Journal of Statistical Software
April 2011, Volume 40, Issue 3. http://www.jstatsoft.org/

Dates and Times Made Easy with lubridate

Garrett Grolemund
Rice University

Hadley Wickham
Rice University

Abstract

This paper presents the lubridate package for R, which facilitates working with dates and times. Date-times create various technical problems for the data analyst. The paper highlights these problems and offers practical advice on how to solve them using **lubridate**. The paper also introduces a conceptual framework for arithmetic with date-times in R.

Keywords: dates, times, time zones, daylight savings time, R.

1. Introduction

Date-time data can be frustrating to work with. Dates come in many different formats, which makes recognizing and parsing them a challenge. Will our program recognize the format that we have? If it does, we still face problems specific to date-times. How can we easily extract components of the date-times, such as years, months, or seconds? How can we switch between time zones, or compare times from places that use daylight savings time (DST) with times from places that do not? Date-times create even more complications when we try to do arithmetic with them. Conventions such as leap years and DST make it unclear what we mean by "one day from now" or "exactly two years away." Even leap seconds can disrupt a seemingly simple calculation. This complexity affects other tasks too, such as constructing sensible tick marks for plotting date-time data.

While base R (R Development Core Team 2011) handles some of these problems, the syntax it uses can be confusing and difficult to remember. Moreover, the correct R code often changes depending on the class of date-time object being used. **lubridate** acknowledges these problems and makes it easier to work with date-time data in R. It also provides tools for manipulating date-times in novel but useful ways. **lubridate** will enhance a user's experience for any analysis that includes date-time data. Specifically, **lubridate** helps users:

- Identify and parse date-time data, see Section 3.
- Extract and modify components of a date-time, such as years, months, days, hours, minutes, and seconds, see Section 4.
- Perform accurate calculations with date-times and timespans, see Sections 5 and 6.
- Handle time zones and daylight savings time, see Sections 7 and 8.

lubridate uses an intuitive user interface inspired by the date libraries of object-oriented programming languages. **lubridate** methods are compatible with a wide-range of common date-time and time series objects. These include `character` strings, `POSIXct`, `POSIXlt`, `Date`, **chron** (James and Hornik 2010), **timeDate** (Wuertz and Chalabi 2010a), **zoo** (Zeileis and Grothendieck 2005), **xts** (Ryan and Ulrich 2010), **its** (Portfolio and Risk Advisory Group, Commerzbank Securities 2009), **tis** (Hallman 2010), **timeSeries** (Wuertz and Chalabi 2010b), **fts** (Armstrong 2009), and **tseries** (Trapletti and Hornik 2009) objects.

Note that **lubridate** overrides the `+` and `-` methods for `POSIXt`, `Date`, and `difftime` objects in base R. This allows users to perform simple arithmetic on date-time objects with the new timespan classes introduced by **lubridate**, but it does not alter the way R implements addition and subtraction for non-**lubridate** objects.

lubridate introduces four new object classes based on the Java language **Joda Time** project (Colebourne and O'Neill 2010). **Joda Time** introduces a conceptual model of the different ways to measure timespans. Section 5 describes this model and explains how **lubridate** uses it to perform easy and accurate arithmetic with dates in R.

This paper demonstrates the convenient tools provided in the **lubridate** package and ends with a case study, which uses **lubridate** in a real life example. This paper describes **lubridate** 0.2, which can be downloaded from the Comprehensive R Archive Network at `http://CRAN.R-project.org/package=lubridate`. Development versions can be found at `http://github.com/hadley/lubridate`.

2. Motivation

To see how **lubridate** simplifies things, consider a common scenario. Given a character string, we would like to read it in as a date-time, extract the month, and change it to February (i.e, 2). Table 1 shows two ways we could do this. On the left are the base R methods we would use for these three tasks. On the right are the **lubridate** methods.

Now we will go a step further. In Table 2, we move our date back in time by one day and display our new date in the Greenwich Meridian time zone (GMT). Again, base R methods are shown on the left, **lubridate** methods on the right.

lubridate makes basic date-time manipulations much more straightforward. Plus, the same **lubridate** methods work for most of the popular date-time object classes (`Date`, `POSIXt`, `chron`, etc.), which is not always true for base R methods.

Table 3 provides a more complete comparison between **lubridate** methods and base R methods. It shows how **lubridate** can simplify each of the common date-time tasks presented in the article "Date and Time Classes in R" (Grothendieck and Petzoldt 2004). It also provides a useful summary of **lubridate** methods.

Base R method	**lubridate** method
`date <- as.POSIXct("01-01-2010", format = "%d-%m-%Y", tz = "UTC")`	`date <- dmy("01-01-2010")`
`as.numeric(format(date, "%m"))` or `as.POSIXlt(date)$month + 1`	`month(date)`
`date <- as.POSIXct(format(date, "%Y-2-%d"), tz = "UTC")`	`month(date) <- 2`

Table 1: **lubridate** provides a simple way to parse a date into R, extract the month value and change it to February.

Base R method	**lubridate** method
`date <- seq(date, length = 2, by = "-1 day")[2]`	`date <- date - days(1)`
`as.POSIXct(format(as.POSIXct(date), tz = "UTC"), tz = "GMT")`	`with_tz(date, "GMT")`

Table 2: **lubridate** easily displays a date one day earlier and in the GMT time zone.

3. Parsing date-times

We can read dates into R using the `ymd()` series of functions provided by **lubridate**. These functions parse character strings into dates. The letters y, m, and d correspond to the year, month, and day elements of a date-time. To read in a date, choose the function name that matches the order of elements in your date-time object. For example, in the following date the month element comes first, followed by the day and then the year. So we would use the `mdy()` function:

```
R> mdy("12-01-2010")

[1] "2010-12-01 UTC"
```

The same character string can be parsed as January 12, 2001 by reversing the month and day element with `dmy()`.

```
R> dmy("12-01-2010")

[1] "2010-01-12 UTC"
```

The `ydm()` series of functions can also parse vectors of dates.

```
R> dmy(c("31.12.2010", "01.01.2011"))
```

Task	**lubridate**	Date	POSIXct
now (system time zone)	`now()`		`Sys.time()`
now (GMT)	`now("GMT")`	`Sys.Date()`	
origin	`origin`	`structure(0, class = "Date")`	`structure(0, class = c("POSIXt", "POSIXct"))`
x days since origin	`origin + days(x)`	`structure(floor(x), class = "Date")`	`structure(x*24*60*60, class=c("POSIXt","POSIXct"))`
next day	`date + days(1)`	`date + 1`	`seq(date, length = 2, by = "day")[2]`
previous day	`date - days(1)`	`date - 1`	`seq(date, length = 2, by = "-1 day")[2]`
		DST and time zones	
x days since date			
(day exactly 24 hours)	`date + ddays(x)`		`seq(date, length = 2, by = paste(x, "day"))[2]`
(allowing for DST)	`date + days(x)`	`date + floor(x)`	`seq(date, length = 2, by = paste(x,"DSTday"))[2]`
display date in new time zone	`with_tz(date, "TZ")`		`as.POSIXct(format(as.POSIXct(date), tz = "TZ"), tz = "TZ")`
keep clock time, replace time zone	`force_tz(date, tz = "TZ")`		
		Exploring	
sequence	`date + c(0:9) * days(1)`	`seq(date, length = 10, by = "day")`	`seq(date, length = 10, by = "DSTday")`
every 2nd week	`date + c(0:2) * weeks(2)`	`seq(date, length = 3, by = "2 week")`	`seq(date, length = 3, by = "2 week"`
first day of month	`floor_date(date, "month")`	`as.Date(format(date, "%Y-%m-01"))`	`as.POSIXct(format(date, "%Y-%m-01"))`
round to nearest first of month	`round_date(date,"month")`		
extract year value	`year(date)`	`as.numeric(format(date,"%Y"))`	`as.numeric(format(date, "%Y"))`
change year value	`year(date) <- z`	`as.Date(format(date, "z-%m-%d"))`	`as.POSIXct(format(date, "z-%m-%d"))`
day of week	`wday(date) # Sun = 1`	`as.numeric(format(date,"%w")) # Sun = 0`	`as.numeric(format(date, "%w")) # Sun = 0`
day of year	`yday(date)`	`as.numeric(format(date, "%j"))`	`as.numeric(format(date, "%j"))`
express as decimal of year	`decimal_date(date)`		
		Parsing dates	
`z = "1970-10-15"`	`ymd(z)`	`as.Date(z)`	`as.POSIXct(z)`
`z = "10/15/1970"`	`mdy(z)`	`as.Date(z, "%m/%d/%Y")`	`as.POSIXct(strptime(z, "%m/%d/%Y"))`
`z = 15101970`	`dmy(z)`	`as.Date(as.character(z), format = "%d%m%Y")`	`as.POSIXct(as.character(z),tz ="GMT", format = "%d%m%Y")`

Durations comparison

Duration	**lubridate**	Base R
1 second	`seconds(1)`	`as.difftime(1, unit = "secs")`
5 days, 3 hours and - 1 minute	`new_duration(day = 5, hour = 3, minute = -1)`	`as.difftime(60 * 24 * 5 + 60 * 3 - 1, unit = "mins")` `# Time difference of 7379 mins`
1 month	`months(1)`	
1 year	`years(1)`	

Table 3: **lubridate** provides a simple alternative for many date and time related operations. Table adapted from Grothendieck and Petzoldt (2004).

Order of elements in date-time	Parse function
year, month, day	`ymd()`
year, day, month	`ydm()`
month, day, year	`mdy()`
day, month, year	`dmy()`
hour, minute	`hm()`
hour, minute, second	`hms()`
year, month, day, hour, minute, second	`ymd_hms()`

Table 4: Parse function names are based on the order that years, months, and days appear within the dates to be parsed.

```
[1] "2010-12-31 UTC" "2011-01-01 UTC"
```

These functions create a `POSIXct` date-time object that matches the date described by the character string. The functions automatically recognize the separators commonly used to record dates. These include: "-", "/", ".", and "" (i.e., no separator). When a `ymd()` function is applied to a vector of dates, **lubridate** will assume that all of the dates have the same order and the same separators. `ymd()` type functions also exist for times recorded with hours, minutes, and seconds. Hour, minute, and second measurements that are not accompanied by a date will be parsed as `period` objects, which are a type of timespan object, see Section 5.4. These functions make it simple to parse any date-time object that can be converted to a character string. See Table 4 for a complete list of `ymd()` type parsing functions.

4. Manipulating date-times

Every date-time is a combination of different elements, each with its own value. For example, most date-times include a year value, a month value, a day value and so on. Together these elements specify the exact moment that the date-time refers to. We can easily extract each element of a date-time with the accessor function that has its name, as shown in Table 5. For example, if we save the current system time

```
R> date <- now()
```

```
[1] "2010-02-25 09:51:48 CST"
```

we can extract each of its elements. Note that this was the system time when this example was written. `now()` will return a different date-time each time it is used.

```
R> year(date)
```

```
[1] 2010
```

```
R> minute(date)
```

Date component	Accessor
Year	`year()`
Month	`month()`
Week	`week()`
Day of year	`yday()`
Day of month	`mday()`
Day of week	`wday()`
Hour	`hour()`
Minute	`minute()`
Second	`second()`
Time zone	`tz()`

Table 5: Each date-time element can be extracted with its own accessor function.

```
[1] 51
```

For the month and weekday elements (wday), we can also specify whether we want to extract the numerical value of the element, an abbreviation of the name of the month or weekday, or the full name. For example,

```
R> month(date)

[1] 2

R> month(date, label = TRUE)

[1] Feb

R> month(date, label = TRUE, abbr = FALSE)

[1] February

R> wday(date, label = TRUE, abbr = FALSE)

[1] Thursday
```

We can also use any of the accessor functions to set the value of an element. This would also change the moment that the date-time refers to. For example,

```
R> day(date) <- 5

[1] "2010-02-05 09:51:48 CST"
```

changes our date to the fifth day of the month. We can also set the elements to more complicated values, e.g.,

```
R> dates <- ymd_hms("2010-01-01 01:00:00", "2010-01-01 01:30:00")
R> minute(dates) <- mean(minute(dates))

[1] "2010-01-01 01:15:00 UTC" "2010-01-01 01:15:00 UTC"
```

Note that if we set an element to a larger value than it supports, the difference will roll over into the next higher element. For example,

```
R> day(date) <- 30

[1] "2010-03-02 09:51:48 CST"
```

Setting the date elements provides one easy way to find the last day of a month. An even easier method is described in Section 6.

```
R> day(date) <- 1
R> month(date) <- month(date) + 1
R> day(date) <- day(date) - 1

[1] "2010-03-31 09:51:48 CDT"
```

lubridate also provides an update method for date-times. This is useful if you want to change multiple attributes at once or would like to create a modified copy instead of transforming in place.

```
R> update(date, year = 2010, month = 1, day = 1)

[1] "2010-01-01 09:51:48 CST"
```

Finally, we can also change dates by adding or subtracting units of time from them. For example, the methods below produce the same result.

```
R> hour(date) <-  12

[1] "2010-02-25 12:51:48 CST"

R> date <- date + hours(3)

[1] "2010-02-25 12:51:48 CST"
```

Notice that `hours()` (plural) is not the same function as `hour()` (singular). `hours()` creates a new object that can be added or subtracted to a date-time. These objects are discussed in the next section.

5. Arithmetic with date-times

Arithmetic with date-times is more complicated than arithmetic with numbers, but it can be done accurately and easily with **lubridate**. What complicates arithmetic with date-times? Clock times are periodically re-calibrated to reflect astronomical conditions, such as the hour of daylight or the Earth's tilt on its axis relative to the sun. We know these re-calibrations as daylight savings time, leap years, and leap seconds. Consider how one of these conventions might complicate a simple addition task. If today were January 1st, 2010 and we wished to know what day it would be one year from now, we could simply add 1 to the years element of our date.

January 1st, 2010 + 1 year = January 1st, 2011

Alternatively, we could add 365 to the days element of our date because a year is equivalent to 365 days.

January 1st, 2010 + 365 days = January 1st, 2011

Troubles arise if we try the same for January 1st, 2012. 2012 is a leap year, which means it has an extra day. Our two approaches above now give us different answers because the length of a year has changed.

January 1st, 2012 + 1 year = January 1st, 2013
January 1st, 2012 + 365 days = December 31st, 2012

At different moments in time, the lengths of months, weeks, days, hours, and even minutes will also vary. We can consider these to be *relative* units of time; their length is relative to when they occur. In contrast, seconds always have a consistent length. Hence, seconds are *exact* units of time.

Researchers may be interested in exact lengths, relative lengths, or both. For example, the speed of a physical object is most precisely measured in exact lengths. The opening bell of the stock market is more easily modeled with relative lengths.

lubridate allows arithmetic with both relative and exact units by introducing four new time related objects. These are *instants*, *intervals*, *durations*, and *periods*. These concepts are borrowed from the **Joda Time** project (Colebourne and O'Neill 2010). Similar concepts for instants, periods, and durations also appear in the C++ library **Boost.Date_Time** (Garland 2011). **lubridate** provides helper functions, object classes and methods for using all four concepts in the R language.

5.1. Instants

An instant is a specific moment in time, such as January 1st, 2012. We create an instant each time we parse a date into R.

```
R> start_2012 <- ymd_hms("2012-01-01 12:00:00")
```

lubridate does not create a new class for instant objects. Instead, it recognizes any date-time object that refers to a moment of time as an instant. We can test if an object is an instant by using `is.instant()`. For example,

```
R> is.instant(364)

[1] FALSE

R> is.instant(start_2012)

[1] TRUE
```

We can also capture the current time as an instant with `now()`, and the current day with `today()`.

5.2. Intervals

Intervals, durations, and periods are all ways of recording timespans. Of these, intervals are the most simple. An interval is a span of time that occurs between two specific instants. The length of an interval is never ambiguous, because we know when it occurs. Moreover, we can calculate the exact length of any unit of time that occurs during it.**lubridate** introduces the `interval` object class for modelling intervals.

We can create `interval` objects by subtracting two instants or by using the command `new_interval()`.

```
R> start_2011 <- ymd_hms("2011-01-01 12:00:00")
R> start_2010 <- ymd_hms("2010-01-01 12:00:00")
R> span <- start_2011 - start_2010

[1] 2010-01-01 12:00:00 -- 2011-01-01 12:00:00
```

We can access the start and end dates of an `interval` object with `int_start()` and `int_end()`. Intervals always begin at the date-time that occurs first and end at the date-time that occurs last. Hence, intervals always have a positive length.

```
R> int_start(span)

[1] "2010-01-01 12:00:00 UTC"

R> int_end(span)

[1] "2011-01-01 12:00:00 UTC"
```

Unfortunately, since intervals are anchored to their start and end dates, they are not very useful for date-time calculations. It only makes sense to add an interval to its start date or to subtract it from its end date.

```
R> start_2010 + span

[1] "2011-01-01 12:00:00 UTC"
```

Adding intervals to other date-times will not produce an error message. Instead **lubridate** will coerce the interval to a `duration` object, which is like an interval but without the reference dates. See Section 5.3.

```
R> start_2011 + span

coercing interval to duration
[1] "2012-01-01 12:00:00 UTC"
```

In most cases this will return the intended result, but accuracy can be ensured by first explicitly converting the interval to either a duration or a period, as described in the next two sections.

We can convert any other type of timespan to an interval by pairing it to a start date with `as.interval()`. For example:

```
R> as.interval(difftime(start_2011, start_2010), ymd("2010-03-05"))

[1] 2010-03-05 -- 2011-03-05
```

5.3. Durations

If we remove the start and end dates from an interval, we will have a generic time span that we can add to any date. But how should we measure this length of time? If we record the time span in seconds, it will have an exact length since seconds always have the same length. We call such time spans *durations*. Alternatively, we can record the time span in larger units, such as minutes or years. Since the length of these units varies over time, the exact length of the time span will depend on when it begins. These non-exact time spans are called *periods* and will be discussed in the next section.

The length of a duration is invariant to leap years, leap seconds, and daylight savings time because durations are measured in seconds. Hence, durations have consistent lengths and can be easily compared to other durations. Durations are the appropriate object to use when comparing time based attributes, such as speeds, rates, and lifetimes. `difftime` objects from base R are one type of duration object. **lubridate** provides a second type: `duration` class objects. These objects can be used with other date-time objects without worrying about what units they are displayed in. A `duration` object can be created with the function `new_duration()`:

```
R> new_duration(60)

[1] 60s
```

For large durations, it becomes inconvenient to describe the length in seconds. For example, not many people would recognize 31557600 seconds as the length of a standard year. For this reason, large `duration` objects are followed in parentheses by an estimated length. Estimated units are created using the following relationships. A minute is 60 seconds, an hour 3600 seconds, a day 86400, a week 604800, and a year 31557600 (365.25 days). Month units are

not used because they are so variable. The estimates are only provided for convenience; the underlying object is always recorded as a fixed number of seconds.

`duration` objects can be easily created with the helper functions `dyears()`, `dweeks()`, `ddays()`, `dhours()`, `dminutes()`, and `dseconds()`. The d in the title stands for duration and distinguishes these objects from `period` objects, discussed in Section 5.4. Each object creates a duration in seconds using the estimated relationships given above. The argument of each function is the number of estimated units we wish to include in the duration. For example,

```
R> dminutes(1)

[1] 60s

R> dseconds(60)

[1] 60s

R> dminutes(2)

[1] 120s

R> 1:3 * dhours(1)

[1]  3600s (1h)   7200s (2h)  10800s (3h)
```

Durations can be added and subtracted to any instant object. For example,

```
R> start_2011 + dyears(1)

[1] "2012-01-01 12:00:00 UTC"

R> start_2012 <- ymd_hms("2012-01-01 12:00:00")
R> start_2012 + dyears(1)

[1] "2012-12-31 12:00:00 UTC"
```

Durations can also be added to or subtracted from intervals and other durations. For example,

```
R> dweeks(1) + ddays(6) + dhours(2) + dminutes(1.5) + dseconds(3)

[1] 1130493s (13.08d)
```

We can also create durations from `interval` and `period` objects using `as.duration()`.

```
R> as.duration(span)

[1] 31536000s (365d)
```

5.4. Periods

Periods record a time span in units larger than seconds, such as years, months, weeks, days, hours, and minutes. For convenience, we can also create a period that only uses seconds, but such a period would have the same properties as a duration. **lubridate** introduces the `period` class to model periods. We construct `period` objects with the helper functions `years()`, `months()`, `weeks()`, `days()`, `hours()`, `minutes()`, and `seconds()`.

```
R> months(3)

[1] 3 months

R> months(3) + days(2)

[1] 3 months and 2 days
```

These functions do not contain a "d" in their name, because they do not create durations; they no longer have consistent lengths (as measured in seconds). For example, `months(2)` always has the length of two months even though the length of two months will change depending on when the period begins. For this reason, we can not compute exactly how long a period will be in seconds until we know when it occurs. However, we can still perform date-time calculations with periods. When we add or subtract a period to an instant, the period becomes anchored to the instant. The instant tells us when the period occurs, which allows us to calculate its exact length in seconds.

As a result, we can use periods to accurately model clock times without knowing when events such as leap seconds, leap days, and DST changes occur.

```
R> start_2012 + years(1)

[1] "2013-01-01 12:00:00 UTC"
```

vs.

```
R> start_2012 + dyears(1)

[1] "2012-12-31 12:00:00 UTC"
```

We can also convert other timespans to `period` objects with the function `as.period()`.

```
R> as.period(span)

[1] 1 year
```

Periods can be added to instants, intervals, and other periods, but not to durations.

5.5. Division with timespans

We often wish to answer questions that involve dividing one timespan by another. For example, "how many weeks are there between Halloween and Christmas?" or "how old is a

person born on September 1, 1976?" Objects of each timespan class—`interval`, `duration`, and `period`—can be divided by objects of the others. The results of these divisions varies depending on the nature of the timespans involved. Modulo operators (i.e, `%%` and `%/%`) also work between timespan classes.

To illustrate this, we make an interval that lasts from Halloween until Christmas.

```
R> halloween <- ymd("2010-10-31")
R> christmas <- ymd("2010-12-25")
R> interval <- new_interval(halloween, christmas)

[1] 2010-10-31 -- 2010-12-25
```

Since durations are an exact measurement of a timespan, we can divide this interval by a duration to get an exact answer.

```
R> interval / dweeks(1)

[1] 7.857143
```

Intervals are also exact measures of timespans. Although it is more work, we could divide an interval by another interval to get an exact answer. This gives the same answer as above because **lubridate** automatically coerces `interval` objects in the denominator to `duration` objects.

```
R> interval / new_interval(halloween, halloween + weeks(1))

interval denominator coerced to duration
[1] 7.857143
```

Division is not possible with periods. Since periods have inconsistent lengths, we can not express the remainder as a decimal. For example, if we were to divide our interval by months, the remainder would be 24 days.

```
R> interval \%\% months(1)

[1] 24 days
```

Should we calculate 24 days as $24/30 = 0.8$ months since November has 30 days? Or should we calculate 24 days as $24/31 = 0.77$ months since December has 31 days? Both November and December are included in our numerator. If we want an exact calculation, we should use a duration instead of a period.

If we attempt to divide by a `period` object, **lubridate** will return a warning message and automatically perform integer division: it returns the integer value equal to the number of whole periods that occur in the timespan. This is equivalent to the `%/%` operator. As shown above, we can retrieve the remainder using the `%%` operator.

```
R> interval / months(1)
```

	Instant	Interval	Duration	Period
Instant	N/A	instant	instant	instant
Interval	instant	interval*	interval	interval
Duration	instant	interval	duration	period
Period	instant	interval	period	period

Table 6: Adding two date-time objects will create the above type of object. (∗ = A duration if the intervals do not align.)

```
estimate only: convert periods to intervals for accuracy
[1] 1

R> interval %/% months(1)

[1] 1
```

In summary, arithmetic with date-times involves four types of objects: instants, intervals, durations, and periods. **lubridate** creates new object classes for intervals, durations, and periods. It recognizes that most common date-time classes, such as `POSIXt` and `Date`, refer to instants. Table 6 describes which objects can be added to each other and what type of object will result.

6. Rounding dates

Like numbers, date-times occur in order. This allows date-times to be rounded. **lubridate** provides three methods that help perform this rounding: `round_date()`, `floor_date()`, and `ceiling_date()`. The first argument of each function is the date-time or date-times to be rounded. The second argument is the unit to round to. For example, we could round April 20, 2010 to the nearest day:

```
R> april20 <- ymd_hms("2010-04-20 11:33:29")
R> round_date(april20, "day")

[1] "2010-04-20 UTC"
```

or the nearest month.

```
R> round_date(april20, "month")

[1] "2010-05-01 UTC"
```

Notice that rounding a date-time to a unit sets the date to the start of that unit (e.g, `round_date(april20, "day")` sets the hours, minutes, and seconds information to 00).

`ceiling_date()` provides a second simple way to find the last day of a month. Ceiling the date to the next month and then subtract a day.

```
R> ceiling_date(april20, "month") - days(1)

[1] "2010-04-30 UTC"
```

7. Time zones

Time zones give multiple names to the same instant. For example, "2010-03-26 11:53:24 CDT" and "2010-03-26 12:53:24 EDT" both describe the same instant. The first shows how the instant is labeled in the United States' central time zone (CDT). The second shows how the same instant is labelled in the United States' eastern time zone (EDT). Time zones complicate date-time data but are useful for mapping clock time to local daylight conditions. When working with instants, it is standard to give the clock time as it appears in the Coordinated Universal time zone (UTC). This saves calculations but can be annoying if your computer insists on translating times to your current time zone. It may also be inconvenient to discuss clock times that occur in a place unrelated to the data.

lubridate eases the frustration caused by time zones in two ways. We can change the time zone in which an instant is displayed by using the function `with_tz()`. This changes how the clock time is displayed, but not the specific instant of time that is referred to. For example,

```
R> date

[1] "2010-01-01 09:51:48 CST"

R> with_tz(date, "UTC")

[1] "2010-01-01 15:51:48 UTC"
```

`force_tz()` does the opposite of `with_tz()`: it changes the actual instant of time saved in the object, while keeping the displayed clock time the same. The new time zone value alerts us to this change. For example, the code below moves us to a new instant that occurs 6 hours earlier.

```
R> date

[1] "2010-01-01 09:51:48 CST"

R> force_tz(date, "UTC")

[1] "2010-01-01 09:51:48 UTC"
```

`with_tz()` and `force_tz()` only work with time zones recognized by your computer's operating system. This list of time zones will vary between computers. See the base R help page for `Sys.timezone()` for more information.

8. Daylight savings time

In many parts of the world, the official clock time springs forward by one hour in the spring and falls back one hour in the fall. For example, in Chicago, Illinois a change in daylight savings time occurred at 2:00 AM on March 14, 2010. The last instant to occur before this change was 2010-03-14 01:59:59 CST.

```
R> dst_time <- ymd_hms("2010-03-14 01:59:59")
R> dst_time <- force_tz(dst_time, "America/Chicago")

[1] "2010-03-14 01:59:59 CST"
```

One second later, Chicago clock times read

```
R> dst_time + dseconds(1)

[1] "2010-03-14 03:00:00 CDT"
```

It seems that we gained an extra hour during that second, which is how daylight savings time works. We can completely avoid the changes in clock time caused by daylight savings times by working with periods instead of durations. For example,

```
R> dst_time + hours(2)

[1] "2010-03-14 03:59:59 CDT"
```

displays the clock time that usually occurs two hours after 1:59:59 AM. When using periods, we do not have to track DST changes because they will not affect our calculations. This will prevent errors when we are trying to model events that depend on clock times, such as the opening and closing bells of the New York Stock Exchange. Adding a duration would give us the actual clock time that appeared exactly two hours later on March 14, 2010. This prevents changes in clock time from interfering with exact measurements, such as the life times of a light bulb, but will create surprises if clock times are important to our analysis.

```
R> dst_time + dhours(2)

[1] "2010-03-14 04:59:59 CDT"
```

If we ever inadvertently try to create an instant that occurs in the one hour "gap" between 2010-03-14 01:59:59 CST and 2010-03-14 03:00:00 CDT, **lubridate** will return `NA` since such times are not available.

We can also avoid the complications created by daylight savings time by keeping our date-times in a time zone such as "UTC", which does not adopt daylight savings hours.

9. Case study 1

The next two sections will work through some techniques using **lubridate**. First, we will use **lubridate** to calculate the dates of holidays. Then we will use **lubridate** to explore an example data set (`lakers`).

9.1. Thanksgiving

Some holidays, such as Thanksgiving (US) and Memorial Day (US) do not occur on fixed dates. Instead, they are celebrated according to a common rule. For example, Thanksgiving is celebrated on the fourth Thursday of November. To calculate when Thanksgiving will occur in 2010, we can start with the first day of 2010.

```
R> date <- ymd("2010-01-01")

[1] "2010-01-01 UTC"
```

We can then add 10 months to our date, or directly set the date to November.

```
R> month(date) <- 11

[1] "2010-11-01 UTC"
```

We check which day of the week November 1st is.

```
R> wday(date, label = TRUE, abbr = FALSE)

[1] Monday
```

This implies November 4th will be the first Thursday of November.

```
R> date <- date + days(3)

[1] "2010-11-04 UTC"

R> wday(date, label = TRUE, abbr = FALSE)

[1] Thursday
```

Next, we add three weeks to get to the fourth Thursday in November, which will be Thanksgiving.

```
R> date + weeks(3)

[1] "2010-11-25 UTC"
```

9.2. Memorial Day

Memorial day also occurs according to a rule; it falls on the last Monday of May. To calculate the date of Memorial day, we can again start with the first of the year.

```
R> date <- ymd("2010-01-01")
```

```
[1] "2010-01-01 UTC"
```

Next, we set the month to May.

```
R> month(date) <- 5
```

```
[1] "2010-05-01 UTC"
```

Now our holiday occurs in relation to the last day of the month instead of the first. We find the last day of the month by rounding up to the next month and then subtracting a day.

```
R> date <- ceiling_date(date, "month") - days(1)
```

```
[1] "2010-05-31 UTC"
```

We can then check which day of the week May 31st falls on. It happens to be a Monday, so we are done. If May 31st had been another day of the week, we could've subtracted an appropriate number of days to get to the last Monday of May.

```
R> wday(date, label = TRUE, abbr = FALSE)
```

```
[1] Monday
```

10. Case study 2

The `lakers` data set contains play by play statistics of every major league basketball game played by the Los Angeles Lakers during the 2008-2009 season. This data is from `http://www.basketballgeek.com/downloads/` (Parker 2010) and comes with the **lubridate** package. We will explore the distribution of Lakers' games throughout the year as well as the distribution of plays within Lakers' games. We choose to use the **ggplot2** (Wickham 2009) package to create our graphs.

The lakers data set comes with a `date` variable which records the date of each game. Using the `str()` command, we see that R recognizes the dates as integers.

```
R> str(lakers$date)
```

```
int [1:34624] 20081028 20081028 20081028 ...
```

Before we can work with them as dates, we must parse them into R as date-time objects. The dates appear to be arranged with their year element first, followed by the month element, and then the day element. Hence, we should use the `ymd()` parsing function.

```
R> lakers$date <- ymd(lakers$date)
R> str(lakers$date)
```

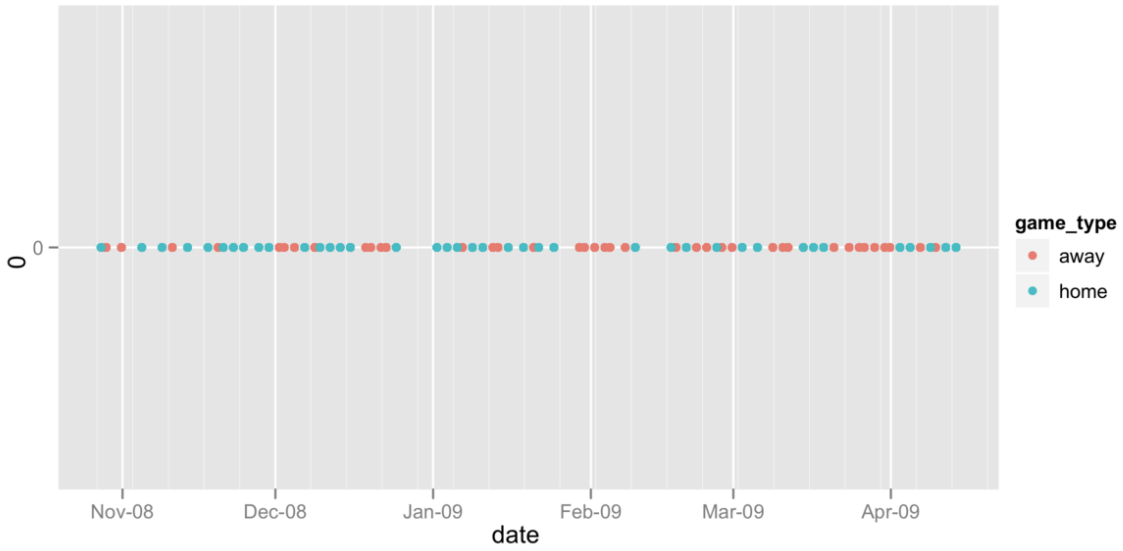

Figure 1: Home games and away games appear to occur in clusters.

```
POSIXct[1:34624], format: "2008-10-28" "2008-10-28" ...
```

R now recognizes the dates as `POSIXct` date-time objects. It will now treat them as date-times in any functions that have `POSIXct` specific methods. For example, if we plot the occurrences of home and away games throughout the season, our x axis will display date-time information for the tick marks (Figure 1).

```
R> qplot(date, 0, data = lakers, colour = game_type)
```

Figure 1 shows that games are played continuously throughout the season with a few short breaks. The frequency of games seems lower at the start of the season and games appear to be grouped into clusters of home games and away games. The tick marks and breaks on the x axis are automatically generated by the **lubridate** method `pretty.dates()`.

Next we will examine how Lakers games are distributed throughout the week. We use the `wday()` command to extract the day of the week of each date.

```
R> qplot(wday(date, label = TRUE, abbr = FALSE), data = lakers,
    geom = "histogram")
```

The frequency of basketball games varies throughout the week (Figure 2). Surprisingly, the highest number of games are played on Tuesdays.

Now we look at the games themselves. In particular, we look at the distribution of plays throughout the game. The `lakers` data set lists the time that appeared on the game clock for each play. These times begin at 12:00 at the beginning of each period and then count down to 00:00, which marks the end of the period. The first two digits refer to the number of minutes left in the period. The second two digits refer to the number of seconds.

The times have not been parsed as date-time data to R. It would be difficult to record the time data as a date-time object because the data is incomplete: a minutes and seconds element

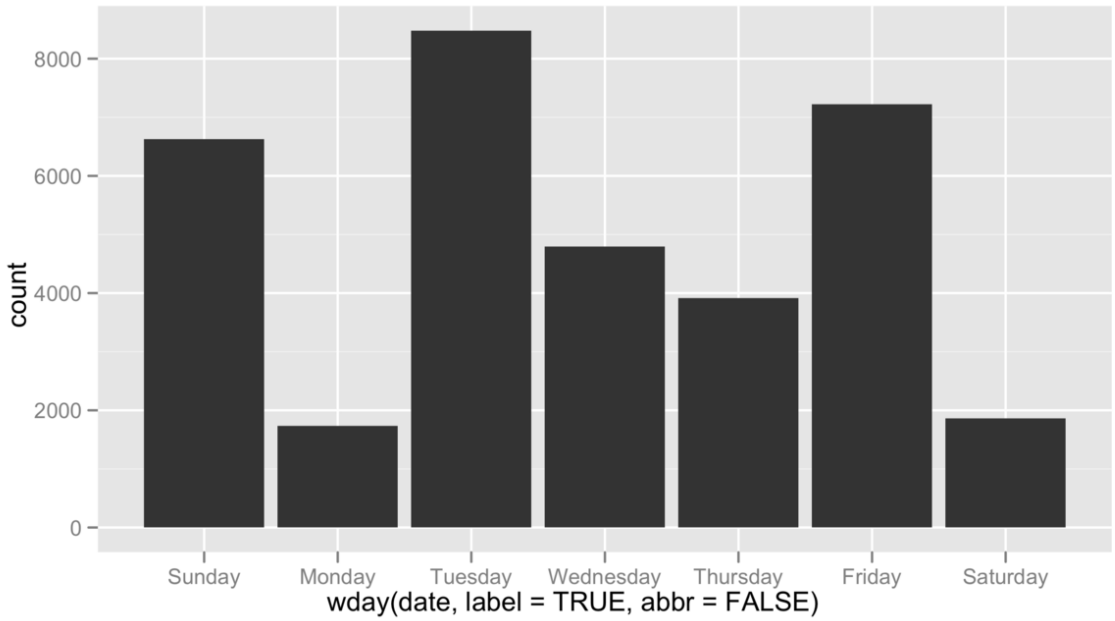

Figure 2: More games occur on Tuesdays than any other day of the week.

are not sufficient to identify a unique instant of time. However, we can store the minutes and seconds information as a `period` object, as defined in Section 5.4, using the `ms()` parse function.

```
R> lakers$time <- ms(lakers$time)
```

Since periods have relative lengths, it is dangerous to compare them to each other. So we should next convert our periods to *durations*, which have exact lengths.

```
R> lakers$time <- as.duration(lakers$time)
```

This allows us to directly compare different durations. It would also allow us to determine exactly when each play occurred by adding the duration to the *instant* the game began. (Unfortunately, the starting time for each game is not available in the data set). However, we can still calculate when in each game each play occurred. Each period of play is 12 minutes long and overtime—the 5th period—is 5 minutes long. At the start of each period, the game clock begins counting down from 12:00. So to calculate how much play time elapses before each play, we subtract the time that appears on the game clock from a duration of 12, 24, 36, 48, or 53 minutes (depending on the period of play). We now have a new duration that shows exactly how far into the game each play occurred.

```
R> lakers$time <- dminutes(c(12, 24, 36, 48, 53)[lakers$period]) -
+    lakers$time
```

We can now plot the number of events over time within each game (Figure 3). We can plot the time of each event as a duration, which will display the number of seconds into the game each play occurred on the x axis,

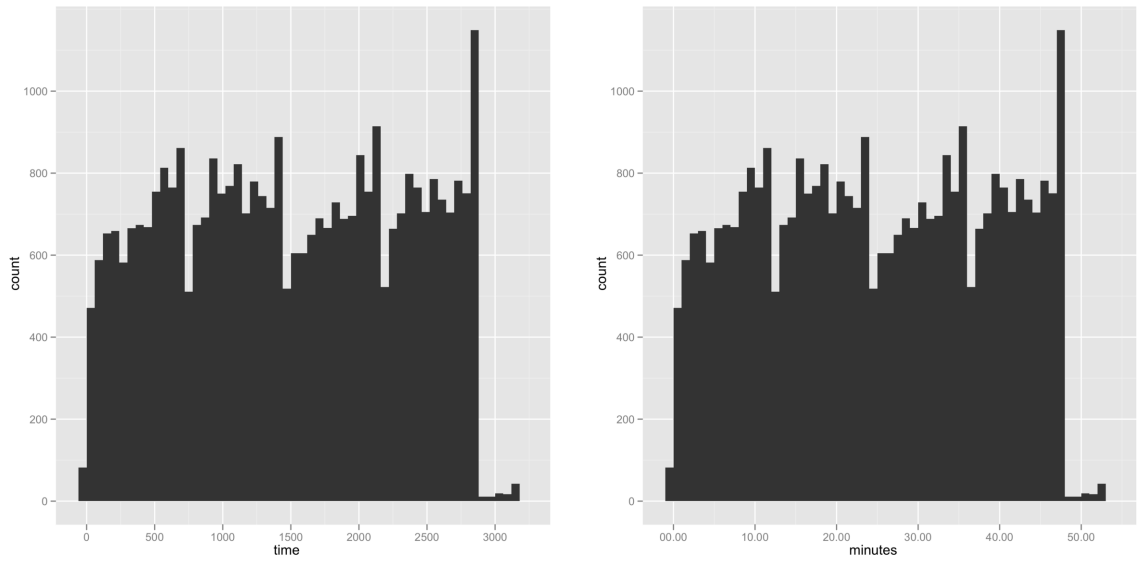

Figure 3: The graph on the left displays seconds on the x axis. The graph on the right uses a more intuitive division with minutes.

```
R> qplot(time, data = lakers, geom = "histogram", binwidth = 60)
```

or we can take advantage of `pretty.date()` to make pretty tick marks by first transforming each duration into a date-time. This helper function recognizes the most intuitive binning and labeling of date-time data, which further enhances our graph. To change durations into date-times we can just add them all to the same date-time. It does not matter which date we chose. Since the range of our data occurs entirely within an hour, only the minutes information will display in the graph.

```
R> lakers$minutes <- ymd("2008-01-01") + lakers$time
R> qplot(minutes, data = lakers, geom = "histogram", binwidth = 60)
```

We see that the number of plays peaks within each of the four periods and then plummets at the beginning of the next period, Figure 3. The most plays occur in the last minute of the game. Perhaps any shot is worth taking at this point or there's less of an incentive not to foul other players. Fewer plays occur in overtime, since not all games go to overtime.

Now lets look more closely at just one basketball game: the game played against the Boston Celtics on Christmas of 2008. We can quickly model the amounts of time that occurred between each shot attempt.

```
R> game1 <- lakers[lakers$date == ymd("20081225"),]
R> attempts <- game1[game1$etype == "shot",]
```

The waiting times between shots will be the timespan that occurs between each shot attempt. Since we have recorded the time of each shot attempt as a duration (above), we can record the differences by subtracting the two durations. This automatically creates a new duration whose length is equal to the difference between the first two durations.

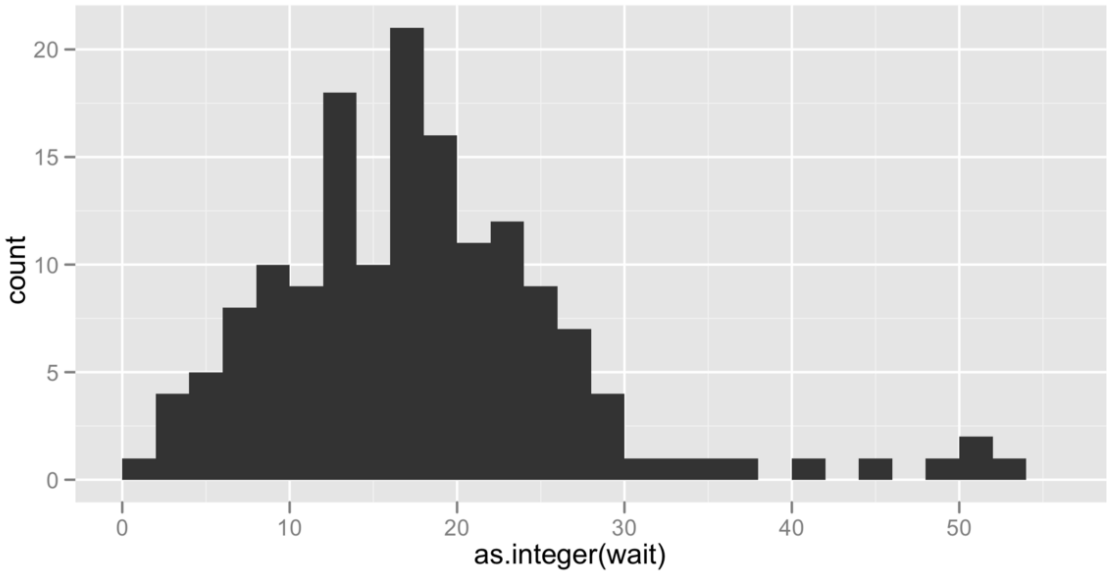

Figure 4: Wait times between shot attempts rarely lasted more than 30 seconds.

```
R> attempts$wait <-  c(attempts$time[1], diff(attempts$time))
R> qplot(as.integer(wait), data = attempts, geom = "histogram", binwidth = 2)
```

Rarely did 30 seconds go by without at least one shot attempt, but on occasion up to 50 seconds would pass without an attempt.

We can also examine changes in the score throughout the game. This reveals that though the game was eventful, the Lakers maintained a lead for the most of the game, Figure 5. Note: the necessary calculations are made simpler by the `ddply()` function from the **plyr** package, which **lubridate** automatically loads.

```
R> game1_scores <- ddply(game1, "team", transform, score = cumsum(points))
R> game1_scores <- game1_scores[game1_scores$team != "OFF",]
R> qplot(ymd("2008-01-01") + time, score, data = game1_scores,
    geom = "line", colour = team)
```

11. Conclusion

Date-times create technical difficulties that other types of data do not. They must be specifically identified as date-time data, which can be difficult due to the overabundance of date-time classes. It can also be difficult to access and manipulate the individual pieces of data contained within a date-time. Arithmetic with date-times is often appropriate, but must follow different rules than arithmetic with ordinal numbers. Finally, date-related conventions such as daylight savings time and time zones make it difficult to compare and recognize different moments of time.

Base R handles many of these difficulties, but not all. Moreover, base R's date-time methods can be complicated and confusing. **lubridate** makes it to easier to work with date-time data

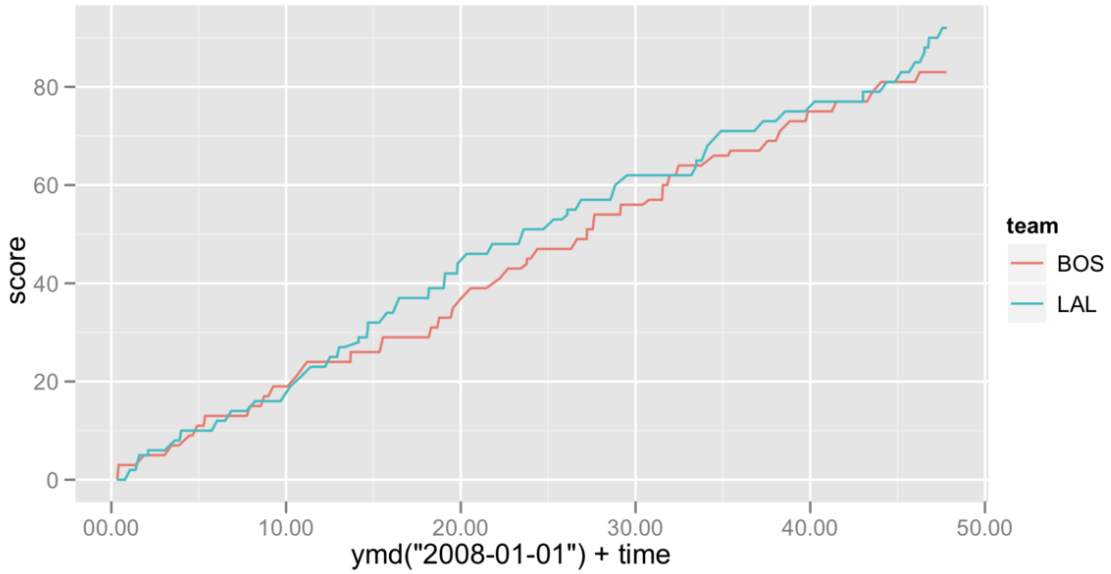

Figure 5: The lead changed between the Lakers and Celtics numerous times during the game.

in R. The package provides a set of standard methods for most common date-time classes. These methods make it simple to parse, manipulate, and perform calculations with date-time objects. By implementing the time concepts pioneered by projects such as **Joda Time** and **Boost.Date_Time**, **lubridate** helps researchers perform precise calculations as well as model tricky time-related processes. **lubridate** also makes it simple to switch between time zones and to use or ignore daylight savings time.

Future work on the **lubridate** package will develop methods for handling partial dates and for modeling recurrent events, such as stock market opening times, business days, or street cleaning hours. In particular, we hope to create methods for R that work with reoccurring temporal date patterns, which were introduced by Fowler (1997) and have been implemented in Ruby by the **runt** project (Lipper 2008).

Acknowledgments

We would like to thank the National Science Foundation. This work was supported by the NSF VIGRE Grant, number DMS-0739420.

References

Armstrong W (2009). ***fts**: R Interface to **tslib** (A Time Series Library in C++)*. R package version 0.7.6, URL http://CRAN.R-project.org/package=fts.

Colebourne S, O'Neill BS (2010). "**Joda-Time** – Java Date and Time API." Release 1.6.2, URL http://joda-time.sourceforge.net/.

Fowler M (1997). "Recurring Events for Calendars." URL http://martinfowler.com/apsupp/recurring.pdf.

Garland J (2011). "**Boost.Date_Time** – C++ library." Release 1.46.1, URL http://www.boost.org/.

Grothendieck G, Petzoldt T (2004). "Date and Time Classes in R." *R News*, **4**(1), 32.

Hallman J (2010). ***tis**: Time Indexes and Time Indexed Series*. R package version 1.12, URL http://CRAN.R-project.org/package=tis.

James D, Hornik K (2010). ***chron**: Chronological Objects which Can Handle Dates and Times*. R package version 2.3-35., URL http://CRAN.R-project.org/package=chron.

Lipper M (2008). "**runt** – Ruby Temporal Expressions." Release 0.7.0, URL http://runt.rubyforge.org/.

Parker RJ (2010). "Basketball Geek: Advancing Our Understanding of the Game of Basketball." URL http://www.basketballgeek.com/data/.

Portfolio and Risk Advisory Group, Commerzbank Securities (2009). ***its**: Irregular Time Series*. R package version 1.1.8, URL http://CRAN.R-project.org/package=its.

R Development Core Team (2011). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org/.

Ryan JA, Ulrich JM (2010). ***xts**: Extensible Time Series*. R package version 0.7-1, URL http://CRAN.R-project.org/package=xts.

Trapletti A, Hornik K (2009). ***tseries**: Time Series Analysis and Computational Finance*. R package version 0.10-22., URL http://CRAN.R-project.org/package=tseries.

Wickham H (2009). ***ggplot2**: Elegant Graphics for Data Analysis*. Springer-Verlag, New York.

Wuertz D, Chalabi Y (2010a). ***timeDate**: Rmetrics – Chronological and Calendarical Objects*. R package version 2110.88, URL http://CRAN.R-project.org/package=timeDate.

Wuertz D, Chalabi Y (2010b). ***timeSeries**: Rmetrics - Financial Time Series Objects*. R package version 2110.87, URL http://CRAN.R-project.org/package=timeSeries.

Zeileis A, Grothendieck G (2005). "**zoo**: S3 Infrastructure for Regular and Irregular Time Series." *Journal of Statistical Software*, **14**(6), 1–27. URL http://www.jstatsoft.org/v14/i06/.

Affiliation:

Garrett Grolemund
Rice University
Houston, TX 77251-1892, United States of America
E-mail: grolemund@rice.edu

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 40, Issue 3 *Submitted:* 2010-07-23
April 2011 *Accepted:* 2010-11-29