xts: High Performance Time Series

Presented at Saint Louis RUG
February 17, 2015
Why yet another time series class?

- Multiple time-series and date/time classes
  - Data: stats::ts, tseries::irts, timeSeries, its, zoo
  - Time: Date, POSIXt, chron, timeDate, yearmon, yearqtr
- Before xts, developers had to:
  - choose the classes they would support, or
  - require the end-user perform conversions
- Coercion via as.* calls often lose attributes
  - Converting from timeSeries to zoo would cause FinCenter, format, and recordID attributes to be lost
What is xts?

- The xts class extends zoo
  - Most details of zoo objects also apply to xts objects
    - S3 class (essentially a matrix with an index attribute)
    - Natural, R-like interface with flexible methods
  - Much in the zoo vignettes also applies to xts
- Written by Jeffrey Ryan and me
What is xts?

- xts modifications
  - True time-based index
    - zoo index can be anything, as long as it is ordered
  - Subset by ISO-8601 time strings
  - Preserve attributes when converting to/from other classes
  - Arbitrary user attributes via xtsAttributes
What is xts?

- Other departures from zoo
  - `lag.xts` does not follow convention of `lag.ts` and `lag.zoo`
    - i.e. `lag.xts(x, 1)` # shifts data back one observation
  - `.index*` functions access POSIXlt components
    - e.g. `.indexhour()` # hour of the day for each row
  - Time-based “apply” functions
    - `period.apply`, `endpoints`, `to.period`
  - Attempt to better handle objects with non-unique index values
Time-based index

- User must provide an index of class Date, POSIXct, chron, yearmon, yearqtr, or timeDate, but...
  - It's POSIX time internally (seconds since epoch)
    - Simplifies xts' C-level code
    - Avoids internal conversion to/from other time-based classes
    - Details are hidden from the user
- `index()` returns the class specified by the user
- `.index()` returns the raw seconds
Subset by ISO 8601 time strings

- Date/times ordered from *most to least significant*
  - YYYY-MM-DD hh:mm:ss [.S]
    - Separators can be omitted: YYYYYMMDDhhmmss
    - Fractional seconds supported (depending on OS)
- All components need not be specified
  - YYYYYMM is valid
- Intervals can be expressed
  - “2015/2015-02-17”
Subset by ISO 8601 time strings

• **Examples**

```r
require(xts)
data(sample_matrix)
x <- as.xts(sample_matrix)
x['2007']         # all of 2007
x['2007-03']      # all of March 2007
x['/2007-01-07']  # 1st obs through January 7, 2007
x['2007/200802']  # January 2007 through February 2008
```

• **ISO 8601 subsetting is done via binary search**
  • Similar speed regardless of object size
Subset by ISO 8601 time strings

• Time-of-day subsetting
  • Extract same time interval from every day
    \[\text{xi} \leftarrow \text{xts}(1:50, \text{seq}(\text{as.POSIXct}("1970-01-01"), \text{as.POSIXct}("1970-01-03")-1, \text{length}=50))\]
    \[\text{yi} \leftarrow \text{xi}[\text{T06:00/T17:00}',] \quad \# \text{6am-5pm}\]

• Requirements
  • Leading zeros (‘\text{T06:00}’, not ‘\text{T6:00}’)
  • 24-hour time (no AM/PM)
Other subsetting tricks

- \texttt{.index*} family of functions

```
library(quantmod)
getSymbols("SPY")
SPY[.indexmon(SPY)==0]  # All Januaries (note zero-based index!)
SPY[.indexmday(SPY)==1] # The first of every month
SPY[.indexwday(SPY)==1] # All Mondays
```

- first and last

```
# last 2 days of the first 3 weeks of my_xts
last(first(my_xts, "3 weeks"), "2 days")
```
Merging

- Custom C-based sort-merge-join algorithm
- Flexible
  - Supports multiple types of joins for 2-object merges
    
    ```r
    x <- xts(4:10, Sys.Date()+4:10)
y <- xts(1:6, Sys.Date()+1:6)
merge(x, y)        # either
merge(x, y, join='inner')  # both
merge(x, y, join='left')   # merge to left
merge(x, y, join='right')  # merge to right
    ```
  - Multi-object merges are also possible
    - Only full-inner and full-outer are supported
Merging

• Fast (even on Big Data™)
  • 10 million rows, 2 columns (~2GB)
    – Not really Big Data™
    – Max memory during merge is ~10GB

Data <- matrix(numeric(2e8),1e8,2)
x <- .xts(Data, 1:1e8)
y <- .xts(Data, 1:1e8)
system.time(z <- merge(x,y))
#    user  system elapsed
#   3.091   0.524   3.616
print(object.size(x), units="Gb")
# 1.9 Gb
Time-based apply functions

- **period.apply**
  - Use endpoints to create non-overlapping blocks
    
    ```r
    require(xts)
data(sample_matrix)
x <- as.xts(sample_matrix)
THE_answer <- period.apply(x, endpoints(x, "days", 42), mean)
    ```

- **apply.(daily, weekly, monthly, quarterly, yearly)**
  - Handle endpoints for you
    - some_answer <- apply.quarterly(x, mean)

- **rollapply**
  - Apply a function to a rolling subset of data
Irregular data... to regular data

- Analyze irregular data at regular intervals

```r
irreg.xts <- xts(1:6, .POSIXct(c(1253623410, 1254387030, 1254404700, 1254408495, 1257957030, 1257959565), tz="UTC"))
# align index into regular (e.g. 3-hour) blocks
reg.xts <- align.time(irreg.xts, n=60*60*3)

# apply your function (e.g. length) to each block
enp <- endpoints(reg.xts, "hours", 3)
count <- period.apply(reg.xts, np, length)
# create an empty xts object with the desired regular index
empty.xts <- xts(, seq(start(reg.xts), end(reg.xts), by="3 hours"))

# merge the counts with the empty xts object
out1 <- merge(empty.xts, count) # leave NA
out2 <- merge(empty.xts, count, fill=na.locf) # fill NA w/last obs
out3 <- merge(empty.xts, count, fill=0) # fill NA with zeros
```
Graphics

- Ross Bennett overhauled \texttt{plot.xts} for a Google Summer of Code (GsoC) project
  - Contained in \texttt{xts::plot.xts} on R-Forge
  - Based on \texttt{quantmod::chart_Series}
- dygraphs
  - Interactive time series plotting via \texttt{xts} and JavaScript
Graphics

- A plot.xts (from R-Forge) example

```r
data(sample_matrix)
sample.xts <- as.xts(sample_matrix)

# plot the close and add a panel with returns
plot(sample.xts$Close)
lines(TTR::ROC(sample.xts$Close), type="h")

# add the 50 period simple moving average to panel 1
lines(TTR::SMA(sample.xts$Close, n=50), on=1, col="blue")

# add legend to panel 1
addLegend("topright", on=1, legend.names=c("Close", "SMA(50)")
  lty=c(1, 1), lwd=c(2, 1), col=c("black", "blue", "red"))
```
Graphics
Developing with xts

- **Use one class internally (xts, of course)**
  - **try.xts**: attempt to convert to xts
    - error argument can be:
      - logical: should error halt processing?
      - character: error message printed to console
      - function: called on 'x' if as.xts(x, …) fails
  - **reclass**: convert back to original class
    - match.to argument:
      - The object returned from try.xts call
      - must be untouched, unless returned object has changed dimension
        (see example on next slide)
Developing with xts

• try.xts / reclass example

```r
period.apply <- function (x, INDEX, FUN, ...) {
  x <- try.xts(x, error = FALSE)
  FUN <- match.fun(FUN)
  xx <- sapply(1:(length(INDEX) - 1), function(y) {
    FUN(x[(INDEX[y] + 1):INDEX[y + 1]], ...)
  })
  if (is.vector(xx))
    xx <- t(xx)
  xx <- t(xx)
  if (is.null(colnames(xx)) && NCOL(x) == NCOL(xx))
    colnames(xx) <- colnames(x)
  reclass(xx, x[INDEX])
}
<environment: namespace:xts>
```
Developing with xts

• Manipulating xts objects from C/C++
  • xtsAPI.h
    – isXts, tryXts, lagXts
    – na_omit_xts, na_locf
    – do_merge_xts, do_rbind_xts
    – endpoints
  • RcppXts
    – Exposes the xts API for use in C++ code
xts “data.frame”

- Another GSoC project
  - Michael Weylandt, then Anton Samoylov
- Still work-in-progress
- Difficult to allow mixed types, while maintaining xts functionality and performance
Conclusion

• xts provides users and developers an extensive set of time-based tools

• Extending zoo to require time-based indexing allows
  • knowledge of the object's purpose, which allows xts to provide useful utilities (e.g. time-based subsetting)

• Developers can use xts internally
  • They only have to code to one class
  • Users aren't required to use one specific class