Conclusions

The R language is a natural environment for the investigation of magic squares and hypercubes; and the discipline of translating published algorithms into R idiom can yield new insight. These insights include a new generalization of Frénicle's standard form to hypercubes, and also what appears to be the first algorithm for generating magic hypercubes of any dimension,

Insofar as magic squares and hypercubes are worthy of attention, it is worth creating fast, efficient routines to carry out the "paper" algorithms of the literature. I hope that the `magic` package will continue to facilitate the study of these fascinating objects.

Acknowledgements

I would like to acknowledge the many stimulating and helpful comments made by the R-help list over the years.

Bibliography

W. H. Benson and O. Jacoby. *New recreations with magic squares*. Dover, 1976. 49

J. R. Hendricks. Magic tesseracts and N-dimensional magic hypercubes. *Journal of Recreational Mathematics*, 6(3):193–201, 1973. 50

R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2004. URL http://www.R-project.org. ISBN 3-900051-07-0. 48

Robin Hankin
Southampton Oceanography Centre
European Way
Southampton
United Kingdom
SO14 3ZH
r.hankin@soc.soton.ac.uk

Programmer's Niche

How Do You Spell That Number?

John Fox

Frank Duan recently posted a question to the `r-help` mailing list asking how to translate numbers into words. The program described in this column is a cleaned-up and slightly enhanced version of my response to his question. I found the problem to be an interesting puzzle, and the solution uses several programming techniques that demonstrate the flexibility of R, including its ability to manipulate character-string data and to employ recursive function calls.

One intriguing aspect of the problem is that it required me to raise into consciousness my subconscious knowledge about how numbers are spoken and written in English. I was much more aware of these conventions in the languages (French and Spanish) that I had studied as a non-native speaker. A bit later, I realized that there are variations among English-speaking countries in the manner in which numbers are spoken and written down. Because I was born in the United States and have lived most of my adult life in Canada, I'm terminally confused about English spelling and usage. Canadian conventions are an amalgam of American and British rules.

In any event, it didn't take much time to see that the numbers from *one* to *nineteen* are represented by individual words; the numbers from *twenty-one* to *ninety-nine* are formed as compound words, with components for the tens and units digits — with the exceptions of multiples of ten (*twenty*, *thirty*, etc.), which are single words. The *Chicago Manual of Style* tells me that these compound words should be hyphenated (but offered little additional useful advice about how numbers are to be written out). Numbers from 100 to 999 are written by tacking on (at the left) a phrase like "six hundred" — that is, composed of a number from *one* to *nine* plus the suffix *hundred* (and there is no hyphen). Above this point, additional terms are added at the left, representing multiples of powers of 1000. In American English (and in Canada), the first few powers of 1000 have the following names, to be used as suffixes:

1000^1	*thousand*
1000^2	*million*
1000^3	*billion*
1000^4	*trillion*

Thus, for example, the number 210,363,258 would be rendered "two hundred ten million, three hundred sixty-three thousand, two hundred fifty-eight." There really is no point in going beyond trillions, because double-precision numbers can represent integers exactly only to about 15 decimal digits, or hundreds of trillions. Of course, I could allow numbers to be specified optionally by arbitrarily long character strings of numerals (e.g., `"21036325834723749231 0"`), but I didn't see a real need to go higher than hundreds of trillions.

One approach to converting numbers to words would be to manipulate the numbers as integers, but

it seemed to me simpler to convert numbers to character strings of numerals, which could then be split into individual characters: (1) larger integers can be represented exactly as double-precision floating-point numbers than as integers in R; (2) it is easier to manipulate the individual numerals than to perform repeated integer arithmetic to extract digits; and (3) having the numerals in character form allows me to take advantage of R's ability to index vectors by element names (see below).

I therefore defined the following function to convert a number to a vector of characters containing the numerals composing the number:

```
> makeDigits <- function(x)
+ strsplit(as.character(x), "")[[1]]
```

Here are some examples of the use of this function:

```
> makeDigits(123456)
[1] "1" "2" "3" "4" "5" "6"
> makeDigits(-123456)
[1] "-" "1" "2" "3" "4" "5" "6"
> makeDigits(1000000000)
[1] "1" "e" "+" "0" "9"
```

Notice the problems revealed by the second and third examples: It's necessary to make provision for negative numbers, and R wants to render certain numbers in scientific notation.[1] By setting the `scipen` ("scientific notation penalty") option to a large number, we can avoid the second problem:

```
> options(scipen=100)
> makeDigits(1000000000)
 [1] "1" "0" "0" "0" "0" "0" "0" "0" "0" "0"
```

It also seemed useful to have a function that converts a vector of numerals in character form back into a number:

```
> makeNumber <- function(x)
+ as.numeric(paste(x, collapse=""))
> makeNumber(c("1", "2", "3", "4", "5"))
[1] 12345
```

Finally, by way of preparation, I constructed several vectors of number words:

```
> ones <- c("zero", "one", "two", "three",
+     "four", "five", "six", "seven",
+     "eight", "nine")
> teens <- c("ten", "eleven", "twelve",
+     "thirteen", "fourteen", "fifteen",
+     "sixteen", " seventeen", "eighteen",
+     "nineteen")
> names(ones) <- names(teens) <- 0:9
> tens <- c("twenty", "thirty", "forty",
+     "fifty", "sixty", "seventy", "eighty",
+     "ninety")
> names(tens) <- 2:9
> suffixes <- c("thousand,", "million,",
+     "billion,", "trillion,")
```

Because the names of the elements of the first three vectors are numerals, they can conveniently be indexed; for example:

```
> ones["5"]
     5
"five"
> teens["3"]
         3
"thirteen"
> tens["7"]
        7
"seventy"
```

The vector of suffixes includes a comma after each word.

Figure 1 shows a function for converting a single integer to words; I've added line numbers to make it easier to describe how the function works:

And here are some examples of its use, wrapping long lines of output to fit on the page:

```
> number2words(123456789)
[1] "one hundred twenty-three million,
      four hundred fifty-six thousand,
      seven hundred eighty-nine"
> number2words(-123456789)
[1] "minus one hundred twenty-three million,
      four hundred fifty-six thousand,
      seven hundred eighty-nine"
> number2words(-123456000)
[1] "minus one hundred twenty-three million,
      four hundred fifty-six thousand"
```

I believe that the first five lines of the function are essentially self-explanatory. The rest of the function probably requires some explanation, however:

[6] If the number is composed of a single digit, then we can find the answer by simply indexing into the vector `ones`; the function `as.vector` is used to remove the name of (i.e., the numeral labelling) the selected element.

[7-9] If the number is composed of two digits and is less than or equal to 19, then we can get the answer by indexing into `teens` with the last digit (i.e., the second element of the `digits` vector). If the number is 20 or larger, then we need to attach the tens digit to the ones digit, with a hyphen in between. If,

[1] I don't want to mislead the reader: I discovered these and other problems the hard way, when they surfaced as bugs. The account here is a reconstruction that avoids my missteps. I can honestly say, however, that it took me much longer to write this column explaining how the program works than to write the original program. Moreover, in the process of writing up the program, I saw several ways to improve it, especially in clarity — a useful lesson.

```
[ 1]  number2words <- function(x){
[ 2]      negative <- x < 0
[ 3]      x <- abs(x)
[ 4]      digits <- makeDigits(x)
[ 5]      nDigits <- length(digits)
[ 6]      result <- if (nDigits == 1) as.vector(ones[digits])
[ 7]      else if (nDigits == 2)
[ 8]          if (x <= 19) as.vector(teens[digits[2]])
[ 9]              else trim(paste(tens[digits[1]], "-", ones[digits[2]], sep=""))
[10]      else if (nDigits == 3) {
[11]          tail <- makeNumber(digits[2:3])
[12]          if (tail == 0) paste(ones[digits[1]], "hundred")
[13]          else trim(paste(ones[digits[1]], "hundred", number2words(tail)))
[14]          }
[15]      else {
[16]          nSuffix <- ((nDigits + 2) %/% 3) - 1
[17]          if (nSuffix > length(suffixes) || nDigits > 15)
[18]              stop(paste(x, "is too large!"))
[19]          pick <- 1:(nDigits - 3*nSuffix)
[20]          trim(paste(number2words(makeNumber(digits[pick])),
[21]              suffixes[nSuffix], number2words(makeNumber(digits[-pick]))))
[22]          }
[23]      if (negative) paste("minus", result) else result
[24]      }
```

Figure 1: A function to convert a single integer into words.

however, the ones digit is 0, `ones["0"]` is `"zero"`, and thus we have an embarrassing result such as `"twenty-zero"`. More generally, the program can produce spurious hyphens, commas, spaces, and the strings `", zero"` and `"-zero"` in appropriate places. My solution was to write a function to trim these off:

```
trim <- function(text){
    gsub("(^\ *)|((\ *|-|,\ zero|-zero)$)",
         "", text)
    }
```

The `trim` function makes use of R's ability to process "regular expressions." See Lumley (2003) for a discussion of the use of regular expressions in R.

`[10-14]` If the number consists of three digits, then the first digit is used for hundreds, and the remaining two digits can be processed as an ordinary two-digit number; this is done by a recursive call to `number2words`[2] — unless the last two digits are 0, in which case, we don't need to convert them into words. The hundreds digit is then pasted onto the representation of the last two digits, and the result is trimmed. Notice that `makeNumber` is used to put the last two digits back into a number (called `tail`).

`[15-22]` Finally, if the number contains more than three digits, we're into the realm of thousands, millions, etc. The computation on line `[16]` determines with which power of 1000 we're dealing. Then, if the number is not too large, the appropriate digits are stripped off from the left of the number and attached to the proper suffix; the remaining digits to the right are recomposed into a number and processed with a recursive call, to be attached at the right.

`[23]` If the original number was negative, the word `"minus"` is pasted onto the front before the result is returned.

The final function, called `numbers2words` (shown in Figure 2), adds some bells and whistles: The various vectors of names are defined locally in the function; the utility functions `makeDigits`, `makeNumbers`, and `trim`, are similarly defined as local functions; and the function `number2words`, renamed `helper`, is also made local. Using a helper function rather than a recursive call permits efficient vectorization, via `sapply`, at the end of `numbers2words`. Were

[2] It's traditional in S to use `Recall` for a recursive function call, but I'm not fond of this convention, and I don't see an argument for it here: It's unlikely that `number2words` will be renamed, and in any event, it will become a local function in the final version of the program (see below).

`numbers2words` to call itself recursively, the local definitions of objects (such as the vector `ones` and the function `trim`) would be needlessly recomputed at each call, rather than only once. Because of R's lexical scoping, objects defined in the environment of `numbers2words` are visible to `helper`. For more on recursion in R, see Venables (2001).

`numbers2words` includes a couple of additional features. First, according to the *Oxford English Dictionary*, the definition of "billion" differs in the U.S. and (traditionally) in Britain: "1. orig. and still commonly in Great Britain: A million millions. (= U.S. trillion.) ... 2. In U.S., and increasingly in Britain: A thousand millions." Thus, if the argument `billion` is set to `"UK"`, a different vector of suffixes is used. Moreover, provision is made to avoid awkward translations that repeat the word "million," such as "five thousand million, one hundred million, ... ," which is instead, and more properly rendered as "five thousand, one hundred million, ..."

Second, Bill Venables tells me that outside of the U.S., it is common to write or speak a number such 101 as "one hundred and one" rather than as "one hundred one." (Both of these phrases seem correct to me, but as I said, I'm hopelessly confused about international variations in English.) I have therefore included another argument, called `and`, which is pasted into the number at the appropriate point. By default, this argument set is to `""` when `billion` is `"US"` and to `"and"` when `billion` is `"UK"`.

Some examples, again wrapping long lines of output:

```
> numbers2words(c(1234567890123, -0123, 1000))
[1] "one trillion,
      two hundred thirty-four billion,
      five hundred sixty-seven million,
      eight hundred ninety thousand,
      one hundred twenty-three"

[2] "minus one hundred twenty-three"
[3] "one thousand"
> numbers2words(c(1234567890123, -0123, 1000),
+     billion="UK")
[1] "one billion,
      two hundred and thirty-four thousand,
      five hundred and sixty-seven million,
      eight hundred and ninety thousand,
      one hundred and twenty-three"

[2] "minus one hundred and twenty-three"
[3] "one thousand"

> numbers2words(c(1234567890123, -0123, 1000),
+     and="and")

[1] "one trillion,
      two hundred and thirty-four billion,
      five hundred and sixty-seven million,
      eight hundred and ninety thousand,
      one hundred and twenty-three"

[2] "minus one hundred and twenty-three"
[3] "one thousand"
```

Finally, a challenge to the reader: At present, `numbers2words` rounds its input to whole numbers. Modify the program so that it takes a `digits` argument (with default `0`), giving the number of places to the right of the decimal point to which numbers are to be rounded, and then make provision for translating such numbers (e.g., `1234567.890`) into words.

John Fox
Sociology, McMaster University
`jfox@mcmaster.ca`

Bibliography

T. Lumley. Programmer's niche: Little bits of string. *R News*, 3(3):40–41, December 2003. URL http://CRAN.R-project.org/doc/Rnews/. 53

B. Venables. Programmer's niche. *R News*, 1(1):27–30, January 2001. URL http://CRAN.R-project.org/doc/Rnews/. 54

```
numbers2words <- function(x, billion=c("US", "UK"),
        and=if (billion == "US") "" else "and"){
    billion <- match.arg(billion)
    trim <- function(text){
        gsub("(^\ *)|((\ *|-|,\ zero|-zero)$)", "", text)
        }
    makeNumber <- function(x) as.numeric(paste(x, collapse=""))
    makeDigits <- function(x) strsplit(as.character(x), "")[[1]]
    helper <- function(x){
        negative <- x < 0
        x <- abs(x)
        digits <- makeDigits(x)
        nDigits <- length(digits)
        result <- if (nDigits == 1) as.vector(ones[digits])
        else if (nDigits == 2)
            if (x <= 19) as.vector(teens[digits[2]])
                else trim(paste(tens[digits[1]], "-", ones[digits[2]], sep=""))
        else if (nDigits == 3) {
            tail <- makeNumber(digits[2:3])
            if (tail == 0) paste(ones[digits[1]], "hundred")
            else trim(paste(ones[digits[1]], trim(paste("hundred", and)),
                helper(tail)))
            }
        else {
            nSuffix <- ((nDigits + 2) %/% 3) - 1
            if (nSuffix > length(suffixes) || nDigits > 15)
                stop(paste(x, "is too large!"))
            pick <- 1:(nDigits - 3*nSuffix)
            trim(paste(helper(makeNumber(digits[pick])),
                suffixes[nSuffix], helper(makeNumber(digits[-pick]))))
            }
        if (billion == "UK"){
            words <- strsplit(result, " ")[[1]]
            if (length(grep("million,", words)) > 1)
                result <- sub(" million, ", ", ", result)
            }
        if (negative) paste("minus", result) else result
        }
    opts <- options(scipen=100)
    on.exit(options(opts))
    ones <- c("zero", "one", "two", "three", "four", "five", "six", "seven",
        "eight", "nine")
    teens <- c("ten", "eleven", "twelve", "thirteen", "fourteen", "fifteen",
        "sixteen", " seventeen", "eighteen", "nineteen")
    names(ones) <- names(teens) <- 0:9
    tens <- c("twenty", "thirty", "forty", "fifty", "sixty", "seventy", "eighty",
        "ninety")
    names(tens) <- 2:9
    suffixes <- if (billion == "US")
                    c("thousand,", "million,", "billion,", "trillion,")
                else
                    c("thousand,", "million,", "thousand million,", "billion,")
    x <- round(x)
    if (length(x) > 1) sapply(x, helper) else helper(x)
    }
```

Figure 2: A function to convert a vector of integers into a vector of strings containing word-equivalents of the integers.