nanotime: A datetime library with nanosecond precision

Dirk Eddelbuettel[1] **and Leonardo Silvestri**[2]

[1]Department of Statistics, University of Illinois, Urbana-Champaign, IL, USA; [2]Baltimore, MD, USA

This version was compiled on April 29, 2020

The nanotime package (Eddelbuettel and Silvestri, 2020) provides a coherent set of temporal types and functions with nanosecond precision. The types are: point in time, interval (which may have an open or closed start/end), period (a human representation of time, such as day, month, etc.), and duration. Of particular interest are the set and arithmetic operations defined on these types as well as the fact that all functionality is designed to correctly handle instances across different time zones. Because these temporal types are based on R built-in types, most functions have an efficient implementation and the types are suitable for use in `data.frame` and `data.table`. `nanotime` is also a better choice than the native `POSIXct` in most cases where fractional seconds are needed.

Implementation of the temporal types

All new types are implemented using one or more signed 64-bit integers. This avoids the floating point issues associated with `POSIXct` and allows for exact representation of nanosecond datetimes in the approximative range of years 1386 to 2554, which is large enough for most applications. It also allows for exact arithmetic and comparison.

Time zones

Time zones are only needed in two cases. First, for representation, because of course the same time is represented differently in different time zones, and secondly, for operations with calendar time, since a year or a day has a different length depending on its location in time and depending in which time zone it is considered.

To avoid all ambiguity, a time zone is never associated as an attribute with any of the temporal types defined in the **nanotime** package. Any operation where a time zone is needed in order to make sense of it requires an explicit specification.

Temporal types

The **nanotime** package implements four temporal types: time (`nanotime`), interval (`nanoival`), duration (`nanoduration`) and period (`nanoperiod`). This set of types seems to have become the consensus for various implementations. It is the set chosen by *Joda Time* (see Jod, 2019), the **lubridate** package (Grolemund and Wickham, 2011) (without nanosecond resolution), and a similar set (minus the interval type) is chosen by the latest Java Date and Time implementation (see Evans and Warburton, 2014).

nanotime. `nanotime` is a point in time with nanosecond resolution. It is implemented as an S4 class containing the type `integer64` from the *bit64* package (see Oehlschlägel, 2015) and represents an offset in nanoseconds from 1970-01-01 UTC. It does not have an associated time zone, but can be displayed in any desired time zone with the `format` or `print` functions. Finally, it uses the POSIX definition of time, usually referred to as *POSIX time* or *UNIX time* (see The Open Group, 2016). This is a representation suitable for many purposes.

It can be constructed either from an `integer64` or from a `character`:

```
as.nanotime(as.integer64("1580274000000000000"))
#  [1] 2020-01-29T05:00:00+00:00
as.nanotime("2020-01-29 13:12:00.000000001 America/New_York")
#  [1] 2020-01-29T18:12:00.000000001+00:00
as.nanotime("2020-01-29 13:12:00.000000001-05:00")
#  [1] 2020-01-29T18:12:00.000000001+00:00
```

Short forms are also allowed:

```
as.nanotime("2020-01-29 Europe/London")
#  [1] 2020-01-29T00:00:00+00:00
as.nanotime("2020-01-29 12:01:01 Africa/Lagos")
#  [1] 2020-01-29T11:01:01+00:00
as.nanotime("2020-01-29 12:01:01.001 America/Tegucigalpa")
#  [1] 2020-01-29T18:01:01.001+00:00
```

nanoival. A `nanoival` is an interval in time defined by two `nanotime`, the first one being the start time and the second one the end time, and by two `logical` that determine if the interval start and end are open (`TRUE`) or closed (`FALSE`), i.e. if the start and end of the interval are excluded (`TRUE`) or included (`FALSE`) in the interval.

A `nanoival` can either be constructed with two `nanotime` and two `logical` or it can be constructed from a `character` string. The string format uses the '-' and '+' signs at the beginning and end to indicate if the interval start and end are open or closed. By default an interval has a closed start and an open end.

```
as.nanoival("-2009-01-01 13:12:00 America/New_York -> 2009-02-01 15:11:03 America/New_York+")
#  [1] -2009-01-01T18:12:00+00:00 -> 2009-02-01T20:11:03+00:00+

start <- nanotime("2009-01-01 13:12:00 America/New_York")
end   <- nanotime("2009-02-01 15:11:00 America/New_York")

nanoival(start, end)                    # by default sopen=F,eopen=T
#  [1] +2009-01-01T18:12:00+00:00 -> 2009-02-01T20:11:00+00:00-
nanoival(start, end, sopen=FALSE, eopen=TRUE)
#  [1] +2009-01-01T18:12:00+00:00 -> 2009-02-01T20:11:00+00:00-
```

Accessors for `nanoival` are provided:

```
ival  <- as.nanoival("-2009-01-01 UTC -> 2009-02-01 UTC+")
nanoival.start(ival)
#  [1] 2009-01-01T00:00:00+00:00
nanoival.end(ival)
#  [1] 2009-02-01T00:00:00+00:00
nanoival.sopen(ival)
#  [1] TRUE
nanoival.eopen(ival)
#  [1] FALSE
```

nanoduration. A `nanoduration` is simply a count of nanoseconds, which may be negative.

```
nanoduration(hours=1, minutes=1, seconds=1, nanoseconds=1)
#  [1] 01:01:01.000_000_001
as.nanoduration("00:00:01")
#  [1] 00:00:01
as.nanoduration("-00:00:01")
#  [1] -00:00:01
as.nanoduration("100:00:00")
#  [1] 100:00:00
as.nanoduration("00:00:00.000_000_001")
#  [1] 00:00:00.000_000_001
```

nanoperiod. A `nanoperiod` represents the calendar or "business" view of a duration with the concepts of month and day. The exact duration of a period is unknown until it is anchored to a point in time and associated with a time zone, first because a month has variable length and secondly because it might span a daylight saving time change.

`nanoperiod` is composed of two parts: a months/days part and a duration. Note that these components may have opposite signs.

For convenience, the constructor syntax allows specifying years and weeks, but they are converted to their representation in months/days.

```
as.nanoperiod("1y1m1w1d/01:01:01.000_000_001")
#  [1] 13m8d/01:01:01.000_000_001
nanoperiod(months=13, days=-1, duration="01:00:00")
#  [1] 13m-1d/01:00:00
```

Accessors for `nanoperiod` components are provided:

```
ones <- as.nanoperiod("1y1m1w1d/01:01:01.000_000_001")
nanoperiod.month(ones)
#  [1] 13
nanoperiod.day(ones)
#  [1] 8
```

```
nanoperiod.nanoduration(ones)
#  [1] 01:01:01.000_000_001
```

Set operations

The set operations `intersect`, `union` and `setdiff` are provided for a more rigorous handling of temporal types. Additionally, `intersect` and `setiff` have counterpart functions `intersect.idx` and `setdiff.idx` that, instead of computing a new set, return the index of the set. Finally, the operator `%in%` is overloaded so as to provide a convenient intersection shorthand particularly suitable for the subsetting of `data.table` time-series.

Here are some examples:

```
ni1 <- c(as.nanoival("+2013-01-01+00:00          -> 2014-01-01+00:00-"),
         as.nanoival("+2015-01-01T12:00:01+00:00 -> 2016-01-01+00:00-"),
         as.nanoival("+2017-01-01+00:00          -> 2018-01-01+00:00-"))
ni2 <-   as.nanoival("-2013-02-02+00:00          -> 2015-06-10+00:00+")
intersect(ni1, ni2)
#  [1] -2013-02-02T00:00:00+00:00 -> 2014-01-01T00:00:00+00:00- +2015-01-01T12:00:01+00:00 -> 2015-06-10T00:00:00+00:0
union(ni1, ni2)
#  [1] +2013-01-01T00:00:00+00:00 -> 2016-01-01T00:00:00+00:00- +2017-01-01T00:00:00+00:00 -> 2018-01-01T00:00:00+00:0
setdiff(ni1, ni2)
#  [1] +2013-01-01T00:00:00+00:00 -> 2013-02-02T00:00:00+00:00+ -2015-06-10T00:00:00+00:00 -> 2016-01-01T00:00:00+00:0
```

Functions and operations

Arith. The usual expected arithmetic operations are defined for various types when these make sense. In particular one can add/subtract a `period` or a `duration` to/from a `nanotime` or a `nanoival` and multiply and divide `period` and `duration` by a scalar.

```
as.nanotime("2020-03-07 01:03:28 America/Los_Angeles") + 999
#  [1] 2020-03-07T09:03:28.000000999+00:00
as.nanotime("2020-03-07 12:03:28+00:00") + as.nanoduration("24:00:00")
#  [1] 2020-03-08T12:03:28+00:00
## daylight saving time transition:
plus(as.nanotime("2020-03-07 12:03:28+00:00"), as.nanoperiod("1d"), "America/Los_Angeles")
#  [1] 2020-03-08T11:03:28+00:00

as.nanoduration("24:00:00")/3
#  [1] 08:00:00
-as.nanoduration("24:00:00")
#  [1] -24:00:00
```

Compare. Compare operations are mostly straightforward except maybe for `nanoival` which is ordered by its start `nanotime`. If both starts are equal, a closed start comes before an open start. If both `sopen` are the same, then the comparison happens on the end of the `nanotime`, with a shorter interval coming before a longer one. `nanoperiod` do not have a meaningful ordering and therefore remain unordered.

```
as.nanoival("+2020-04-03 00:12:00 UTC -> 2020-04-04 00:12:00 UTC-") <
    as.nanoival("-2020-04-03 00:12:00 UTC -> 2020-04-04 00:12:00 UTC-")
#  [1] TRUE
nanotime(1) <= nanotime(2)
#  [1] TRUE
as.nanoduration(1) > as.nanoduration(2)
#  [1] FALSE
```

Sequence generation. Sequence generation is provided for `nanotime` and `nanoival`. The increment can either be a `nanoduration` or a `nanoperiod`. Since a period is sensitive to the time zone in which the operation takes place, the additional `tz` argument must be provided to `seq`.

```
seq(nanotime("2020-03-28+00:00"), by=as.nanoduration("24:00:00"), length.out=3)
#  [1] 2020-03-28T00:00:00+00:00 2020-03-29T00:00:00+00:00 2020-03-30T00:00:00+00:00
seq(nanotime("2020-03-28+00:00"), by=as.nanoperiod("1d"), length.out=3, tz="Europe/London")
#  [1] 2020-03-28T00:00:00+00:00 2020-03-29T00:00:00+00:00 2020-03-29T23:00:00+00:00
```

```
ival <- as.nanoival("+2020-03-28T13:00:00+00:00 -> 2020-03-28T15:00:00+00:00-")
print(seq(ival, by=as.nanoperiod("1m"), length.out=3, tz="Europe/London"), tz="Europe/London")
#  [1] +2020-03-28T13:00:00+00:00 -> 2020-03-28T15:00:00+00:00- +2020-04-28T13:00:00+01:00 -> 2020-04-28T15:00:00+01:
```

Note that nanoperiod is time zone correct, even on the rare hourly events where a transition occurs from a time zone offset with an hourly difference to a time zone offset with a half-hourly difference.

```
print(seq(as.nanotime("2006-04-14 22:00:00 Asia/Colombo"),
          by=as.nanoperiod("01:00:00"),
          length.out=4,
          tz="Asia/Colombo"),
      tz="Asia/Colombo")
#  [1] 2006-04-14T22:00:00+06:00 2006-04-14T23:00:00+06:00 2006-04-15T00:00:00+06:00 2006-04-15T01:00:00+05:30
```

Year/month/day. Utilities are provided for getting in numerical format the day of the week (nano_wday), the day of the month (nano_mday), the month (nano_month) and the year (nano_year) from a given nanotime. Remember that a time zone is never associated with a nanotime and therefore, to have meaning, all the functions take as second argument the time zone for the computation. Note that the convention for the day of the week is a count from 0 to 6, with 0 falling on Sunday.

```
tm  <- as.nanotime("2019-12-31 20:00:00", tz="UTC")
nano_wday(tm, "Australia/Melbourne")
#  [1] 3
nano_wday(tm, "America/New_York")
#  [1] 2
nano_mday(tm, "Africa/Nairobi")
#  [1] 31
nano_month(tm, "Indian/Reunion")
#  [1] 1
nano_year(tm, "Asia/Irkutsk")
#  [1] 2020
```

Rounding operations. The functions nano_floor and nano_ceiling are provided in order to perform rounding to an arbitrary precision. An origin argument of type nanotime can be optionally specified so there is full control over the reference chosen for the rounding.

These functions are also to be understood in the context of vectors of nanotime where the precision defines a grid interval. These functions will pick a reasonable reference for the alignment. In particular, when using a nanoperiod, the functions will check if a precision is a multiple of a larger unit. If so, the rounding will happen with the larger unit as origin. For instance, if the precision is 6 hours - a multiple of a day - the rounding will be performed in such a way as to align the vector with a day, i.e. the rounding will be done at hours 0, 6, 12 and 18. On the other hand, if the origin is explicitly specified, then it is this value that will be taken as starting point for the rounding. For instance, if the origin is set to 2020-04-27 23:57:04 then the rounding will be done at 23:57:04, 05:57:04, 11:57:04 and 17:57:04.

```
nano_floor(as.nanotime("2020-04-27 23:57:04.123456678 UTC"), as.nanoduration("00:00:00.001"))
#  [1] 2020-04-27T23:57:04.123+00:00
nano_ceiling(as.nanotime("2020-04-27 23:57:04.123456678 UTC"), as.nanoduration("00:00:00.001"))
#  [1] 2020-04-27T23:57:04.124+00:00

nano_floor(as.nanotime("2020-04-27 23:57:04 UTC"), as.nanoperiod("06:00:00"), tz="UTC")
#  [1] 2020-04-27T18:00:00+00:00
nano_ceiling(as.nanotime("2020-04-27 23:57:04 UTC"), as.nanoperiod("06:00:00"), tz="UTC")
#  [1] 2020-04-28T00:00:00+00:00

nano_floor(as.nanotime("2020-04-27 23:57:04 America/New_York"), as.nanoperiod("1m"), tz="America/New_York")
#  [1] 2020-04-01T04:00:00+00:00
nano_ceiling(as.nanotime("2020-04-27 23:57:04 America/New_York"), as.nanoperiod("1m"), tz="America/New_York")
#  [1] 2020-05-01T04:00:00+00:00
```

Use with data.frame and data.table

All the new types are compatible with data.frame and data.table. By having an ordered nanotime column it is thus easy to define a time-series. One can then use nanoival subsetting.

```
idx <- seq(nanotime("2020-04-02+00:00"), by=as.nanoperiod("1d"), length.out=20, tz="UTC")
dt <- data.table(idx, v1=1:20, v2=c(TRUE, FALSE))
ival <- as.nanoival(c("+2020-04-05 UTC -> 2020-04-07 UTC+",
                      "+2020-04-15 UTC -> 2020-04-17 UTC+"))
dt[idx %in% ival]
#                        idx v1    v2
#  1: 2020-04-05T00:00:00+00:00  4 FALSE
#  2: 2020-04-06T00:00:00+00:00  5  TRUE
#  3: 2020-04-07T00:00:00+00:00  6 FALSE
#  4: 2020-04-15T00:00:00+00:00 14 FALSE
#  5: 2020-04-16T00:00:00+00:00 15  TRUE
#  6: 2020-04-17T00:00:00+00:00 16 FALSE
```

With the rounding functions, it is possible to perform aggregations on `data.table` instances:

```
idx <- seq(as.nanotime("2020-03-08 UTC"), as.nanotime("2020-03-10 UTC"), by=as.nanoduration("00:01:00"))
dt <-  data.table(idx, a=1:length(idx))
dt[, .(mean=mean(a)), by=nano_ceiling(idx, as.nanoduration("06:00:00"))]
#                nano_ceiling   mean
#  1: 2020-03-08T00:00:00+00:00    1.0
#  2: 2020-03-08T06:00:00+00:00  181.5
#  3: 2020-03-08T12:00:00+00:00  541.5
#  4: 2020-03-08T18:00:00+00:00  901.5
#  5: 2020-03-09T00:00:00+00:00 1261.5
#  6: 2020-03-09T06:00:00+00:00 1621.5
#  7: 2020-03-09T12:00:00+00:00 1981.5
#  8: 2020-03-09T18:00:00+00:00 2341.5
#  9: 2020-03-10T00:00:00+00:00 2701.5
```

Input and output Format

nanotime. The input and output format is by default "%Y-%m-%dT%H:%M:%EXS%Ez" where the 'X' specifies a variable number of digits for the nanosecond portion. When no overriding format is defined, the output will include only the relevant nanotime precision for the vector without right 0 padding; for example:

```
2020-12-12T00:00:00+00:00 which is equivalent to 2020-12-12T00:00:00.000000000+00:00
2020-12-12T00:00:00.123+00:00 which is equivalent to 2020-12-12T00:00:00.123000000+00:00
2020-12-12T00:00:00.123456+00:00 which is equivalent to 2020-12-12T00:00:00.123456000+00:00
2020-12-12T00:00:00.123456789+00:00
```

When no overriding format is defined (see ...) the parsing has some flexibility and the time portion can be omitted. Additionally, the separator `'_'` can be used to separate nanosecond groups of 3. So the following examples will parse correctly:

```
2020-04-03
2020-04-03 12:23:00
2020-04-03 12:23:00.1
2020-04-03 12:23:00.123
2020-04-03 12:23:00.123356789
2020-04-03 12:23:00.123_356_789
```

Date separators can be `' '`, `'-'` and `'/'` whereas the separator between date and time can be 'T' or `' '`:

```
2020 04 03
2020/04/03
2020-04-03T12:23:00
```

nanoival. The format is based on `nanotime` because a `nanoival` is composed of a `nanotime` start and end as well as two booleans that indicate if the boundaries of the interval are open or closed. This open and closed is indicated by prefixing and postfixing with the the characters `'-'` and `'+'`. The start and end are separated by `'->'`. Here are a few examples:

```
+2020-12-12 UTC -> 2020-12-13 UTC-
-2020-12-12T00:00:01.123 America/New_York -> 2020-12-14+00:00+
```

Table 1. Comparison of as.nanotime, base R and fasttime

test	replications	elapsed	relative
as.nanotime	10000	0.366	3.211
as.nanotime_with_format	10000	0.487	4.272
as.nanotime_with_tz	10000	0.490	4.298
as.POSIXct	10000	6.137	53.833
fastPOSIXct	10000	0.114	1.000

nanoduration. The format is immutable and the same as the hour/minute/second/nanosecond portion of a `nanotime`; for example:

```
12:23:00
12:23:00.1
12:23:00.123
12:23:00.123356789
12:23:00.123_356_789
```

nanoperiod. The format is here too immutable and is composed of two parts, a month/day part and a `nanoduration` part that are separated by `'/'`. In input, years, months, weeks and days are specified with a signed integer following, respectively by the letters `'y'`,`'m'`,`'w'`,`'d'`. The `nanoduration` that composes the second part is specified like for a standalone `nanoduration`. Each of these two parts is optional. In output, only months and days are specified as years can be expressed as 12 months and weeks as 7 days. Here are some examples:

```
1y1m1w1d/00:00:00.123 which is, in output, simplified to 13m8d/00:00:00.123
-2y
00:00:00.123
12m1s/01:00:00
```

Technical Details

All four new types in this package are built with S4 classes containing an R primitive type that is then reinterpreted. `nanotime` and `duration` are (indirectly) based on `double` via the type `integer64` from the **bit64** package (Oehlschlägel, 2015), whereas `nanoival` and `period` are based on `complex`, which allows the storage of 128 bits.

All the heavy lifting is done at C++ level using the **Rcpp** package (Eddelbuettel *et al.*, 2019).

Performance

The `as.POSIXct` function in R provides a useful baseline as it is also implemented in compiled code. The `fastPOSIXct` function from the **fasttime** package (Urbanek, 2016) excels at converting one (and only one) input format *fast* to a (UTC-only) datetime object. A simple benchmark converting 100 input strings 10,000 times shows that the `nanotime` constructor is much closer to the optimal `fastPOSIXct` (see Table 1).

Summary

We describe the **nanotime** package which offers a coherent set of types and operations with nanosecond precision.

We show that the **nanotime** package provides the building blocks to build more complicated and interesting functions, in particular within the context of `data.table` time-series.

Appendix

The benchmark results shown in table 1 are based on the code included below, and obtained via execution under R version 3.6.3 running under Ubuntu 20.04 with Linux kernel 5.4.0-25 on an Intel Xeon E-2176M CPU.

```r
library(nanotime)
library(rbenchmark)
library(fasttime)


x_posixct       <- rep("2020-03-19 22:55:23", 100)
x_nanotime      <- rep("2020-03-19 22:55:23.000000001+00:00", 100)
x_nanotime_tz   <- rep("2020-03-19 22:55:23.000000001 America/New_York", 100)
x_nanotime_cctz <- rep("03-19-2020 22:55:23.000000001+00:00", 100)
```

```
benchmark(
    "as.POSIXct" = { x <- as.POSIXct(x_posixct) },
    "fastPOSIXct" = { x <- fastPOSIXct(x_posixct) },
    "as.nanotime" = { x <- as.nanotime(x_nanotime) },
    "as.nanotime with tz" = { x <- as.nanotime(x_nanotime_tz) },
    "as.nanotime with format" = { x <- as.nanotime(x_nanotime_tz, format="%m-%d-%YT%H:%M:%E9S%Ez") },
    replications = 10000,
    columns = c("test", "replications", "elapsed", "relative"))
#                        test replications elapsed relative
#  3             as.nanotime        10000   0.287    2.842
#  5 as.nanotime with format        10000   0.375    3.713
#  4     as.nanotime with tz        10000   0.373    3.693
#  1              as.POSIXct        10000   1.687   16.703
#  2             fastPOSIXct        10000   0.101    1.000
```

References

(2019). *Joda-Time*. URL https://www.joda.org/joda-time/.

Eddelbuettel D, François R, Allaire J, Ushey K, Kou Q, Russel N, Chambers J, Bates D (2019). *Rcpp: Seamless R and C++ Integration*. R package version 1.0.1, URL https://CRAN.R-project.org/CRAN=package=Rcpp.

Eddelbuettel D, Silvestri L (2020). *nanotime: Nanosecond-Resolution Time for R*. R package version 0.2.4.5, URL https://CRAN.R-project.org/package=nanotime.

Evans B, Warburton R (2014). "Java SE 8 Date and Time." URL http://www.oracle.com/technetwork/articles/java/jf14-date-time-2125367.html.

Grolemund G, Wickham H (2011). "Dates and Times Made Easy with lubridate." *Journal of Statistical Software*, **40**(3), 1–25. URL http://www.jstatsoft.org/v40/i03/.

Oehlschlägel J (2015). *bit64: A S3 Class for Vectors of 64bit Integers*. R package version 0.9-5, URL https://CRAN.R-project.org/package=bit64.

The Open Group (2016). "The Open Group Base Specifications Issue 7, Rationale, section 4.16 Seconds Since the Epoch." *Technical Report Std 1003.1-2008*. URL http://pubs.opengroup.org/onlinepubs/9699919799/xrat/V4_xbd_chap04.html#tag_21_04_16.

Urbanek S (2016). *fasttime: Fast Utility Function for Time Parsing and Conversion*. R package version 1.0.2, URL https://CRAN.R-project.org/package=fasttime.