, L, or V. But what is the objection to, say, IM, VM, or XM? Indeed, historical examples of subtractive composites other than those specified above can be found. However, the point is not to question the rules but to state what they are usually reported to be. Either way, Roman numerals are best thought of as presentation numerics rather than computational numerics.

3 Converting variables containing Roman numerals

3.1 Principles

Let us imagine a string variable with values that are Roman numerals such as `"I"`, `"II"`, `"III"`, `"IV"`, `"V"`, and so forth. The initial problem is to convert to decimal numbers such as 1 to 5. If only a few small numbers were so represented, it might be feasible to define value labels that could then be used with the `encode` command (see [D] **encode**), but this approach is not practical otherwise, at least without a tool created for the purpose. More subtly, a value-label approach would fail if there were different possibilities for representation, as if `IIII` were also allowed as an alternative to `IV`, or `XXXX` as an alternative to `XL`, or `CCCC` as an alternative to `CD`. A curiosity is that `IIII` is often shown on clockfaces and watch faces using Roman numerals.

Hence, we will need to set up our own conversion code. The idea that the data come as variables will for the while lie behind discussion. In a later section, we will focus on a Mata-based approach that extends the scope.

One possible starting point is to think through how somebody conversant with the rules would convert a Roman numeral by eye. However, recipes suitable for people are not always those most suitable for programs, as every programmer learns. I have not tried to write Stata code to parse Roman numerals from left to right, as many of us were taught to do in our early education. The approach I have tried looks for subtractive composites first, given that their occurrence trumps atom-by-atom interpretation.

Given a string Roman numeral to be converted to a decimal number, here is the core of the algorithm:

1. Initialize the number to 0.
2. Find any occurrences of CM, CD, XC, XL, IX, and IV. Increment the number in each case by 900, 400, 90, 40, 9, or 4, as appropriate. Blank out those occurrences.
3. Find any occurrences of M, D, C, L, X, V, and I. Increment the number in each case by 1000, 500, 100, 50, 10, 5, or 1, as appropriate. Blank out those occurrences.
4. Whatever remains is regarded as problematic input and is flagged to the user.

   You can easily mimic this algorithm with examples such as `"MMXI"` and `"MCMLII"` by using pencil and paper and crossing out rather than blanking out. In Stata, the function for blanking out is `subinstr()`, which is used in this case to replace substrings by blanks or empty strings (`""`).

   As yet, this algorithm includes no checking for malformed numerals. `"IM"` would be converted to 1001 rather than be rejected as malformed. This problem will be addressed shortly.

   More positively, there are three easy extensions to handle other possibilities.

5. Strings often contain spaces, which should usually just be ignored. In particular, any leading and trailing spaces might just be side effects of data entry. However, if a user had composite strings such as `"II III"`, meaning 2 and 3 as two separate occurrences, then applying the `split` command (see [D] **split**) beforehand would be recommended.
6. The possibility of lowercase numerals (m, d, c, l, x, v, and i) would be easy to handle, because the function `upper()` can be used to convert to uppercase beforehand.
7. Any occurrences of j for i or of u for v—seemingly rare now but mentioned in the literature as historic variants—could be accommodated with other applications of `subinstr()`.

3.2 Code

The Stata code published with this column includes a Stata program, `fromroman`, and a Mata function with the same name for converting Roman numerals. It would be possible to write a program entirely in Stata for this problem, but a Stata program that calls a Mata function is a more attractive solution. Greater speed is sometimes a motive for using Mata, but not for this problem because the computations are trivial in machine time. Convenience of the programmer is a greater motive for using Mata because Mata provides much of the elegance and generality of modern programming languages with the hooks needed to interface with Stata.

The remainder of this subsection presupposes some acquaintance with Mata for a full understanding, but even if Mata is new to you, you might want to skim along. The logic of the problem is not difficult, and the Mata details should make some sense, even if you could not have written them down yourself. Much of the code is mundane, but some sections are more distinctive and worth some comment.

At the heart of `fromroman` is Mata code that converts a string column vector `sin` of Roman numerals to a numeric column vector of decimal numbers `nout`. These column vectors are in turn input from a Stata string variable and intended for output to a Stata numeric variable. `nout` is initialized as all zeros.

```
sin = st_sdata(., varname, usename)
nout = J(rows(sin), 1, 0)
```

Two vectors of string and numeric constants are set up to define the conversion mapping. Anyone wanting to use different conversion rules would need to modify these vectors. It is arbitrary that they are set up as column vectors, because they are not aligned with the other column vectors. Row vectors would also work fine.

```
rom = ("CM", "CD", "XC", "XL", "IX", "IV", "M", "D", "C", "L", "X", "V", "I")´
num = (900, 400, 90, 40, 9, 4, 1000, 500, 100, 50, 10, 5, 1)´
```

There is then a loop over both of these vectors. We must look for any of the composites (CM, CD, XC, XL, IX, and IV) before we look for any of the atoms (M, D, C, L, X, V, and I), so to that extent, the order is important.

In an early version of the program, I wrote the segment below. The code can be much improved, as I will explain.

```
for (i = 1; i <= rows(rom); i++) {
        while (sum(strpos(sin, rom[i]))) {
                nout = nout + num[i] * (strpos(sin, rom[i]) :> 0)
                sin = subinstr(sin, rom[i], "", 1)
        }
}
```

A small point of style here is that the loop is written as over the elements of the vector as from 1 to `rows(rom)`. We could wire in 13 (the number of elements in both vectors) and save ourselves a tiny amount of computation, but the downside is to require anyone trying to understand the code to puzzle out what the constant 13 is. This choice is reinforced by the idea that someone wanting different rules would need to change the vectors. The opposite decision might be made if the problem were, say, looping over the decimal digits 0 to 9, when it should be obvious why 10 is the number of elements.

That said, let us focus on the conversion itself. There is a test of whether each element of `rom[i]` is contained within `sin`, which is done within the line

```
nout = nout + num[i] * (strpos(sin, rom[i]) :> 0)
```

`strpos()` returns the position of that element. To see how that works, follow through as we start with `rom[1]`, which is `"CM"`.

If we were processing `"MCMLII"`, then `strpos("MCMLII", "CM")` would be returned as 2 because the string `"CM"` does occur within `"MCMLII"`, and it starts at the second character of `"MCMLII"`. In contrast, `strpos("MMXI", "CM")` is returned as 0 because the string `"CM"` does not occur within `"MMXI"`. More generally, `strpos(`*str1*`, `*str2*`)` returns a positive integer for the first position of *str2* within *str1*, or it returns 0 if there is no such position. (Longtime users of Stata may have met `strpos()` in Stata under the earlier name `index()`.)

Thus the comparison `strpos(sin, rom[i]) :> 0` yields 1 whenever the element occurs and 0 whenever it does not occur because those cases yield positive and 0 results from `strpos()`, and the latter never returns a negative result. Multiplying 1 or 0 by `num[i]` adds `num[i]` or 0, as appropriate. Thus with `CM`, 900 would be added to 0 (the initial value) for the input `"MCMLII"`, but 0 would be added for the input `"MMXI"`.

Once we have taken numeric account of the first occurrence of each pertinent element of `rom[]`, we can blank it out within `sin`:

```
sin = subinstr(sin, rom[i], "", 1)
```

Within Mata, as typically within Stata, this calculation is vectorized so that Mata deals with all the elements of `sin`, which correspond to the values held within various observations for a string variable. That is why the inequality comparison is elementwise, as shown by the colon prefix in `:>`.

However, as already mentioned, `strpos()` identifies at most the first occurrence of one string within another. That suffices when dealing with the composites such as `"CM"`, which we expect to occur at most once within any numeral. However, we need to be able to handle multiple occurrences of `"M"`, `"C"`, `"X"`, and `"I"`, which are predictable possibilities. These are handled by continuing to process such elements as long as they are found. The two statements we have looked at in detail are within a `while` loop:

```
while (sum(strpos(sin, rom[i]))) {
        nout = nout + num[i] * (strpos(sin, rom[i]) :> 0)
        sin = subinstr(sin, rom[i], "", 1)
}
```

Recall that we blank out each occurrence of the elements of `rom[]` within `sin[]` as we process it. We can monitor whether there is anything left to process by summing `strpos(sin, rom[i])`. The sum, like the individual results from `strpos()`, is positive given any occurrence and 0 otherwise, but this time the check is for occurrences within the entire vector. A `while` loop continues so long as the argument of `while()` is positive. Some might prefer to spell out the logic as

```
while (sum(strpos(sin, rom[i])) > 0)
```

The choice is one of style. The result is the same.

Let us back up and consider another way, which is that now used in `fromroman`. Instead of blanking out occurrences of each element of `rom[i]` one by one, we can blank them all out at once. We can track how many occurrences there were from a comparison of string lengths. `strlen()` returns the length of a string. (Longtime users of Stata may have met `strlen()` in Stata under the earlier name `length()`.) Here is the new code:

```
for (i = 1; i <= rows(rom); i++) {
        sin2 = subinstr(sin, rom[i], "", .)
        nout = nout + num[i] * (strlen(sin) - strlen(sin2)) / strlen(rom[i])
        sin = sin2
}
```

So the number of elements blanked out is calculated from the difference between string lengths before and after. Division by `strlen(rom[i])` is needed because composite elements are two characters long, and atoms are just one character long.

This way, the repetition encoded in `while()` loops becomes quite unnecessary. The more ambitious code is in fact simpler, which is reminiscent of what Pólya (1957, 121) codified as the inventor's paradox: the more ambitious plan may have more chances of success.

3.3 Checking

We have postponed discussion of the need to check that input does actually obey the rules we are using. If it does not, further questions arise for the user: Is the supposedly bad input some kind of data error? Is a variation on the rules needed? The latter would arise if (for example) CCCC, XXXX, or IIII were regarded as acceptable.

A good way to check input is to define acceptable input using a so-called regular expression. Regular expressions are a little language for defining the patterns that strings may take. In practice, they are a little language with many different dialects, implemented in a variety of software. Stata's own regular expression implementation is most fully documented by Turner (2005). Much fuller discussions of regular expressions in general are available: the excellent account by Friedl (2006) is accessible and useful to learners. All that is needed to solve the problem here is contained within its first chapter. Brief introductions to be found in many books (for example, Kernighan and Pike [1984]; Aho, Kernighan, and Weinberger [1988]; Abrahams and Larson [1997]; and Raymond [2004]) may also be useful.

A regular expression for Roman numerals is

```
^M*(C|CC|CCC|CD|D|DC|DCC|DCCC|CM)?(X|XX|XXX|XL|L|LX|LXX|LXXX|XC)?
  (I|II|III|IV|V|VI|VII|VIII|IX)?$
```

This can be parsed as follows:

1. `^` marks the start of the numeral.
2. `M*` means that the character M may occur zero or more times.

3. `(C|CC|CCC|CD|D|DC|DCC|DCCC|CM)?` means that one of C, CC, CCC, CD, D, DC, DCC, DCCC, or CM may occur (meaning precisely, may occur zero or one time).
4. Similarly, `(X|XX|XXX|XL|L|LX|LXX|LXXX|XC)?` and `(I|II|III|IV|V|VI|VII|VIII|IX)?` mean that just one of the items in each parenthesized list may occur.
5. `$` marks the end of the numeral.

Thus the idea here is that `^`, `*`, `?`, `|`, `()`, and `$`—so-called meta-characters—have special syntactic meaning, while the other characters are all to be taken literally.

Much more could be said about regular expressions, which are an important area in their own right, but a few comments will have to do. You should not imagine that the use of meta-characters (which include others beyond those used here) rules out defining regular expressions in which the same characters occur literally. There are simple tricks to meet such needs.

The regular expression here for Roman numerals makes no particular element compulsory. A side effect is that it is satisfied by empty strings, which break none of the rules. This is not a problem so long as we remember to check that a string is not empty or we ensure that no empty strings are fed to the regular expression.

In writing down this regular expression, I plumped for clarity rather than brevity. Following a personal communication from Kerry Kammire,

```
^M*(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$
```

can be offered as another way to write it down.

Once you have a regular expression, the Stata function `regexm()` yields 1 if it is satisfied and 0 otherwise. Thus we can reject input, or more helpfully, flag it if it fails such a test.

I offer a further aside: In an earlier version of `fromroman`, I went through the possible elements one by one using code based on `if` to test for elements that might occur once and code based on `while` to test for elements that might occur more than once. What lay behind this design was twofold. For a small problem, I like to get a rough program working as quickly as possible. Once code is in front of me, it is often much easier to see how to improve it. More specifically, the underlying idea was to include some degree of checking against malformed numerals. Once code had been written to check input all at once using a regular expression, it became obvious that the repetitive mix of `if` and `while` code could be replaced by a single `for` loop. Once again, the more ambitious code is in fact simpler.

While in a vein of dispensing homespun philosophy, it need only be added that the art of solving large problems is often to reduce them to a series of small problems.

4 Converting variables to Roman numerals

4.1 Principles

Let us now imagine the reverse or inverse problem. We have a variable with decimal numeric values and a need to convert to a string variable containing Roman numerals. In practice, we are concerned with positive integers such as 42; 1952; or 2011. Here is one algorithm for producing Roman numerals:

1. Initialize the numeral to `""`.
2. For each of the elements 1000, 900, 500, 400, 100, 90, 50, 40, 10, 5, 4, and 1, follow these steps:
    a. Try to subtract that element from the number.
    b. If the result is positive or 0, add (concatenate to the right) the corresponding numeral atom or composite, and subtract the element, replacing the number with a new one.
    c. If the result is positive, repeat 2.a and 2.b with the same element and the new number.
    d. If the result is 0, stop.
    e. If the result is negative, proceed to the next element.

Some constraints on the process are satisfied automatically with this procedure. For example, the fact that `"CM"` can occur at most once is satisfied because if any number is less than 1000, then 900 can be subtracted from it at most once. In the same way, other key facts are all consequences of the algorithm. That is, the other subtractive composites, and also `"D"`, `"L"`, and `"V"`, can each occur at most once; and `"C"`, `"X"`, and `"I"` can each occur at most three times.

4.2 Code

The Stata code published with this column includes a Stata program, `toroman` (and a Mata function with the same name), for converting decimal numbers to Roman. As before, comment is restricted to the most distinctive part of the code. If you skipped or skimmed earlier material on Mata, you might want to repeat that now.

In the Mata function at the heart of `fromroman`, a numeric column vector `nin` is mapped to a string column vector `sout`, which is initialized to empty (`""`) in each element. The conversion is defined as before by two aligned vectors. This time, the vectors are ordered largest number first.

This code is an implementation of the algorithm just given:

```
nin = st_data(., varname, usename)
sout = J(rows(nin), 1, "")
rom = ("M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I")´
num = (1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1)´
for (i = 1; i <= rows(rom); i++) {
        toadd = nin :- num[i] :>= 0
        while (sum(toadd)) {
                sout = sout :+ toadd :* rom[i]
                nin = toadd :* (nin :- num[i]) + (!toadd) :* nin
                toadd = nin :- num[i] :>= 0
        }
}
```

The main complication to be tackled is converting a vector all at once. As of Stata 11, Mata lacks an elementwise version of the conditional operator (that used in the example `a > b ?  a :  b`), so we mimic that for ourselves using a vector `toadd` containing 1s and 0s and its negation, `!toadd`. As the name suggests, `toadd` has values of 1 whenever we can add an element of `rom[]` and 0 otherwise. (Ben Jann's `moremata` package, downloadable from the Statistical Software Components archive, does include a conditional function that works elementwise. Typing `findit moremata` in Stata would point you to more information.)

Once we have worked out that conditional calculation, we can add the Roman numeral. In the line

```
sout = sout :+ toadd :* rom[i]
```

we are multiplying strings as well as adding them. In Mata, adding strings is just concatenation (joining lengthwise), and multiplying strings is just repeated concatenation so that `2 * "Stata"` yields `"StataStata"`. Here both the addition and the multiplication are elementwise, as shown again by the colon prefixes. Thus whenever `toadd` is 1, we add the corresponding element of `1 :* rom[i]`, which is just `rom[i]`; and whenever `toadd` is 0, we add `0 :* rom[i]`, which is always empty.

This is not exactly the code used in `toroman`. It is just a naïve version encoding one subtraction at each step, but there is no reason to limit ourselves to that. We can rewrite the loop to get the number of subtractions needed directly:

```
for (i = 1; i <= rows(rom); i++) {
        sout = sout :+ floor(nin/num[i]) :* rom[i]
        nin = nin :- num[i] :* floor(nin/num[i])
}
```

The more ambitious code is, yet again, much simpler. Let us see how that works with a simple numerical example. The vector given by

```
: floor((2011, 1952, 42)'/1000)
        1
    ┌─────┐
  1 │  2  │
  2 │  1  │
  3 │  0  │
    └─────┘
```

is the number of times we can write "M" at the start of the corresponding Roman numerals and the number of times to subtract 1000 for the next step of the loop. I like to use `floor()` as the function here because its name so clearly evokes rounding down (Cox 2003). The Mata function `trunc()`, the equivalent of Stata's `int()` function, would work, too.

The moral is simple but crucial. We can easily underestimate the scope of languages like Stata and Mata to do several things at once if we translate too closely from recipes using one little step at a time.

However, we need to watch out. Suppose that somehow -1 was fed to the code segment above. The Roman numeral that would emerge would be "CMXCIX", which looks crazy but can be explained. It emerges because `floor(-1/1000)` is -1. Subtracting -1000, so adding 1000, yields 999, which is then correctly encoded. There are various ways around this. One is to trap negative values beforehand. We will look at the question of checking shortly. Another is to work with the larger of 0 and `floor(num / rom[i])`, which ensures that negative values are mapped to empty strings and so ignored.

A final thought is this: If we were doing this by hand, then naturally we would stop whenever the job was done. Given 2000, "MM" is clearly the solution because repeated subtraction has yielded 0, and we would know not to try anything else. Is it worth building in a check that we are done?

Consider the statistics. Of possible Roman numerals, the frequencies per 1000 finished with each possible element are "M", 1; "CM", 1; "D", 1; "CD", 1; "C", 6; "XC", 10; "L", 10; "XL", 10; "X", 60; "IX", 100; "V", 100; "IV", 100; and "I", 600. Hence, testing to allow leaving a loop early will not in practice help us appreciably. It might well slow us down!

4.3 Checking

As mentioned in the previous subsection, numeric values for this problem should in practice mean positive integers only. A careful approach requires trapping any fractional, zero, and negative numbers given as input. This is simple in Stata using an `if` condition applied before calling the Mata function doing the encoding.

A more subtle issue is the upper limit for this calculation. Possible candidates for conversion include year dates and page numbers, which are both most unlikely to exceed a few thousand and so will be encoded by at most a few characters in a string. Nevertheless, it is worth thinking carefully about the limits to any conversion program.

As of Stata 11, the largest (meaning widest) type of string variable allowed in Stata is `str244`. Thus 245,000, which would convert to a numeral containing 245 Ms, could not be held as a Roman numeral within a Stata variable. In fact, not all smaller numbers can be so held. We can work out the precise limit as follows.

Inspection makes clear that numbers with digits of 8 produce the longest Roman numerals. VIII is the longest for a one-digit decimal number, LXXXVIII and DCCCLXXXVIII are the longest for two- and three-digit numbers, and so on. The smallest problematic number will be the smallest number that needs 245 characters, which is 233,888, represented by 233 Ms followed by the 12 characters DCCCLXXXVIII.

As mentioned in an earlier section, other and much better ways to hold numbers that are several thousand or more characters as Roman numerals were used in history. The point is rather to find the upper limit within Stata to the procedure using the rules in section 2. It seems most unlikely that an upper limit of 233,887 would ever bite in practice, which is good news.

5 Syntax for fromroman and toroman

5.1 fromroman

`fromroman` *romanvar* [*if*] [*in*], `generate(`*numvar*`)` [`re(`*regex*`)`]

Description

`fromroman` creates a numeric variable *numvar* from a string variable *romanvar* following these rules:

1. Any spaces are ignored.
2. Lowercase letters are treated as if uppercase.
3. Numerals must match the Stata regular expression `^M*(CM|DCCC|DCC|DC|D|CD|CCC|CC|C)?(XC|LXXX|LXX|LX|L|XL|XXX|XX|X)?(IX|VIII|VII|VI|V|IV|III|II|I)?$`. This forbids, for example, CCCC, XXXX, or IIII, but see documentation of the `re()` option below.
4. Single occurrences of CM, CD, XC, XL, IX, and IV are treated as 900, 400, 90, 40, 9, and 4, respectively.
5. M, D, C, L, X, V, and I are treated as 1000, 500, 100, 50, 10, 5, and 1, respectively, as many times as they occur.
6. The results of 4 and 5 are added.

7. Input of any other expression or characters is trapped as an error and results in missing. Examples would be minus signs and decimal points.

There is no explicit upper limit for the integer values created. In practice, the limit is implied by the limits on string variables so that, using these rules, any numbers greater than 244,000 (and some numbers less than that) could not be stored as Roman numerals in a Stata string variable. (The smallest problematic number is 233,888, which would convert to a Roman numeral consisting of 233 Ms followed by DCCCLXXXVIII—that is, a numeral 245 characters long.) See [D] **data types**.

Options

`generate(`*numvar*`)` specifies the name of the new numeric variable to be created. `generate()` is required.

`re(`*regex*`)` specifies a regular expression other than the default for checking input.

5.2 toroman

`toroman` *numvar* [*if*] [*in*] `, generate(`*romanvar*`)` [`lower`]

Description

`toroman` creates a string variable *romanvar* containing Roman numerals from a numeric variable *numvar* following these rules:

1. Negative, zero, and fractional numbers are ignored.
2. The conversion uses M, D, C, L, X, V, and I to represent 1000, 500, 100, 50, 10, 5, and 1 as many times as they occur, except that CM, CD, XC, XL, IX, and IV are used to represent 900, 400, 90, 40, 9, 4.
3. No number that is 233,888 or greater is converted. This limit is implied by the limits on string variables so that, using these rules, any number greater than 244,000 (and some numbers less than that) could not be stored as Roman numerals in a Stata string variable. (The smallest problematic number is 233,888, which would convert to a Roman numeral consisting of 233 Ms followed by DCCCLXXXVIII—that is, a numeral 245 characters long.) See [D] **data types**.

Options

`generate(`*romanvar*`)` specifies the name of the new string variable to be created. `generate()` is required.

`lower` specifies that numerals are to be produced as lowercase letters, such as `"mmxi"` rather than `"MMXI"`.

6 Mata functions

Two Mata functions are also included with the media for this column in the file `roman.mata`. These functions could be used either within Mata or within Stata.

Before we look at how to use those, first let us set aside the question of checking using regular expressions in Mata. Mata has a function called `regexm()`. It was not documented in Stata 9 or 10 and is undocumented in Stata 11, meaning that it is documented in a help file but not in a corresponding manual entry. However, `regexm()` in Mata acts exactly as you would expect given knowledge of the function with the same name in Stata.

Given a string input, define first a suitable regular expression

```
: regex =
> "^M*(CM|DCCC|DCC|DC|D|CD|CCC|CC|C)?(XC|LXXX|LXX|LX|L|XL|XXX|XX|X)?
> (IX|VIII|VII|VI|V|IV|III|II|I)?$"
```

and then input that does not pass muster can be shown by something like

```
: test = ("MMXI", "MCMLII", "XLII", "MIX", "foo")´
: select(test, !regexm(test, regex))
  foo
```

Let us see how the Mata functions operate. In both cases, there is mapping from matrix to matrix. So row vector input will work fine as a special case. `toroman()` ignores zeros and negative numbers, as well as any numeric missings, and returns empty strings in such cases. Fractional numbers are not ignored: given x, `toroman()` works with the floor $\lfloor x \rfloor$.

```
: ntest = (2011, 1952, 42, 0, -1, .)
: toroman(ntest)
            1        2        3        4        5        6
    +-------------------------------------------------------+
  1 |    MMXI   MCMLII     XLII                              |
    +-------------------------------------------------------+
```

`fromroman()` gives positive integers and flags problematic (nonempty) input.

```
: fromroman(toroman(ntest))
          1      2      3      4      5      6
     +-------------------------------------------+
   1 |  2011   1952     42      .      .      .  |
     +-------------------------------------------+

: stest = ("MMXI", "MCMLII", "XLII", "", "foo")
: fromroman(stest)
  Problematic input:
  foo
          1      2      3      4      5
     +------------------------------------+
   1 |  2011   1952     42      .      .  |
     +------------------------------------+
```

These Mata functions could be called from within a Stata session. That way, they would make up for the lack of any Stata functions in this territory. (We have just discussed two new Stata commands, a different matter.) Applications include conversion of Stata local or global macros or scalars to Roman numeral form. Suppose, for example, that you want to work with value labels for Roman numerals and that you know that only small numbers (say, 100 or smaller) will be used. You can define the value labels within a loop like this:

```
forval i = 1/100 {
        mata : st_local("label", toroman(`i´))
        label def roman `i´ "`label´", modify
}
```

That way, the tedious and error-prone business of defining several value labels one by one can be avoided. You could go on to assign such labels to a numeric variable or to use them with the `encode` command (see [D] **encode**).

7 Conclusions

By tradition, teaching discrete probability starts with problems in tossing coins and throwing dice. Outside of sport and gambling, few people care much about such processes, but they are easy to envisage and work well as vehicles for deeper ideas. In the same way, Roman numerals are not themselves of much note in Stata use, but the issues that arise in handling them are of more consequence.

Various simple Stata morals are illustrated by this problem of conversion between special string codes and numeric equivalents.

Write conversion code for both ways. If someone wants one way now, someone will want the other way sooner or later. Writing both is less than twice the work.

Define and check for acceptable input. Regular expressions are one very useful way of doing this. Flag unacceptable input. Consider the limits on conversions, even if they are unlikely to cause a problem in practice.

Combine Mata and Stata. Stata–Mata solutions give you the best of both worlds. Mata functions can be useful outside Mata.

Know your functions. Functions like `subinstr()`, `strpos()`, `strlen()`, and `floor()` give you the low-level manipulations you need.

Even rough solutions can often be improved. Often you have to write down code and mull it over before better code will occur to you.

More ambitious code can often be simpler. Handling special cases individually is sometimes necessary, but it often is a signal that a more general structure is needed.

8 Acknowledgments

Peter A. "Tony" Lachenbruch suggested the problem of handling Roman numerals on Statalist. This column is dedicated to Tony on his retirement as a small token of recognition of his many services to the statistical (and Stata) community.

Sergiy Radyakin's comments on Statalist provoked more error checking. Kerry Kammire suggested an alternative regular expression.

9 References

Abrahams, P. W., and B. R. Larson. 1997. *UNIX for the Impatient.* 2nd ed. Reading, MA: Addison–Wesley.

Aho, A. V., B. W. Kernighan, and P. J. Weinberger. 1988. *The AWK Programming Language.* Reading, MA: Addison–Wesley.

Allen, A. 1999. Review of Mathematics: From the Birth of Numbers, by Jan Gullberg. *American Mathematical Monthly* 106: 77–85.

Cajori, F. 1928. *A History of Mathematical Notations. Volume I: Notation in Elementary Mathematics.* Chicago: Open Court.

Cox, N. J. 2002. Speaking Stata: On numbers and strings. *Stata Journal* 2: 314–329.

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

Dauben, J. 2002. Review of The Universal History of Numbers and The Universal History of Computing, Parts I and II. *Notices of the American Mathematical Society* 49: 32–38 and 211–216.

Friedl, J. E. F. 2006. *Mastering Regular Expressions.* 3rd ed. Sebastopol, CA: O'Reilly.

Gullberg, J. 1997. *Mathematics: From the Birth of Numbers.* New York: W. W. Norton.

Ifrah, G. 1998. *The Universal History of Numbers: From Prehistory to the Invention of the Computer.* London: Harvill.

Kernighan, B. W., and R. Pike. 1984. *The UNIX Programming Environment.* Englewood Cliffs, NJ: Prentice Hall.

Menninger, K. 1969. *Number Words and Number Symbols: A Cultural History of Numbers*. Cambridge, MA: MIT Press.

Pólya, G. 1957. *How to Solve It: A New Aspect of Mathematical Method*. 2nd ed. Princeton, NJ: Princeton University Press.

Raymond, E. S. 2004. *The Art of UNIX Programming*. Boston, MA: Addison–Wesley.

Turner, K. S. 2005. FAQ: What are regular expressions and how can I use them in Stata? http://www.stata.com/support/faqs/data/regex.html.

Zebrowski, E., Jr. 2001. Review of The Universal History of Numbers: From Prehistory to the Invention of the Computer, by Georges Ifrah. *Isis* 92: 584–585.

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*.