Lagged objects in package 'lagged'

Georgi N. Boshnakov

May 20, 2017

Contents

This message is printed at the start of the tangled .R file to alert against editing that file:

```
## Do not edit this file manually.
## It has been automatically generated from *.org sources.
```

1 Class "Lagged"

"Lagged" is the base lagged class. It is virtual and defines a slot `data` from class "ANY".

```
setClass("Lagged", slots = c(data = "ANY"), contains = "VIRTUAL")
                               # setClass("Lagged", slots = c(data = "vector") )
                               # setClass("Lagged", slots = c(data = "structure") )
```

Actual classes inherit from "Lagged" and restrict the data slot. One special subclass of "Lagged" is "FlexibleLagged", which can represent objects from any subclass of "Lagged". This is achieved by setting the data slot to be "Lagged" along with methods for indexing, `maxLag`, and similar to ensure that the Lagged functionality is provided.

```
setClass("FlexibleLagged", contains = "Lagged", slots = c(data = "Lagged") )
```

The initilisation method for "FlexibleLagged" objects does the obvious thing if argument `data` is "Lagged". Otherwise it converts argument `data` to a suitable "Lagged" object before assigning it to the data slot. If the function is not able to infer a suitable "Lagged" class it still passes `data` on to the next method which usually leads to an error.

The following convenience function is used to infer a suitable "Lagged" class for argument `data`.:

```
.whichNativeLagged <- function(x){
    if(is(x, "Lagged"))
        "FlexibleLagged"
    else if(is.vector(x))
        "Lagged1d"
    else if(is.matrix(x))
        "Lagged2d"
    else if(is.array(x)  && length(dim(x)) == 3)
        "Lagged3d"
    else
        NA

}
```

This is the initialisation method. Note that it gets rid of recursive "FlexibleLagged" data slots, i.e. the data slot is of the returned object is "Lagged" but not "FlexibleLagged". This probably should be taken care of by a validation method.

```
setMethod("initialize", "FlexibleLagged",
          function(.Object, data, ...){
              if(missing(data))
                  return(callNextMethod(.Object, ...))

              while(is(data, "FlexibleLagged"))
                  data <- data@data

              if(!is(data, "Lagged")){
                  clname <- .whichNativeLagged(data)
                  if(!is.na(clname))
                      data <- new(clname, data = data)
                  ##else don't know what to do with data, pass it on
                  ##     and let others complain if not appropriate.
              }
              .Object <- callNextMethod(.Object, data = data, ...)

              .Object
          }
          )
```

In general, "FlexibleLagged" can be used as superclass of classes which wish to represent any possible subclasses of "Lagged". For slots, it is sufficient (and more efficient) to use "Lagged".

Since class "FlexibleLagged" is special, it has its own implementations of some core methods defined for "Lagged".

TODO: Decide what support to offer for the native S3 class "acf". Turn it into S4 using `setOldClass`? Or just adapt the various methods and constructors to convert it properly to Lagged? For now putting some code in `Lagged()` to accept "acf" objects.

1.1 Core methods for lagged objects

The methods in this section are ok for objects inheriting from "Lagged". Where necessary, specialised methods are defined for "FlexibleLagged".

1.1.1 Subscripting with "["

Subscripting with `i` missing, returns the raw data.

```
setMethod("[", c(x = "Lagged", i = "missing"), function(x) x@data )
setMethod("[", c(x = "FlexibleLagged", i = "missing"), function(x) x@data[] )
```

When `i` is present, indexing depends on the type of the data slot and so is defined by subclasses. For indices larger than `maxLag(x)` the values are filled with NA's.

TODO: consider making the 1d method the default one?

For "[", the default for `drop` is `FALSE`. **TODO:** check that the existing methods follow this convention!

Indexing "FlexibleLagged" simply transfers the operation to the data slot (it is "Lagged"):

```
setMethod("[", c(x = "FlexibleLagged"), function(x, i, ...) x@data[i, ...] )
```

1.1.2 Subscript-replacement with "[<-"

Similarly to "[", subscript-replacement "[<-" replaces the contents of the data. The method for "Lagged" does not check the validity of argument `value` but the assignment will raise an error if it is not appropriate. Subclasses that wish to provide finer control over this can define suitable methods (e.g. to coerce `value` appropriately).

```
setReplaceMethod("[", c(x = "Lagged", i = "missing"),
          function(x, i, value){
              x@data <- value
              x
          })
```

Assignment to "FlexibleLagged", when `i` is missing, attempts to coerce `value` to a suitable lagged class before assigning it (using `.whichNativeLagged()`, as the initialisation function does, but raising an error if unsuccessful). Further methods can be defined using `"value = xxx"` in the signature to accommodate additional types or overwrite the default method here.

```
setReplaceMethod("[", c(x = "FlexibleLagged", i = "missing"),
                 function(x, i, value){
                     if(is(value, "FlexibleLagged"))
                         x@data <- value@data
                     else if(is(value, "Lagged"))
                         x@data <- value
                     else{
                         clname <- .whichNativeLagged(value)
                         if(is.na(clname))
                             stop("Don't know what Lagged class to use for this value")
                         else
                             x@data <- new(clname, data = value) # as(value, clname)
                     }
                     x
                 })
```

When `i` is present, no attempt is made to coerce it:

```
setReplaceMethod("[", c(x = "FlexibleLagged", i = "numeric"),
                 function(x, i, value){
                     x@data[i] <- value # not i+1, since x@data is a "Lagged" object here.
                     x
          })
```

```
## Ne, tezi zasega ne gi pravya, pravya vischko bez "value = xxx" - tova pozvolyava da se
## definirat metodi ako tryabva.
##
## setReplaceMethod("[", c(x = "FlexibleLagged", i = "missing", value = "vector"),
##           function(x, i, value){
##               x@data <- as(value, "Lagged1d")
##               x
##           })
##
## setReplaceMethod("[", c(x = "FlexibleLagged", i = "missing", value = "matrix"),
##           function(x, i, value){
##               x@data <- as(value, "Lagged2d")
##               x
##           })
```

1.1.3 Methods for "[[" and "[[<-"

Indexing with "[[" returns the value for the specified lag. This is the recommended way to extract the value at a single index.

This defines a default method. For efficiency specific classes can define versions that avoid calling the generic "[". If multi-seasons are supported the check for length equal to one should be adapted accordingly.

```
setMethod("[[", c(x = "Lagged", i = "numeric"),
          function(x, i){
              if(length(i) == 1)
                  x[i, drop = TRUE]
              else
                  stop("length of argument ‘i’ must be equal to one")
          }
          )
```

Note the use of `drop = TRUE`.

TODO: The use of `drop = TRUE` maybe needs some further thought. Maybe something that drops only the index corresponding to the lag is preferable and such behaviour should be documented!

The replace method works similarly:

```
setReplaceMethod("[[", c(x = "Lagged", i = "numeric"),
                 function(x, i, value){
                     if(length(i) == 1)
                         x[i] <- value
                     else
                         stop("length of argument ‘i’ must be equal to one")
                     x
                 })
```

1.1.4 Arithmetic and other operations (Ops group)

Operations in the `Ops` group involving lagged objects are defined "naturally" on their data. However, they are more restrictive than base R's conventions for atomic objects and do not follow the recycling rules.

The "Ops" methods return values from one of the core lagged classes, even if the objects are from classes inheriting from "Lagged". The reason is that, for example, the difference between autocovariance functions is not necessarilly autocovariance, but it is still a lagged object. It would be very confusing if the result was not guaranteed to be "Lagged".

Of course, methods defined for subclasses of lagged objects may preserve the actual classes when appropriate but should not introduce confusion on indexing.

In the default methods below, the result of these operations is a strict lagged object, i.e. an object from the core lagged classes (**TODO:** explain). The exact type of lagged object is determined by the data. The net effect is that the value of the Ops operation is also a lagged object, a core one, with indexing starting from zero but additional structure is lost.

TODO: Should operations between "Lagged" and base R objects be permitted at all? For users of "Lagged" the returned "Lagged" value is natural and expected. But what about users who are not aware that there are "Lagged" objects among the arguments? What to do when the "ordinary" argument is of length one - should this be an exception? But then the user may not know that the length is one, leading to surprises. Also, there is a conceptual difference here between the additive and multiplicative operations. /All this should be documented in a vignette. It seems sufficient that the recycling rule is banned. Need to finalise operation with singletons./

Operations between two lagged objects give a lagged object. If their `maxLag()` properties are different, the shorter data slot is extended with NA's before applying the binary operation.

1.1.5 "Ops" involving "Lagged"

```
## TODO: do not allow mixing Lagged1d with Lagged2d, etc.?
setMethod("Ops", c(e1 = "Lagged", e2 = "Lagged"),
          function(e1, e2){
              wrk <- if(length(e1@data) == length(e2@data) ) # TODO: allow %%==0 as elsewhere?
                         callGeneric(e1@data, e2@data)
                     else{
                         maxlag <- max(maxLag(e1), maxLag(e2))
                         v1 <- e1[0:maxlag]
                         v2 <- e2[0:maxlag]
                         callGeneric(v1, v2)
                     }
              clname <- whichLagged(e1, e2)
              new(clname, data = wrk)
          })
```

TODO: the current mechanism to decide the lagged class of the return value is not very satisfactory, see `whichLagged()` which encapsulates it. Also, forbid mixing 1d with 2d, etc.?

When only one of the object is "Lagged", the operations are defined if:

1. the length of the other object is equal to the length of the data part of the "Lagged" object,

2. the other object is of length one,

3. the other object is a singleton with the same dimensions as a single element of the "Lagged" object.

old todo: document behaviour if `length(object@data) == 0` (minor issue)?

2017-05-20 TODO: Change `length(e1[[0]]) == length(e2))` below to `dim(e1[[0]]) == dim(e2))` but needs more care (note though that the scalar case is covered by `length(e2) == 1`.

Notice that "vector" in the signatures is the S4 class "vector" (TODO: check!), see `showClass("vector")` for its subclasses.

```
> is.vector(array(0, dim = c(2,2,2)))    # S3
[1] FALSE

> is(array(0, dim = c(2,2,2)), "vector") # S4
[1] TRUE
```

```
setMethod("Ops", c(e1 = "Lagged", e2 = "vector"),
          function(e1, e2){
              wrk <- if(length(e2) == 1  || length(e1@data) == length(e2)
                               # 2017-05-20 was:
                               #    || length(e2) > 0  && (length(e1@data) %% length(e2)) == 0
                          || length(e2) > 0  && length(e1[[0]]) == length(e2))
                          callGeneric(e1@data, e2)
                     else
                          stop("Incompatible length of operands in a binary operation")

              new(whichLagged(e1), data = wrk)
          })

setMethod("Ops", c(e1 = "vector", e2 = "Lagged"),
          function(e1, e2){
              wrk <- if(length(e1) == 1  || length(e1) == length(e2@data)
                               # 2017-05-20 was:
                               #    || length(e1) > 0  && (length(e2@data) %% length(e1)) == 0
                          || length(e1) > 0  && length(e2[[0]]) == length(e1))
                          callGeneric(e1, e2@data)
                     else
                          stop("Incompatible length of operands in a binary operation")

              new(whichLagged(e2), data = wrk)
          })
```

1.1.6 "Ops" involving "FlexibleLagged"

Operations involving "FlexibleLagged" objects use those defined for "Lagged" by operating on the data slot (which is "Lagged").

```
setMethod("Ops", c(e1 = "FlexibleLagged", e2 = "Lagged"),
          function(e1, e2){
              callGeneric(e1@data, e2)
          })

setMethod("Ops", c(e1 = "Lagged", e2 = "FlexibleLagged"),
          function(e1, e2){
              callGeneric(e1, e2@data)
          })

setMethod("Ops", c(e1 = "FlexibleLagged", e2 = "FlexibleLagged"),
          function(e1, e2){
              callGeneric(e1@data, e2@data)
          })


setMethod("Ops", c(e1 = "FlexibleLagged", e2 = "vector"),
          function(e1, e2){
              callGeneric(e1@data, e2)
          })

setMethod("Ops", c(e1 = "vector", e2 = "FlexibleLagged"),
          function(e1, e2){
              callGeneric(e1, e2@data)
          })
```

TODO: methods for "matrix", "array", these probably should be for specific "Lagged" subclasses, like "Lagged2d".

1.2 S3 methods for as.vector() and related functions for "Lagged"

```
## TODO: check if the S3 methods understand S4 inheritance (I think they do)
as.vector.Lagged <- function(x, mode) as.vector(x@data) # todo: use mode?
as.double.Lagged <- function(x, ...)  as.double(x@data ) # note: this is for as.numeric()
as.matrix.Lagged <- function(x, ...)  as.matrix(x@data)
 as.array.Lagged <- function(x, ...)  as.array(x@data)
```

Converting from "Lagged" to base atomic or structure objects applies the requested operation to the data slot. Define first the generic S3 methods:

Somewhat more efficient methods for these:

```
as.vector.Lagged1d <- function(x, mode) x@data
as.matrix.Lagged2d <- function(x, ...) x@data
as.array.Lagged3d  <- function(x, ...) x@data
```

1.3 setAs() methods for "Lagged"

These methods call the corresponding S3 methods defined above:

```
setAs("Lagged", "vector", function(from) as.vector(from) )
setAs("Lagged", "matrix", function(from) as.matrix(from) )
setAs("Lagged", "array",  function(from) as.array(from) )
```

1.4 Generic function maxLag()

The default method for `maxLag()` handles objects inheriting from S3 class "acf". In all other cases it raises an error. Notice that in "acf" the lag is in the first dimension.

```
maxLag <- function(object, ...){
   if(inherits(object, "acf"))
       dim(acf$acf)[1] - 1
   else
       stop("No applicable method to compute maxLag")
}

setGeneric("maxLag")
```

The convention for "Lagged" objects is that the last dimension carries the lag. So, the methods for basic objects compute the maximal lag as the last dimention minus one.

```
setMethod("maxLag", c(object = "vector"), function(object) length(object) - 1)
setMethod("maxLag", c(object = "matrix"), function(object) ncol(object) - 1 )
setMethod("maxLag", c(object = "array"),
          function(object){
                  d <- dim(object)
                  d[length(d)] - 1
          })
```

Note again that `acf()` puts the lag in the first index.

The `maxLag()` method for "Lagged" objects simply calls `maxLag()` on the data slot. Classes inheriting from "Lagged" may define specific methods if the (in)efficiency of this method is a concern.

```
setMethod("maxLag", c(object = "Lagged"), function(object) maxLag(object@data) )
```

1.5 Length of "Lagged" objects - S3 method for length()

The length of "Lagged" objects is defined to be `maxLag(x)+1`, not the length of the data in the "Lagged" object. In most cases of direct use `maxLag(x)` is more appropriate.

This defines an S3 method for function `length()` for "Lagged" objects.

```
length.Lagged <- function(x) maxLag(x) + 1
```

TODO: Check if other base R functions need S3 methods for "Lagged" objects.

2 Default classes and methods for lagged objects

```
                                                # setClass("X", slots = c(data = "structure"))
setClass("Lagged1d", contains = "Lagged", slots = c(data = "vector") )
setClass("Lagged2d", contains = "Lagged", slots = c(data = "matrix") )
setClass("Lagged3d", contains = "Lagged", slots = c(data = "array") )
                    # TODO: check validity for Lagged3d: 3 dimensional.
```

2.1 Methods for "["

```
setMethod("[", c(x = "Lagged1d", i = "numeric"),
          function(x, i, drop) x@data[i+1] )

## TODO: argument "drop"?
setMethod("[", c(x = "Lagged2d", i = "numeric"),
          function(x, i, drop = FALSE) x@data[ , i+1, drop = drop] )

## TODO: change autocovariances(), etc to this convention!!
setMethod("[", c(x = "Lagged3d", i = "numeric"),
          function(x, i, drop = FALSE) x@data[, , i+1, drop = drop] )
```

2.2 whichLagged()

For now `whichLagged()` is not exported. It could be exported to allow core "Lagged" classes defined in other packages to add functionality. But if it is to be exported, it would need streamlining. Currently it is a hack.

Making it generic is lazy but avoids writing obscure code but see note above. The default returns "FlexibleLagged".

```
.matLagged <- matrix("FlexibleLagged", 4, 4)
diag(.matLagged) <- c("FlexibleLagged", "Lagged1d", "Lagged2d", "Lagged3d")

rownames(.matLagged) <- c("FlexibleLagged", "Lagged1d", "Lagged2d", "Lagged3d")
colnames(.matLagged) <- c("FlexibleLagged", "Lagged1d", "Lagged2d", "Lagged3d")


whichLagged <- function(x, y){
    .matLagged[whichLagged(x), whichLagged(y)]
}
setGeneric("whichLagged")

## TODO: define methods for "numeric", "matrix", etc?
setMethod("whichLagged", c(x = "ANY"     , y = "missing"), function(x) "FlexibleLagged")
setMethod("whichLagged", c(x = "Lagged1d", y = "missing"), function(x) "Lagged1d")
setMethod("whichLagged", c(x = "Lagged2d", y = "missing"), function(x) "Lagged2d")
setMethod("whichLagged", c(x = "Lagged3d", y = "missing"), function(x) "Lagged3d")
```

2.3 Methods for "[<-"

```
setReplaceMethod("[", c(x = "Lagged1d", i = "numeric"),
          function(x, i, value){
              x@data[i+1] <- value
              x
          })

setReplaceMethod("[", c(x = "Lagged2d", i = "numeric"), #Include value = "matrix" in signature?
          function(x, i, value){
              x@data[ , i+1]  <- value
              x
          })

## Include value = "array" in the signature? Will still need to check the dimensions
setReplaceMethod("[", c(x = "Lagged3d", i = "numeric"),
          function(x, i, value){
                        # was: x@data[i+1, , ]  <- value
              x@data[ , , i+1]  <- value
              x
          })
```

3 show() methods

```
## .printVecOrArray <- function(x){
##     if(is.vector(x)){
##         if(is.null(names(x)) || length(names(x)) == 0)
##             names(x) <- paste0("Lag_", 0:(length(x) - 1))
##         print(x)
##     }else if(is.matrix(x)){
##         ## TODO:
##         print(x)
##     }else if(is.array(x)){
##         ## TODO:
##         print(x)
##     }else
##         print(x)
## }

setMethod("show", "Lagged1d",
          function(object){
              .reportClassName(object, "Lagged1d")
              cat("Slot *data*:", "\n")

              x <- object@data
              if(is.null(names(x)) || length(names(x)) == 0)
                  names(x) <- paste0("Lag_", 0:(length(x) - 1))
              print(x)
              ## cat("\n")
          }
          )

## TODO: "show", "Lagged2d"

setMethod("show", "Lagged3d",
          function(object){
              .reportClassName(object, "Lagged3d")
              cat("Slot *data*:", "\n")

              x <- object@data
              if(is.null(dimnames(x)) || length(dimnames(x)) == 0){
                  d <- dim(x)
                  dimnames(x) <- list(rep("", d[1]), rep("", d[2]),
                                      paste0("Lag_", 0:(d[3] - 1)) )
              }
              print(x)
              ## cat("\n")
          }
          )

## Commenting out since causes trouble by precluding default methods from printing.
##
## setMethod("show", "Lagged",
##           function(object){
##               ## .reportClassName(object, "Lagged") # this is silly: never writes!
##               ## callNextMethod()
##               wrk <- object@data
##               cat("Slot *data*:", "\n")
##               .printVecOrArray(wrk)
##               cat("\n")
##               ## callNextMethod() # in case the object inherits from other classes
##               ##                  # unfortunately, it prints slot data again.
##           }
##           )

setMethod("show", "FlexibleLagged",
          function(object){
              .reportClassName(object, "FlexibleLagged")
              cat("Slot *data*:", "\n")
              show(object@data)
          }
          )
```

4 Further constructors for lagged objects

Function `new()` can be used to create objects from the lagged classes. In this section we define some functions to make this more convenient.

First, a function to convert objects from S3 class "acf" (created by `acf()`) to "Lagged":

```
acf2Lagged <- function(x){
    acv <- x$acf
    d <- dim(acv)
    if(d[2] == 1 && d[3] == 1){
        data <- as.vector(acv)
        if(x$type == "partial") # lag-0 is missing, insert it
            data <- c(1, data)
        new("Lagged1d", data = data)
    }else{
        ## transpose to make the 3rd index corresponding to lag.
        ##   (taken from acfbase2sl() in package pcts, see the comments there)
        ##
        ## TODO: test!
        ## Note: in pcts:::acfbase2sl() the analogous command is aperm(acv, c(3,2,1))
        ##       i.e. R[k] is transposed => check if that is correct!
        data <- aperm(acv, c(2, 3, 1))

        if(x$type == "partial"){ # lag-0 is missing, insert it
            datanew <- array(NA_real_, dim(data) + c(0,0,1) )
            datanew[ , , -1] <- data
            data <- datanew
        }

        new("Lagged3d", data = data)
    }
}
```

Function "Lagged" looks at the supplied data argument and chooses an appropriate class inheriting from "Lagged". **TODO:** Make `Lagged()` generic?

```
Lagged <- function(data, ...){
    if(is.vector(data)){
        new("Lagged1d", data = data, ...)
    }else if(is.matrix(data)){
        new("Lagged2d", data = data, ...)
    }else if(is.array(data)){
        new("Lagged3d", data = data, ...)
    }else if(is(data, "Lagged")){
        new("FlexibleLagged", data = data, ...)
    }else if(inherits(data, "acf")){    # for S3 class "acf"
        acf2Lagged(data)
    }else
        stop("Cannot create a Lagged object from the given data")
}
```

TODO: Tests!