Regular expressions and reshaping using data tables and the nc package

by Toby Dylan Hocking

Abstract Regular expressions are powerful tools for extracting tables from non-tabular text data. Capturing regular expressions that describe information to extract from column names can be especially useful when reshaping a data table from wide (one row with many columns) to tall (one column with many rows). We present the R package **nc** (short for named capture), which provides functions for data reshaping, regular expressions, and a uniform interface to three C libraries (PCRE, RE2, ICU). We describe the main features of **nc**, then provide detailed comparisons with related R packages (**stats**, **utils**, **data.table**, **tidyr**, **reshape2**, **cdata**).

Introduction

Regular expressions are powerful tools for text processing that are available in many programming languages, including R. A regular expression *pattern* defines a set of *matches* in a *subject* string. For example, consider as subjects the column names of the famous iris data set in R, `Sepal.Length` `Petal.Width`, `Species`, etc. The pattern `.*[.].*` matches zero or more non-newline characters, followed by a period, followed by zero or more non-newline characters. It would match `Sepal.Length` and `Petal.Width` but it would not match `Species`.

The focus of this article is patterns with capture groups, which are typically defined using parentheses. For example, the pattern `(.*)[.](.*)` results in the same matches as the pattern in the previous paragraph, and it additionally allows the user to capture and extract the substrings by group index (e.g. group 1 matches `Sepal`, group 2 matches `Length`).

Named capture groups allow extracting the substring by name rather than by index. Using names rather than indices is useful in order to create more readable regular expressions (names document the purpose of each sub-pattern), and to create more readable R code (it is easier to understand the intent of named references than numbered references). For example, the pattern `(?<part>.*)[.](?<dimension>.*)` documents that the flower part appears before the measurement dimension; the `part` group matches `Sepal` and the `dimension` group matches `Length`.

Recently, Hocking (2019a) proposed a new syntax for defining named capture groups in R code. Using this new syntax, named capture groups are specified using named arguments in R, which results in code that is easier to read and modify than capture groups defined in string literals. For example, the pattern in the previous paragraph can be written as `part = ".*", "[.]", dimension = ".*"`. Sub-patterns can be grouped for clarity and/or re-used using lists, and numeric data may be extracted with user-provided type conversion functions.

A main thesis of this article is that regular expressions can greatly simplify the code required to specify wide-to-tall data reshaping operations. For one such operation the input is a "wide" table with many columns, and the desired output is a "tall" table with more rows, and some of the input columns converted into a smaller number of output columns (Figure 1). To clarify the discussion we first define three terms that we will use to refer to the different types of columns involved in this conversion:

Reshape columns contain the data which is present in the same amount but in different shapes in the input and output. There are equivalent terms used in different R packages: `varying` in `utils::reshape`, `measure.vars` in `melt` (**data.table**, **reshape2**), etc.

Copy columns contain data in the input which are each copied to multiple rows in the output (`id.vars` in `melt`).

Capture columns are only present in the output, and contain data which come from matching a capturing regex pattern to the input reshape column names.

For example the wide iris data (W in Figure 1) have four numeric columns to reshape: `Sepal.Length`, `Sepal.Width`, `Petal.Length`, `Petal.Width`. For some purposes (e.g. displaying a histogram of each reshape input column using facets in **ggplot2**) the desired reshaping operation results in a table with a single reshape output column (S in Figure 1), two copied columns, and two columns captured from the names of the reshaped input columns. For other purposes (e.g. scatterplot to compare sepal and petal sizes) the desired reshaping operation results in a table with multiple reshape output columns (M1 with `Sepal` and `Petal` columns in Figure 1), two copied columns, and one column captured from the names of the reshaped input columns. We propose to use the new regular expression syntax of Hocking (2019a), e.g. `part = ".*", "[.]", dimension = ".*"`, to define both types of wide-to-tall

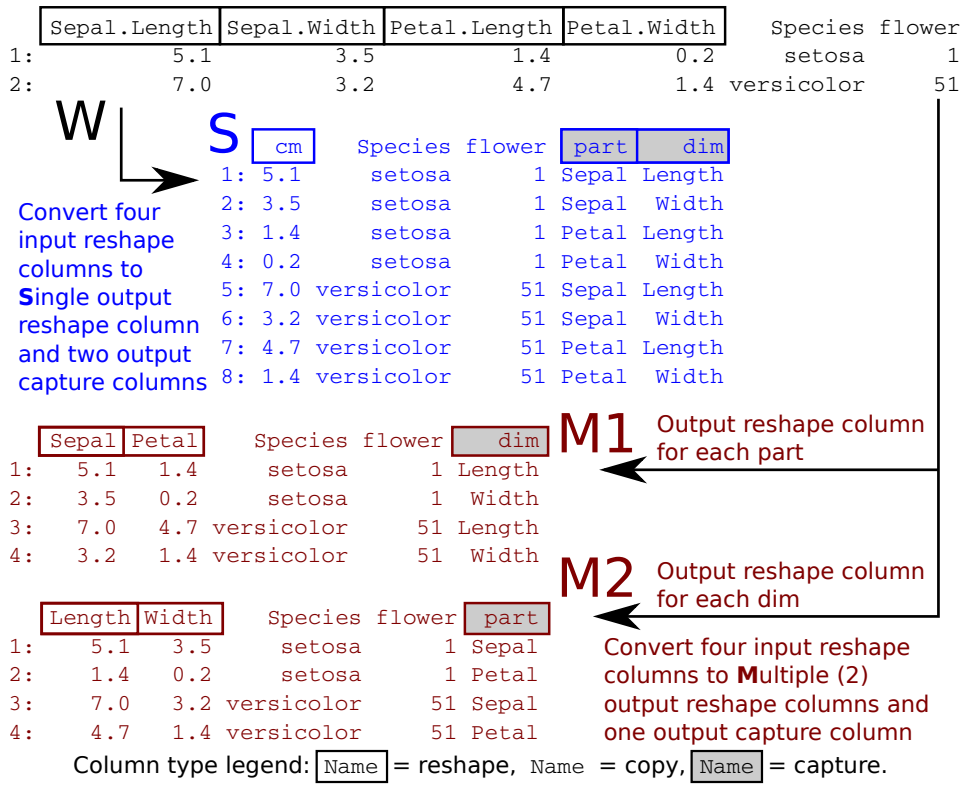


Figure 1: Two rows of the iris data set (W, black) are considered as the input to a wide-to-tall reshape operation. Four input reshape columns are converted to either a single output reshape column (S, blue) or multiple (2) output reshape columns (M1, M2, red). Other output columns are either copied from the non-reshaped input data, or captured from the names of the reshaped input columns.

data reshaping operations. In particular, we propose using a single capturing regular expression for defining both (1) the subset of reshape input columns to convert, and (2) the additional capture output columns. We will show that this results in a simple, powerful, non-repetitive syntax for wide-to-tall data reshaping.

In this article our original contribution is the R package **nc** which provides a new implementation of the previously proposed named capture regex syntax of Hocking (2019a), in addition to several new functions that perform wide-to-tall data reshaping using regular expressions. The main new ideas are (1) using un-named capture groups in the regex pattern string to provide a uniform interface to three regex C libraries, (2) integration of **data.table** functionality (Dowle and Srinivasan, 2019), and (3) specifying wide-to-tall reshape operations with a concise syntax which results in less repetitive user code than other packages. A secondary contribution of this article is a detailed comparison of current R functions for reshaping data, in terms of syntax, computation times, and functionality (Table 1).

The organization of this article is as follows. The rest of this introduction provides an overview of current R packages for regular expressions and data reshaping. The second section describes the proposed functions of the **nc** package, then the third section provides detailed comparisons with other R packages. The article concludes with a summary and discussion.

Related work

There are many R functions which can extract tables from non-tabular text using regular expressions. Recommended R package functions include `base::regexpr` and `base::gregexpr` as well as `utils::strcapture`. CRAN packages which provide various functions for text processing using regular expressions include **namedCapture** (Hocking, 2019b), **rematch2** (Csárdi, 2017), **rex** (Ushey et al., 2017), **stringr** (Wickham, 2018), **stringi** (Gagolewski, 2018), **tidyr** (Wickham and Henry, 2018), and **re2r** (Wenfeng, 2017). We refer the reader to our previous research paper for a detailed comparison of these packages (Hocking, 2019a).

For reshaping data from wide (one row with many columns) to tall (one column with many rows), there are several different R functions that provide similar functionality. Each function supports a different set of features (Table 1); each feature/column is explained in detail below:

pkg::function	single	multiple	regex	na.rm	types	list
nc::capture_melt_multiple	no	unsorted	capture	yes	any	yes
nc::capture_melt_single	yes	no	capture	yes	any	yes
tidyr::pivot_longer	yes	unsorted	capture	yes	some	yes
stats::reshape	yes	sorted	capture	no	some	no
data.table::melt,patterns	yes	sorted	match	yes	no	yes
tidyr::gather	yes	no	no	yes	some	yes
reshape2::melt	yes	no	no	yes	no	no
cdata::rowrecs_to_blocks	yes	unsorted	no	no	no	yes
cdata::unpivot_to_blocks	yes	no	no	no	no	yes
utils::stack	yes	no	no	no	no	no

Table 1: Reshaping functions in R support various features: "single" for converting input columns into a single output column; "multiple" for converting input columns (either "sorted" in a regular order, or "unsorted" for any order) into multiple output columns of different types; "regex" for regular expressions to "match" input column names or to "capture" and create new output column names; "na.rm" for removal of missing values; "types" for converting input column names to non-character output columns; "list" for output of list columns.

single refers to support for converting input reshape columns of the same type to a single reshape output column.

multiple refers to support for converting input reshape columns of possibly different types to multiple output reshape columns; "sorted" means that conversion works correctly only if the input reshape columns are sorted in a regular order, e.g. `Sepal.Length`, `Sepal.Width`, `Petal.Length`, `Petal.Width`; "unsorted" means that conversion works correctly even if the they are not sorted, e.g. `Sepal.Length`, `Sepal.Width`, `Petal.Width`, `Petal.Length`.

regex refers to support for regular expressions; "match" means a pattern is used to match the input column names; "capture" means that the specified pattern is used to create new output capture columns — this is especially useful when the names consist of several distinct pieces of information, e.g. `Sepal.Length`; "no" means that regular expressions are not directly supported (although `base::grep` can always be used).

na.rm refers to support for removing missing values.

types refers to support for converting captured text to numeric output columns.

list refers to support for output of list columns.

Recommended R package functions include `stats::reshape` and `utils::stack` for reshaping data from wide to tall. Of the features listed in Table 1, `utils::stack` only supports output with a single reshape column, whereas `stats::reshape` supports the following features. For data with regular input column names (output column, separator, time value), regular expressions can be used to specify the separator (e.g. in `Sepal.Length`, `Sepal` is output column, dot is separator, `Length` is time value). Multiple output columns are supported, but incorrect output may be computed if input columns are not sorted in a regular order. The time value is output to a capture column named `time` by default. Automatic type conversion is performed on time values when possible, but custom type conversion functions are not supported. There is neither support for missing value removal nor list column output.

The **tidyr** package provides two functions for reshaping data from wide to tall format: `gather` and `pivot_longer`. The older `gather` function only supports converting input reshape columns to a single output reshape column (not multiple). The input reshape columns to convert may not be directly specified using regular expressions; instead R expressions such as `x:y` can be used to indicate all columns starting from `x` and ending with `y`. It does support limited type conversion; if the `convert = TRUE` argument is specified, the `utils::type.convert` function is used to convert the input column names to numeric, integer, or logical. In contrast the newer `pivot_longer` also supports multiple output reshape columns (even if input reshape columns are unsorted), and regular expressions for specifying output capture columns (but to specify input reshape columns with a regex, `grep` must be used). Limited type conversion is also supported in `pivot_longer`, via the `names_ptypes` argument, which should be a list with names corresponding to output columns and values corresponding to prototypes (zero-length atomic vectors, e.g. `numeric()`). Both functions support list columns and removing missing values, although different arguments are used (`na.rm` for `gather`, `values_drop_na` for `pivot_longer`).

The **reshape2** and **data.table** packages each provide a `melt` function for converting data from wide to tall (Wickham, 2007; Dowle and Srinivasan, 2019). The older **reshape2** version only supports converting input reshape columns to a single output reshape column, whereas the newer **data.table** version also supports multiple output reshape columns. Regular expressions are not supported in **reshape2**, but can be used with `data.table::patterns` to match input column names to convert (although the output can be incorrect if columns are not sorted in a regular order). Neither function supports type conversion, and both functions support removing missing values from the output using the `na.rm` argument. List column output is supported in **data.table** but not **reshape2**.

The **cdata** package provides several functions for data reshaping, including `rowrecs_to_blocks` and `unpivot_to_blocks` which can convert data from wide to tall (Mount and Zumel, 2019). The simpler of the two functions is `unpivot_to_blocks`, which supports a single output reshape column (interface similar to `reshape2::melt/tidyr::gather`). The user of `rowrecs_to_blocks` must provide a control table that describes how the input should be reshaped into the output. It therefore supports multiple output reshape columns, for possibly unsorted input columns. Both functions support list column output, but other features from Table 1 are not supported (regular expressions, missing value removal, type conversion).

New features in nc for regular expressions and data reshaping

The **nc** package provides new regular expression functionality based on the syntax recently proposed by Hocking (2019a). In this section we first discuss how the **nc** package implements this syntax as a front-end to three regex engines, and we then discuss the new features for data reshaping using regular expressions.

Uniform interface to three regex engines

Several C libraries providing regular expression engines are available in R. The standard R distribution has included and the Perl-Compatible Regular Expressions (PCRE) C library since 2002 (R Core Team, 2002). CRAN package **re2r** provides the RE2 library, and **stringi** provides the ICU library. Each of these regex engines has a unique feature set, and may be preferred for different applications. For example, PCRE is installed by default, RE2 guarantees matching in polynomial time, and ICU provides strong unicode support. For a more detailed comparison of the relative strengths of each regex library, we refer the reader to our previous research paper (Hocking, 2019a).

Each regex engine has a different R interface, so switching from one engine to another may require non-trivial modifications of user code. In order to make switching between engines easier, Hocking (2019a) introduced the **namedCapture** package, which provides a uniform interface for capturing text using PCRE and RE2. The user may specify the desired engine via an option; the **namedCapture** package provides the output in a uniform format. However **namedCapture** requires the engine to support specifying capture group names in regex pattern strings, and to support output of the group names to R (which ICU does not support).

Our proposed **nc** package provides support for the ICU engine in addition to PCRE and RE2. The **nc** package implements this functionality using un-named capture groups, which are supported in all three regex engines. In particular, a regular expression is constructed in R code that uses named arguments to indicate capturing sub-patterns, which are translated to un-named groups when passed to the regex engine. For example, consider a user who wants to capture the two pieces of the column names of the iris data, e.g. `Sepal.Length`. The user would typically specify the capturing regular expression as a string literal, e.g. `"(.*)[.](.*)"`. Using **nc** the same pattern can be applied to the iris data column names via

```
> nc::capture_first_vec(
+   names(iris), part = ".*", "[.]", dim = ".*", engine = "ICU", nomatch.error = FALSE)

    part    dim
1: Sepal Length
2: Sepal  Width
3: Petal Length
4: Petal  Width
5:  <NA>   <NA>
```

Above we see an example usage of `nc:capture_first_vec`, which is for capturing the first match of a regex from a character vector subject (the first argument). There are a variable number of other arguments (`...`) which are used to define the regex pattern. In this case there are three pattern arguments: `part = ".*","[.]",dim = ".*"`. Each named R argument in the pattern generates an

un-named capture group by enclosing the specified value in parentheses, e.g. `(.*)`. All of the sub-patterns are pasted together in the sequence they appear in order to create the final pattern that is used with the specified regex engine. The `nomatch.error = FALSE` argument is given because the default is to stop with an error if any subjects do not match the specified pattern (the fifth subject `Species` does not match). Under the hood, the following function is called to parse the pattern arguments:

```
> str(compiled <- nc::var_args_list(part = ".*", "[.]", dim = ".*"))
```

```
List of 2
 $ fun.list:List of 2
  ..$ part:function (x)
  ..$ dim :function (x)
 $ pattern : chr "(.*)[.](.*)"
```

This function is intended mostly for internal use, but can be useful for viewing the generated regex pattern (or using it as input to another regex function). The return value is a named list of two elements: `pattern` is the capturing regular expression which is generated based on the input arguments, and `fun.list` is a named list of type conversion functions. Group-specific type conversion functions are useful for converting captured text into numeric output columns (Hocking, 2019a). Note that the order of elements in `fun.list` corresponds to the order of capture groups in the pattern (e.g. first capture group named `part`, second `dim`). These data can be used with any regex engine that supports un-named capture groups (including ICU) in order to get a capture matrix with column names, e.g.

```
> m <- stringi::stri_match_first_regex(names(iris), compiled$pattern)
> colnames(m) <- c("match", names(compiled$fun.list))
> m
```

```
     match          part    dim
[1,] "Sepal.Length" "Sepal" "Length"
[2,] "Sepal.Width"  "Sepal" "Width"
[3,] "Petal.Length" "Petal" "Length"
[4,] "Petal.Width"  "Petal" "Width"
[5,] NA             NA      NA
```

Again, this is not the recommended usage of **nc**, but here we give these details in order to explain how it works. Note that the result from **stringi** is a character matrix with three columns: first for the entire match, and another column for each capture group. Using the same pattern with `base::regexpr` (PCRE engine) or `re2r::re2_match` (RE2 engine) yields output in varying formats. The **nc** package takes care of converting these different results into a standard format which makes it easy to switch regex engines (by changing the value of the `engine` argument).

Note that the standard format used by **nc**, as shown above with `nc::capture_first_vec`, is a data table. The main reason that data tables are always output by **nc** matching functions is in order to support numeric output columns, when type conversion functions are specified. A secondary reason is for the convenient syntax, as we will show in the next section.

Data table integration and `nc::field` to avoid repetition

In the previous section we mentioned that every **nc** matching function outputs a data table, which can be queried/summarized/joined/etc using the convenient syntax of the **data.table** package. In this section we show an example of how this syntax makes it easy to parse non-tabular text output files from a bioinformatics program. We also show an example of how the new `nc::field` function can be used to avoid repetition when defining a pattern to match fields of the form `variable = value`.

We consider a bioinformatics program called SweeD which outputs two data files that are linked by an integer alignment ID number (Pavlidis et al., 2013). The "Report" file is similar to tab-separated values (TSV), except that there are some blank lines, followed by two forward slashes, followed by the integer alignment number. For example, the first few lines of the first two alignments look like:

```
> report.txt.gz <- system.file("extdata", "SweeD_Report.txt.gz", package = "nc")
> report.vec <- readLines(report.txt.gz)
> cat(report.vec[1:5], sep = "\n")
```

```
//1
Position        Likelihood        Alpha
700.0000        4.637328e-03        2.763840e+02
130585.6172        3.781283e-01        8.490200e-04
```

```
> cat(report.vec[1003:1008], sep = "\n")

129756434.0000        3.850623e-01        4.812648e+01

//2
Position        Likelihood        Alpha
135.0000        7.282316e-01        3.163686e+01
111533.0625        2.548831e+00        4.932014e-04
```

The file above can be easily parsed by `nc::capture_all_str`, which is for finding all matches of a regular expression in a multi-line text file: nn

```
> report.alignments <- nc::capture_all_str(
+   report.vec, "//", Alignment = "[0-9]+", TSV = "[^/]+")
> class(report.alignments)

[1] "data.table" "data.frame"

> dim(report.alignments)

[1] 10  2

> nchar(report.alignments$TSV)

 [1] 40170 40028 39996 39946 39924 39914 39907 39906 39894 39892

> substr(as.matrix(report.alignments[1:2]), 1, 50)

     Alignment TSV
[1,] "1"       "\nPosition\tLikelihood\tAlpha\n700.0000\t4.637328e-03\t2"
[2,] "2"       "\nPosition\tLikelihood\tAlpha\n135.0000\t7.282316e-01\t3"
```

The code above computes a data table with ten rows, one for each match in the text file. There are two columns: `Alignment` is a unique identifier, and `TSV` contains the tab-separated values in each block. Each `TSV` entry is a long text string (about 40,000 characters); we display the first 50 characters of the first two blocks above. Because the result is a data table, we can parse the `TSV` using a call to `fread` inside of a by block:

```
> (report.positions <- report.alignments[, data.table::fread(text = TSV), by = Alignment])

       Alignment   Position   Likelihood        Alpha
    1:         1      700.0 4.637328e-03 2.763840e+02
    2:         1   130585.6 3.781283e-01 8.490200e-04
    3:         1   260471.2 3.602315e-02 4.691340e-03
    4:         1   390356.9 7.618749e-01 5.377668e-04
    5:         1   520242.5 2.979971e-08 1.411765e-01
   ---
 9996:        10 82991564.8 8.051006e-03 1.357819e-03
 9997:        10 83074967.8 7.048433e-03 1.825764e-03
 9998:        10 83158370.8 1.012360e-07 7.999999e-03
 9999:        10 83241773.8 3.977189e-08 9.999997e-01
10000:        10 83325174.0 3.980538e-08 1.200000e+03
```

A bioinformatics analyst may desire to plot the `Likelihood` or `Alpha` values in a genome browser, as a function of `Position` along the chromosome. Each alignment has a chromosome name which therefore needs to be added/joined to this table; the second "Info" file has the chromosome name, as well as some other summary statistics for each alignment:

```
> info.txt.gz <- system.file("extdata", "SweeD_Info.txt.gz", package = "nc")
> info.vec <- readLines(info.txt.gz)
> info.vec[24:40]

 [1] " Alignment 1"                  ""
 [3] "\t\tChromosome:\t\tscaffold_0"     "\t\tSequences:\t\t14"
 [5] "\t\tSites:\t\t\t1670366"            "\t\tDiscarded sites:\t1264068"
 [7] ""                              "\t\tProcessing:\t\t155.53 seconds"
 [9] ""                              "\t\tPosition:\t\t8.936200e+07"
[11] "\t\tLikelihood:\t\t4.105582e+02"   "\t\tAlpha:\t\t\t6.616326e-06"
[13] ""                              ""
[15] " Alignment 2"                  ""
[17] "\t\tChromosome:\t\tscaffold_1"
```

Again, the data above can be parsed to a data table via a regular expression:

```
> nc::capture_all_str(
+   info.vec, " ",
+   "Alignment", " ", Alignment = "[0-9]+",
+   "\n\n\t\t",
+   "Chromosome", ":\t\t", Chromosome = ".*")
```

```
    Alignment Chromosome
 1:         1 scaffold_0
 2:         2 scaffold_1
 3:         3 scaffold_2
 4:         4 scaffold_3
 5:         5 scaffold_4
 6:         6 scaffold_5
 7:         7 scaffold_6
 8:         8 scaffold_7
 9:         9 scaffold_8
10:        10 scaffold_9
```

The result above is a data table with ten rows, one for each alignment. There are two columns: the first is the `Alignment` identifier, and the second is the `Chromosome` name. Note the repetition in the code above: `Alignment/Chromosome` appear as capture group names as well as in the pattern string literals. Such repetition can be avoided by using the `nc::field` helper function, which uses its first argument for the capture group name as well as a pattern to match:

```
> (info.alignments <- nc::capture_all_str(
+   info.vec, " ",
+   nc::field("Alignment", " ", "[0-9]+"),
+   "\n\n\t\t",
+   nc::field("Chromosome", ":\t\t", ".*")))
```

```
    Alignment Chromosome
 1:         1 scaffold_0
 2:         2 scaffold_1
 3:         3 scaffold_2
 4:         4 scaffold_3
 5:         5 scaffold_4
 6:         6 scaffold_5
 7:         7 scaffold_6
 8:         8 scaffold_7
 9:         9 scaffold_8
10:        10 scaffold_9
```

Note that the repetition in the code has been removed, and the result is the same (because the generated regex pattern is the same). Also note that the order of the regex string literals is unchanged, so the pattern is as easy to read/understand as the original repetitive version. Internally, `nc::field` generates a pattern by concatenating all of its arguments; the first argument is also used as the capture group name for the third argument. In general, `nc::field` is useful whenever the subject contains fields of the form `variable = value`.

In order to create the desired output table with `Chromosome` and `Position` columns, we join the two data tables:

```
> info.alignments[report.positions, on = "Alignment"]
```

```
       Alignment Chromosome   Position   Likelihood        Alpha
    1:         1 scaffold_0      700.0 4.637328e-03 2.763840e+02
    2:         1 scaffold_0   130585.6 3.781283e-01 8.490200e-04
    3:         1 scaffold_0   260471.2 3.602315e-02 4.691340e-03
    4:         1 scaffold_0   390356.9 7.618749e-01 5.377668e-04
    5:         1 scaffold_0   520242.5 2.979971e-08 1.411765e-01
   ---
 9996:        10 scaffold_9 82991564.8 8.051006e-03 1.357819e-03
 9997:        10 scaffold_9 83074967.8 7.048433e-03 1.825764e-03
 9998:        10 scaffold_9 83158370.8 1.012360e-07 7.999999e-03
 9999:        10 scaffold_9 83241773.8 3.977189e-08 9.999997e-01
10000:        10 scaffold_9 83325174.0 3.980538e-08 1.200000e+03
```

To conclude this section, we have shown that `nc::capture_all_str` inputs a multi-line text file subject, then outputs a data table with one row per match and one column per capture group. We have also shown how these data tables may be further processed (e.g. `fread`, `by = Alignment`) and joined using the convenient data table syntax. Finally, we showed two examples of using `nc::field` to define patterns that match/capture fields of the form `variable = value`.

New nc functions for wide-to-tall data reshaping

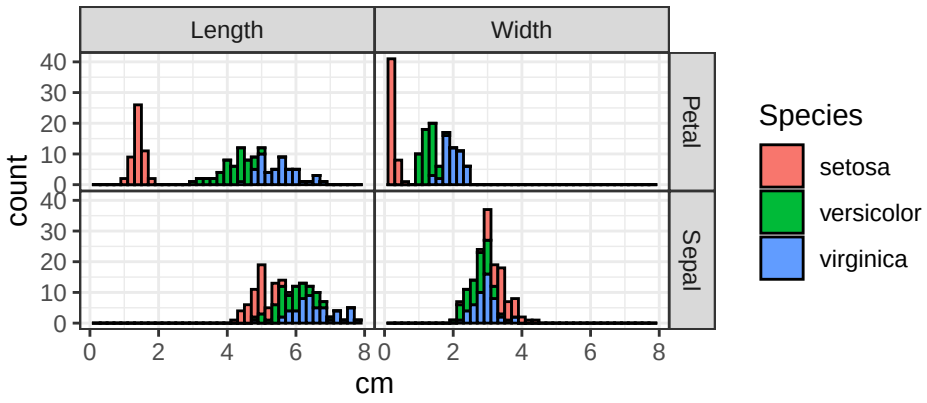
In this section we show how new **nc** functions can be used to reshape wide data (with many columns) to tall data (with fewer columns, and more rows). We begin by considering the two data visualization problems mentioned in the introduction, involving the familiar iris data set. First, suppose we would like to visualize the univariate distribution of each numeric variable. One way would be to use a histogram of each numeric variable, with row facets for flower part and column facets for measurement dimension. Our desired output therefore needs a single column with all of the reshaped numeric data to plot (Figure 1, W→S). We can perform this operation using `nc::capture_melt_single`, which inputs a data frame and a pattern which should match the names of the input columns to reshape. Any input columns with names that do not match the pattern are considered copy columns; the output also contains a capture column for each group specified in the pattern:

```
> (iris.tall.single <- nc::capture_melt_single(
+   iris, part = ".*", "[.]", dim = ".*", value.name = "cm"))

       Species  part    dim  cm
  1:    setosa Sepal Length 5.1
  2:    setosa Sepal Length 4.9
  3:    setosa Sepal Length 4.7
  4:    setosa Sepal Length 4.6
  5:    setosa Sepal Length 5.0
 ---
596: virginica Petal  Width 2.3
597: virginica Petal  Width 1.9
598: virginica Petal  Width 2.0
599: virginica Petal  Width 2.3
600: virginica Petal  Width 1.8
```

The output above consists of one copy column (`Species`), two capture columns (`part`, `dim`), and a single reshape column (`cm`). The `value.name` argument is not considered part of the pattern, and instead specifies the name of the output reshape column. These data can be used to create the desired histogram with **ggplot2** via:

```
> library(ggplot2)
> ggplot(iris.tall.single) + facet_grid(part~dim) +
+   theme_bw() + theme(panel.spacing = grid::unit(0, "lines")) +
+   geom_histogram(aes(cm, fill = Species), color = "black", bins = 40)
```



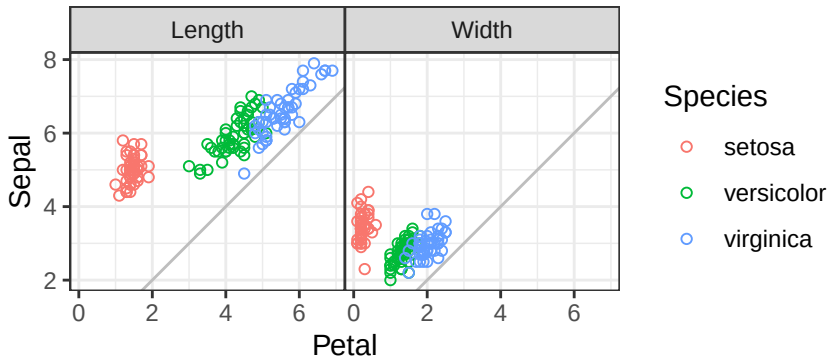
For the second data reshaping task, suppose we want to determine whether or not sepals are larger than petals, for each measurement dimension and species. We could use a scatterplot of sepal versus petal, with a facet for measurement dimension. We therefore need a data table with two reshape output columns: a `Sepal` column to plot against a `Petal` column (Figure 1, W→M1). We can perform this operation using another function, `nc::capture_melt_multiple`, which inputs a data frame and a pattern which must contain the special `column` group and at least one other named group:

```
> (iris.parts <- nc::capture_melt_multiple(
+   iris, column = ".*", "[.]", dim = ".*"))

       Species    dim Petal Sepal
  1:    setosa Length   1.4   5.1
  2:    setosa Length   1.4   4.9
  3:    setosa Length   1.3   4.7
  4:    setosa Length   1.5   4.6
  5:    setosa Length   1.4   5.0
 ---
296: virginica  Width   2.3   3.0
297: virginica  Width   1.9   2.5
298: virginica  Width   2.0   3.0
299: virginica  Width   2.3   3.4
300: virginica  Width   1.8   3.0
```

Again, any input columns with names that do not match the pattern are considered copy columns (`Species`). Each unique value captured in the special `column` group becomes the name of an output reshape column (`Petal`, `Sepal`); other groups are used to create output capture columns (`dim`). These data can be used to create the scatterplot using **ggplot2** via:

```
> ggplot(iris.parts) + facet_grid(.~dim) +
+   theme_bw() + theme(panel.spacing = grid::unit(0, "lines")) +
+   coord_equal() + geom_abline(slope = 1, intercept = 0, color = "grey") +
+   geom_point(aes(Petal, Sepal, color = Species), shape = 1)
```



It is clear from the plot that sepals are larger than petals, for both dimensions, and for every measured flower.

To conclude this section, **nc** provides two new functions for data reshaping using regular expressions. Both functions input a data frame to reshape, and a pattern to match to the column names. For `nc::capture_melt_single`, all matching input columns are reshaped in the output to a single column which is named using the `value.name` argument. For `nc::capture_melt_multiple` the output is multiple reshape columns with names defined by the values captured in the special `column` group. Values from other groups are stored in capture columns in the output. Both functions support output of numeric capture columns via user-specified type conversion functions, as we will see in the next section.

Comparison with other data reshaping packages

In this section we compare the new data reshaping functions in the **nc** package with similar functions in other packages. We aim to demonstrate that the new **nc** syntax is more powerful and less repetitive, without sacrificing speed.

Comparison with tidyr for a single output reshape column

In terms of functionality for wide-to-tall data reshaping, the most similar package to **nc** is **tidyr** (Table 1). One advantage of **nc** is that complex patterns may be defined in terms of simpler subpatterns, which can include group names and type conversion functions. Integrating these three pieces results in a syntax that is easy to read as well; it is more difficult to build and read complex patterns using **tidyr** syntax, which requires specifying regex pattern strings, group names, and types as

separate arguments. Another advantage of **nc** is that the range of possible type conversions for capture output columns is essentially unlimited, since the user may provide an arbitrary type conversion function for each group. In contrast the **tidyr** user may specify a prototype for each group, which results in a limited range of possible conversions. Of course, post-processing may be used, but that introduces some repetition (of capture column names) in the code. For example, consider a data set from the World Health Organization (WHO):

```
> data(who, package = "tidyr")
> set.seed(1);sample(names(who), 10)
```

```
 [1] "newrel_f3544" "year"         "new_ep_m65"   "country"      "new_ep_m1524"
 [6] "new_sn_m4554" "new_ep_f3544" "new_sp_f2534" "new_sp_f65"   "newrel_m4554"
```

Each reshape column name starts with `new` and has three distinct pieces of information: diagnosis type (e.g. `ep`, `rel`), gender (`m` or `f`), and age range (e.g. `1524`, `4554`). We extract all three pieces of information below, and include a function for converting gender to a factor with levels in a specific (non-default) order:

```
> nc.who.sub.pattern <- list(
+   "new_?", diagnosis = ".*", "_",
+   gender = ".", function(mf)factor(mf, c("m", "f")))
> nc.who.ages <- nc::capture_melt_single(who, nc.who.sub.pattern, ages = ".*")
> print(nc.who.ages[1:2], class = TRUE)
```

```
       country   iso2   iso3  year diagnosis gender   ages value
        <char> <char> <char> <int>    <char> <fctr> <char> <int>
1: Afghanistan     AF    AFG  1997        sp      m    014     0
2: Afghanistan     AF    AFG  1998        sp      m    014    30
```

First note that `nc.who.sub.pattern` is a sub-pattern list variable that we have used as the first part of the pattern in the call to `nc::capture_melt_single` above (and we will use that sub-pattern again below). Sub-pattern lists may contain regex character strings (patterns to match), functions (for converting the previous capture group), or other sub-pattern lists. The reshaped output is a data table with `gender` converted to a factor — this can also be done using `tidyr::pivot_longer`:

```
> tidyr.who.sub.pattern <- "new_?(.*)_(.)"                      #L1
> tidyr.who.pattern <- paste0(tidyr.who.sub.pattern, "(.*)")    #L2
> tidyr::pivot_longer(                                          #L3
+   who, cols = grep(tidyr.who.pattern, names(who)),            #L4
+   names_to = c("diagnosis", "gender", "ages"),                #L5
+   names_ptypes = list(gender = factor(levels = c("m", "f"))), #L6
+   names_pattern = tidyr.who.pattern)[1:2,]                    #L7
```

```
# A tibble: 2 x 8
  country     iso2  iso3   year diagnosis gender ages  value
  <chr>       <chr> <chr> <int> <chr>     <fct>  <chr> <int>
1 Afghanistan AF    AFG    1980 sp        m      014      NA
2 Afghanistan AF    AFG    1980 sp        m      1524     NA
```

In the code above we first define a sub-pattern variable for the `diagnosis` and `gender` capture groups, as we did using **nc**. One difference is that the **tidyr** pattern is a string literal with un-named capture groups, whereas the **nc** pattern is a list which includes capture group names as well as type conversion functions. These three parameters are specified as three separate arguments in **tidyr**, which results in some separation (e.g. group names defined on L5 but sub-patterns are defined on L1) and repetition (e.g. gender appears on L5 and L6) in the code. The pattern also must be repeated: first in the `cols` argument (L4) to specify the set of input reshape columns, second in the `names_pattern` argument (L7) to specify the conversion from input reshape column names to output capture column values.

Now suppose we want to extract two numeric columns from `ages`, for example to use as interval-censored outputs in a survival regression. Using **nc** we can use the previously defined sub-pattern (including the previously defined group names and type conversion function) as the first part of a larger pattern:

```
> who.typed <- nc::capture_melt_single(
+   who, nc.who.sub.pattern, ages = list(
+     ymin = "0|[0-9]{2}", as.numeric,
+     ymax = "[0-9]{0,2}", function(x)ifelse(x == "", Inf, as.numeric(x))))
> who.typed[, .(rows = .N), by = .(ages, ymin, ymax)]
```

```
   ages ymin ymax  rows
1:  014    0   14 10882
2: 1524   15   24 10868
3: 2534   25   34 10850
4: 3544   35   44 10875
5: 4554   45   54 10876
6: 5564   55   64 10851
7:   65   65  Inf 10844
```

Note in the code above that each group name, regex pattern string, and corresponding type conversion function appears on the same line — this syntax keeps these three related pieces of information close together, which makes complex patterns easier to read and build from smaller pieces. Also, note how an anonymous function is used to convert the values captured in the `ymax` group to numeric (and it maps the empty string to `Inf`). Such custom type conversion functions are not supported by **tidyr**, so post-processing must be used:

```
> tidyr.who.pattern2 <- paste0(tidyr.who.sub.pattern, "((0|[0-9]{2})([0-9]{0,2}))")
> transform(tidyr::pivot_longer(
+   who, cols = grep(tidyr.who.pattern2, names(who)),
+   names_to = c("diagnosis", "gender", "ages", "ymin", "ymax"),
+   names_ptypes = list(gender = factor(levels = c("m", "f")), ymin = numeric()),
+   names_pattern = tidyr.who.pattern2),
+   ymax = ifelse(ymax == "", Inf, as.numeric(ymax)))[1:7,]

      country iso2 iso3 year diagnosis gender ages ymin ymax value
1 Afghanistan   AF  AFG 1980        sp      m  014    0   14    NA
2 Afghanistan   AF  AFG 1980        sp      m 1524   15   24    NA
3 Afghanistan   AF  AFG 1980        sp      m 2534   25   34    NA
4 Afghanistan   AF  AFG 1980        sp      m 3544   35   44    NA
5 Afghanistan   AF  AFG 1980        sp      m 4554   45   54    NA
6 Afghanistan   AF  AFG 1980        sp      m 5564   55   64    NA
7 Afghanistan   AF  AFG 1980        sp      m   65   65  Inf    NA
```

The code above uses `transform` as a post-processing step to compute a numeric `ymax` column, which involves some repetition (`ymax` appears four times). Note that numeric conversions which do not require special logic (e.g. `ymin` above) can be specified as prototypes using the `names_ptypes` argument (but specifying `ymax = numeric()` as a prototype results in a lossy cast error).

To conclude this comparison, we have seen that defining a complex pattern is much more readable/understandable using **nc** syntax, because it keeps group-specific names and type conversion functions near the corresponding sub-patterns. We have also shown that repetition is often necessary with **tidyr** (e.g. pattern, group names), whereas such repetition can be avoided by using **nc**.

Comparison with data.table and `stats::reshape` for multiple reshape output columns

In this section we demonstrate the advantages of using **nc** over **data.table** (which is used to implement **nc** functionality). A major advantage is that **data.table** only supports regular expressions for defining the set of input columns to reshape; it does not support capture output columns. Another advantage is that **nc** always returns a correct output data set with multiple reshape columns, even when the input columns are not sorted in a regular order. For example, consider the following simple data set in which the columns are not in regular order:

```
> library(data.table)
> (TC <- data.table(
+   treatment.age = 13,
+   control.gender = "M",
+   treatment.gender = "F",
+   control.age = 25))

   treatment.age control.gender treatment.gender control.age
1:            13              M               F          25
```

It is clear from the table above that the treatment group consists of a teenage female, whereas the control group consists of a male aged 25 (not the best experimental design, but easy to remember for the demonstration in this section). Assume we need an output data table with two reshape columns (`age` and `gender`) as well as a capture column (`group`). The **nc** syntax we would use is:

```
> nc::capture_melt_multiple(TC, group = ".*", "[.]", column = ".*")

       group age gender
1:   control  25      M
2: treatment  13      F
```

The correct result is computed above because **nc** reshapes based on the input column names (the order of the input columns is not relevant). A naïve user may attempt to perform this reshape using `data.table::patterns`:

```
> melt(TC, measure.vars = patterns(age = "age", gender = "gender"))

   variable age gender
1:        1  13      M
2:        2  25      F
```

First, note that the syntax above requires repetition of `age` and `gender` (in names and in pattern strings). Also it is clear that the result is incorrect! Actually, the `patterns` function is working as documented; it "returns the matching indices" of the provided regex. However, since the input columns are not sorted in regular order, `melt` returns an incorrect result. To get a correct result, we can provide a list of index vectors:

```
> melt(TC, measure.vars = list(age = c(1,4), gender = c(3,2)))

   variable age gender
1:        1  13      F
2:        2  25      M
```

This is what **nc** does internally; it also converts the `variable` output column to a more interpretable/useful capture column (e.g. `group` above).

The `stats::reshape` function suffers from the same issue. Another issue with this function is that it assumes the output reshape column names are the first part of the input column names (e.g. Figure 1, W→M1). When input column names have a different structure (e.g. Figure 1, W→M2), they must be renamed, putting the desired output reshape column names first:

```
> TC.renamed <- structure(TC, names = sub("(.*)[.](.*)", "\\2.\\1", names(TC)))
> stats::reshape(TC.renamed, 1:4, direction = "long", timevar = "group")

       group age gender id
1: treatment  13      M  1
2:   control  25      F  1
```

However the result above still contains incorrect results in the `gender` column. The correct result can be obtained by sorting the input column names:

```
> TC.sorted <- data.frame(TC.renamed)[, sort(names(TC.renamed))]
> stats::reshape(TC.sorted, 1:4, direction = "long", timevar = "group")

                group age gender id
1.control     control  25      M  1
1.treatment treatment  13      F  1
```

After renaming and sorting the input columns, the correct result is obtained using `stats::reshape`. Another way to obtain a correct result is with the **cdata** package:

```
> cdata::rowrecs_to_blocks(TC, controlTable = data.frame(
+   group = c("treatment", "control"),
+   age = c("treatment.age", "control.age"),
+   gender = c("treatment.gender", "control.gender"),
+   stringsAsFactors = FALSE))

      group age gender
1 treatment  13      F
2   control  25      M
```

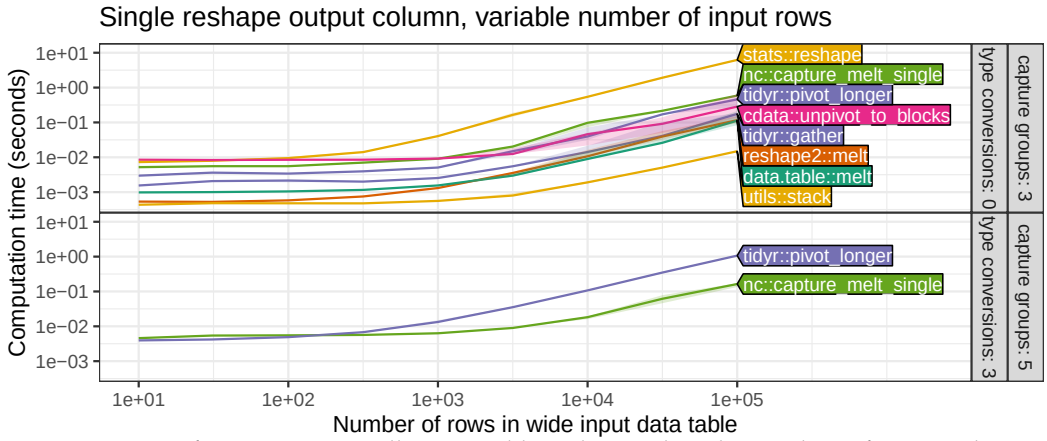


Figure 2: Timings for computing a tall output table with a single reshape column from a wide input table with 56 reshape columns and a variable number of rows (x axis). **Top** panel shows functions which provide some wide-to-tall data reshaping functionality; **Bottom** panel shows only the functions which output capture columns based on a regular expression.

The **cdata** package is very powerful, and can handle many more types of data reshaping operations than **nc**. However, it requires a very explicit definition of the desired conversion in terms of a control table, which results in rather verbose code. In contrast, the terse regular expression syntax of **nc** is a more implicit approach, which assumes the input columns to reshape have regular names.

To conclude this section, we have discussed some advantages of **nc** relative to other R packages. Regular expressions can be used in **nc** to specify capture output columns, which are not provided in the results from other functions such as `data.table::melt`. Input columns with regular names do not need to be renamed/sorted for **nc** functions, whereas renaming/sorting may be necessary using other functions (`data.table::melt`, `stats::reshape`). Verbose/explicit control table code is always necessary with **cdata**, whereas a terse/implicit regular expression syntax is used with **nc** to simplify the definition of reshape operations.

Comparing computation times of functions for wide-to-tall data reshaping

In previous sections we have showed that the **nc** package provides a convenient syntax for defining wide-to-tall reshape operations. In this section we investigate whether this convenience comes at the cost of increased computation time. We aim to demonstrate that computation time required for the proposed **nc** package is comparable with other packages for data reshaping. In particular, since **nc** is implemented using **data.table**, we expect that **nc** should be slightly slower than **data.table** (by only constant factors). In our result figures we show the median and quartiles over 10 timings using the **microbenchmark** package on an Intel Core i7-8700 3.20GHz processor. We varied the number of rows/columns in each experiment by copying/duplicating the rows/columns in each source data set. The package versions that we used were:

```
> compare.pkgs <- c("nc", "tidyr", "cdata", "data.table", "reshape2")
> sapply(compare.pkgs, function(x)paste(packageVersion(x)))

          nc        tidyr        cdata   data.table     reshape2
"2019.10.19"      "1.0.0"      "1.1.2"     "1.12.6"      "1.4.3"

> R.version$version.string

[1] "R version 3.6.1 (2019-07-05)"
```

First, we performed timings on a version of the WHO data with a variable number of rows, and the original number of columns (56). We first consider a pattern with three capture groups and no type conversions. We ran reshaping functions from several packages (**nc**, **stats**, **utils**, **tidyr**, **reshape2**, **data.table**, **cdata**) that can compute the desired output table with a single reshape output column. As we expected, all functions appear to have similar asymptotic time complexity, and differ only in terms of constant factors (Figure 2, top). The slowest function was `stats::reshape` (about 10 seconds for 10^5 rows) and the fastest function was `utils::stack` (about 30 milliseconds). The other functions, including `nc::capture_melt_single`, showed intermediate speeds (about 100 milliseconds to 1 second).

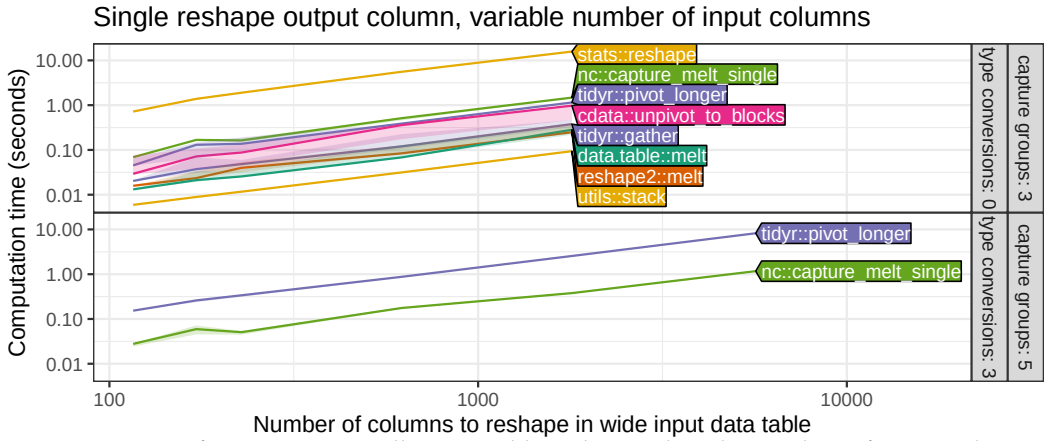


Figure 3: Timings for computing a tall output table with a single reshape column from a wide input table with 7240 rows and a variable number of columns to reshape (x axis). **Top** panel shows functions which provide some wide-to-tall data reshaping functionality; **Bottom** panel shows only the functions which output capture columns based on a regular expression.

We also tried using a regular expression with five capture groups and three type conversions for capture output columns (`gender` to factor, `ymin` and `ymax` to numeric). In this test, we ran only `tidyr::pivot_longer` and `nc::capture_melt_single` because these are the only two functions which natively support capture output columns. They both appear to have similar asymptotic time complexity (Figure 3, bottom), although `nc::capture_melt_single` (mean $\pm$ sd, 0.327 ± 0.106 seconds for 10^5 rows) is about five times faster than `tidyr::pivot_longer` (1.774 ± 0.147).

Second, we performed similar timings on variants of the WHO data with a variable number of columns, and the original number of rows (7240). The desired output again has a single reshape output column, and we tried the same two regular expressions and type conversions as in the tests described earlier in this section. We observed similar asymptotic trends as in the previous comparison (Figure 3). These data indicate that for converting data into a single reshape output column, **nc** has similar or faster speeds than comparable R packages.

Third, we performed timings on variants of the iris data with a variable number of rows, and twice the original number of reshape columns (8). The input reshape column names were of the form `day1.Sepal.Length`, `day2.Sepal.Length`, `day1.Sepal.Width`, etc. Since the desired output has two reshape columns (`Sepal` and `Petal`), we considered packages which support multiple output columns (**cdata**, **stats**, **tidyr**, **nc**, **data.table**). As in the previous tests, we tried one experiment without type conversions (for which we tested all packages), and one experiment with converting the `day` capture group to an integer (for which we only tested **tidyr** and **nc**). In both experiments we observed that all algorithms have similar asymptotic time complexity (Figure 4). We observed that **nc** is slightly slower than **data.table** (by constant factors), and slightly faster than the other packages (**cdata**, **stats**, **tidyr**)

Finally, we performed similar timings on variants of the iris data with a variable number of columns, and the original number of rows (150). Again there are two desired reshape output columns, and we tested a regular expression with three capture groups (with or without type conversion). As in previous experiments, we expected that all functions would have similar asymptotic time complexity. Surprisingly, we observed on the log-log plot (Figure 5) that **nc** and **data.table** have smaller asymptotic slopes (1.009–1.061) than the other packages (1.805–2.232). These data suggest that the computation time of **nc** and **data.table** is linear $O(C)$ for C input reshape columns, whereas the other packages are quadratic $O(C^2)$. In practice this means that the speed of **nc** and **data.table** relative to the other packages increases as the number of input reshape columns C increases. For example with $C = 4 \times 10^5$ input reshape columns, **nc** takes only about 15 seconds, wheras **tidyr** takes over an hour. Therefore **nc** or **data.table** should be preferred over the other packages (**cdata**, **stats**, **tidyr**) for fast conversion into multiple reshape output columns when there are many input reshape columns. Overall, our experiments indicate that for converting data to multiple reshape output columns, **nc** has similar or faster speeds than comparable R packages.

Discussion and conclusions

In this paper we described the **nc** package and its new functions for regular expressions and data reshaping. The **nc** package allows a user to define a regular expression in R code, along with capture group names and corresponding type conversion functions. We showed how this syntax makes it easy

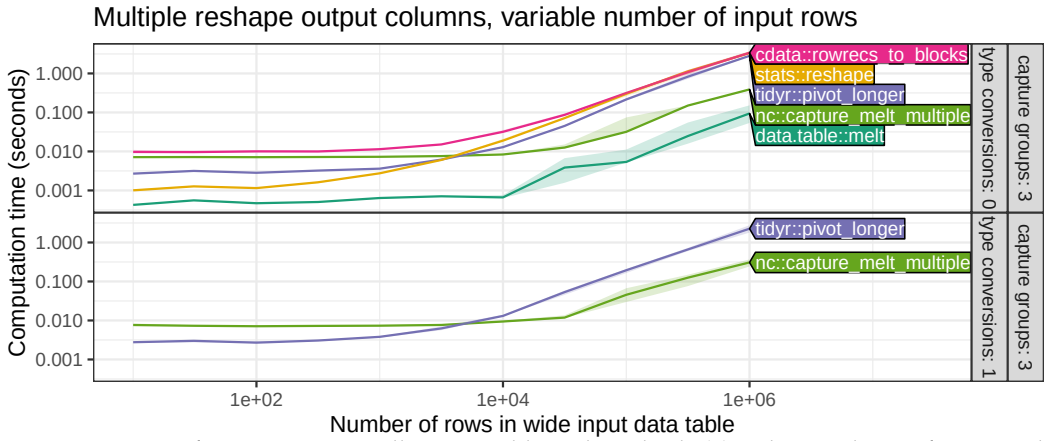


Figure 4: Timings for computing a tall output table with multiple (2) reshape columns from a wide input table with 8 reshape columns and a variable number of rows (x axis). **Top** panel shows functions which provide some wide-to-tall data reshaping functionality; **Bottom** panel shows only the functions which output capture columns based on a regular expression.

to define complex regular expressions in terms of simpler sub-patterns, while providing a uniform interface to three regex engines (ICU, PCRE, RE2). We showed several examples of how **nc** can be used for parsing non-tabular text data, and for wide-to-tall data reshaping. We provided a detailed comparison with other data reshaping functions in terms of syntax, functionality, and computation time.

In all of our speed comparisons, we observed that **nc** is at least as fast as comparable R packages. In general, we observed that other functions which provide fewer features are slightly faster, and other functions which provide more features are slightly slower (by constant factors). For example, the fastest function for converting data into a single reshape output column was `utils::stack`. This empirical observation is consistent with our theoretical assessment of this function (Table 1); since it provides the fewest features, it does the least amount of work, and should be the fastest.

The `tidyr::pivot_longer` function provides a feature set which is most similar to **nc** data reshaping functions. We showed that both packages can perform the same data reshaping operations, but **nc** provides a syntax that reduces repetition in user code. When the input reshape column names are regular, **nc** may be preferable in order to avoid repetition and to support a wider range of possible type conversions. Also, our empirical timings suggest that **nc** is faster when there are type conversions, or when there are many input reshape columns and multiple output reshape columns.

We observed a surprising result in our empirical timings with multiple reshape output columns and a variable number of input reshape columns C. We expected that all of the packages would have similar asymptotic timings (differing only in constant factors). However for large C we observed computation times in **nc** and its dependent package **data.table** that were linear $O(C)$, which was much faster than the other packages (**cdata**, **stats**, **tidyr**) which were quadratic $O(C^2)$. This result suggests that the speed of these other packages could be improved by adopting the linear time algorithm used in the **data.table** package.

In **nc** there are two different functions for wide-to-tall data reshaping: `nc::capture_melt_single` computes a single output reshape column, and `nc::capture_melt_multiple` computes multiple output reshape columns. In contrast, other functions that support multiple output reshape columns also support a single output reshape column (Table 1). It is natural to ask whether these two **nc** functions could be combined into a single function that could handle both kinds of output. Of course it is possible, but we prefer to keep the two functions separate in order to provide more specific/informative documentation, examples, and error messages.

We have shown how the **nc** package provides a powerful and efficient new syntax for wide-to-tall data reshaping using regular expressions. The inverse operation, tall-to-wide data reshaping, is not supported. For tall-to-wide reshaping operations, we recommend using the efficient implementation in `data.table::dcast`. For future work, we will be interested to explore other operations related to machine learning and data visualization which could be simplified using regular expressions.

Acknowledgements. Thanks to Marc Tollis for providing the example data output from the SweeD program.

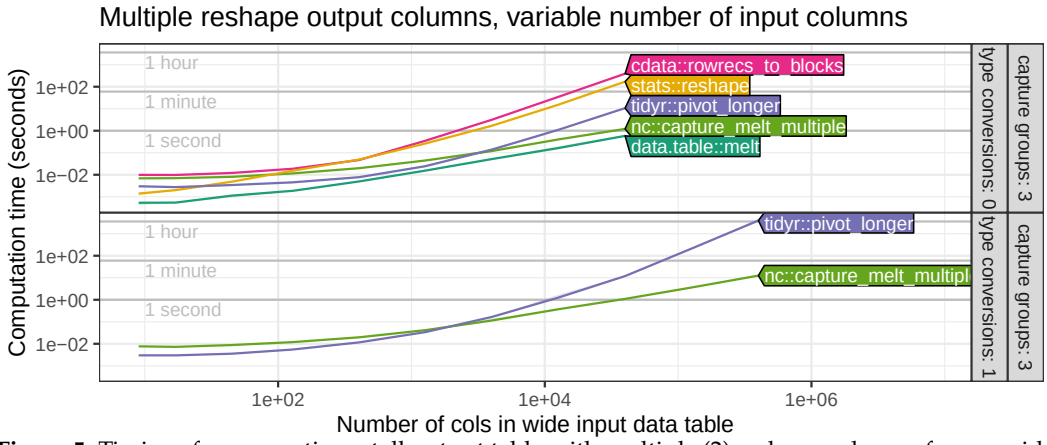


Figure 5: Timings for computing a tall output table with multiple (2) reshape columns from a wide input table with 150 rows and a variable number of columns to reshape (x axis). Timings for wide-to-tall data reshaping on an input data table with 150 rows and a variable number of columns to reshape (x axis). **Top** panel shows functions which provide some wide-to-tall data reshaping functionality; **Bottom** panel shows only the functions which output capture columns based on a regular expression.

Reproducible research statement. The source code for this article can be freely downloaded from https://github.com/tdhock/nc-article

Bibliography

G. Csárdi. *rematch2: Tidy Output from Regular Expression Matching*, 2017. URL https://CRAN.R-project.org/package=rematch2. R package version 2.0.1. [p2]

M. Dowle and A. Srinivasan. *data.table: Extension of 'data.frame'*, 2019. http://r-datatable.com. [p2, 4]

M. Gagolewski. *R package stringi: Character string processing facilities*, 2018. URL http://www.gagolewski.com/software/stringi/. [p2]

T. D. Hocking. Comparing namedcapture with other r packages for regular expressions. *R Journal*, 2019a. [p1, 2, 4, 5]

T. D. Hocking. *namedCapture: Named Capture Regular Expressions*, 2019b. R package version 2019.01.14. [p2]

J. Mount and N. Zumel. *cdata: Fluid Data Transformations*, 2019. URL https://CRAN.R-project.org/package=cdata. R package version 1.1.2. [p4]

P. Pavlidis, D. Živković, A. Stamatakis, and N. Alachiotis. SweeD: Likelihood-Based Detection of Selective Sweeps in Thousands of Genomes. *Molecular Biology and Evolution*, 30(9):2224–2234, 06 2013. ISSN 0737-4038. doi: 10.1093/molbev/mst112. [p5]

R Core Team. News for the 1.x series, 2002. URL https://github.com/tdhock/regex-tutorial/blob/master/R.NEWS.1.txt. [p4]

K. Ushey, J. Hester, and R. Krzyzanowski. *rex: Friendly Regular Expressions*, 2017. URL https://CRAN.R-project.org/package=rex. R package version 1.1.2. [p2]

Q. Wenfeng. *re2r: RE2 Regular Expression*, 2017. URL https://CRAN.R-project.org/package=re2r. R package version 0.2.0. [p2]

H. Wickham. Reshaping data with the reshape package. *Journal of Statistical Software*, 21(12):1–20, 2007. URL http://www.jstatsoft.org/v21/i12/. [p4]

H. Wickham. *stringr: Simple, Consistent Wrappers for Common String Operations*, 2018. URL https://CRAN.R-project.org/package=stringr. R package version 1.3.1. [p2]

H. Wickham and L. Henry. *tidyr: Easily Tidy Data with 'spread()' and 'gather()' Functions*, 2018. URL https://CRAN.R-project.org/package=tidyr. R package version 0.8.2. [p2]

Toby Dylan Hocking
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
Flagstaff, Arizona
USA
toby.hocking@nau.edu