Writing functions with the "raster" package

Robert J. Hijmans

November 2, 2018

1 Introduction

The '`raster`' package has a number of 'low-level' functions (e.g. to read and write files) that allow you to write your own 'high-level' functions. Here I explain how to use these low-level functions in developing 'memory-safe' high-level functions. With 'memory-safe' I refer to function that can process raster files that are too large to be loaded into memory. To understand the material discussed in this vignette you should be already somewhat familiar with the raster package. It is also helpful to have some general knowledge of S4 classes and methods.

I should also note that if you are not sure, you probably do not need to read this vignette, or use the functions described herein. That is, the high-level functions in the raster package are already based on these techniques, and they are very flexible. It is likely that you can accomplish your goals with the functions already available in 'raster' and that you do not need to write your own.

2 How not to do it

Let's start with two simple example functions, f1 and f2, that are NOT memory safe. The purpose of these simple example functions is to add a numerical constant 'a' to all values of RasterLayer 'x'.

To test the functions, we create a RasterLayer with 100 cells and values 1 to 100.

```
> library(raster)
> r <- raster(ncol=10, nrow=10)
> r[] <- 1:100
> r
```

```
class       : RasterLayer
dimensions  : 10, 10, 100  (nrow, ncol, ncell)
resolution  : 36, 18  (x, y)
extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
```

```
data source : in memory
names       : layer
values      : 1, 100  (min, max)
```

Below is a simple function, f1, that we use to add 5 to all cell values of 'r'

```
> f1 <- function(x, a) {
+         x@data@values <- x@data@values + a
+         return(x)
+ }
> s <- f1(r, 5)
> s

class       : RasterLayer
dimensions  : 10, 10, 100  (nrow, ncol, ncell)
resolution  : 36, 18  (x, y)
extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
data source : in memory
names       : layer
values      : 1, 100  (min, max)
```

f1 is a really bad example. It looks efficient but it has several problems. First of all, the slot x@data@values may be empty, which is typically the case when a raster object is derived from a file on disk. But even if all values are in memory the returned object will be invalid. This is because the values of a multiple slots need to be adjusted when changing values. For example, the returned 'x' may still point to a file (now incorrectly, because the values no longer correspond to it). And the slots with the minimum and maximum cell values have not been updated. While it can be OK (but normally not necessary) to directly read values of slots, you should not set them directly. The raster package has functions that set values of slots. This is illustrated in the next example.

```
> f2 <- function(x, a) {
+         v <- getValues(x)
+         v <- v + a
+         x <- setValues(x, v)
+         return(x)
+ }
> s <- f2(r, 5)
> s

class       : RasterLayer
dimensions  : 10, 10, 100  (nrow, ncol, ncell)
resolution  : 36, 18  (x, y)
extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
```

```
data source : in memory
names       : layer
values      : 6, 105  (min, max)
```

f2 is much better than f1. Function **getValues** gets the cell values whether they are on disk or in memory (and will return NA values if neither is the case). **setValues** sets the values to the RasterLayer object assuring that other slots are updated as well.

However, this function could fail when used with very large raster files, depending on the amount of RAM your computer has and that R can access, because all values are read into memory at once. Processing data in chunks circumvents this problem.

3 Row by row

The next example shows how you can read, process, and write values row by row.

```
> f3 <- function(x, a, filename) {
+         out <- raster(x)
+         out <- writeStart(out, filename, overwrite=TRUE)
+         for (r in 1:nrow(out)) {
+                 v <- getValues(x, r)
+                 v <- v + a
+                 out <- writeValues(out, v, r)
+         }
+         out <- writeStop(out)
+         return(out)
+ }
> s <- f3(r, 5, filename='test')
> s

class       : RasterLayer
dimensions  : 10, 10, 100  (nrow, ncol, ncell)
resolution  : 36, 18  (x, y)
extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
data source : c:\temp\Rtmp2RnLVF\Rbuild12fc69f5496e\raster\vignettes\test.grd
names       : layer
values      : 6, 105  (min, max)
```

Note how, in the above example, first a new empty RasterLayer, 'out' is created using the bf raster function. 'out' has the same extent and resolution as 'x', but it does not have the values of 'x'.

4 Multiple rows at once

Row by row processing is easy to do but it can be a bit inefficient because there is some overhead associated with each read and write operation. An alternative is to read, calculate, and write by block; here defined as a number of rows (1 or more). The function bf blockSize is a helper function to determine appropriate block size (number of rows).

```
> f4 <- function(x, a, filename) {
+         out <- raster(x)
+         bs <- blockSize(out)
+         out <- writeStart(out, filename, overwrite=TRUE)
+         for (i in 1:bs$n) {
+                 v <- getValues(x, row=bs$row[i], nrows=bs$nrows[i] )
+                 v <- v + a
+                 out <- writeValues(out, v, bs$row[i])
+         }
+         out <- writeStop(out)
+         return(out)
+ }
> s <- f4(r, 5, filename='test.grd')
> blockSize(s)

$row
[1]  1  4  7 10

$nrows
[1] 3 3 3 1

$n
[1] 4
```

5 Filename optional

In the above examples (functions f3 and f4) you must supply a filename. However, in the raster package that is never the case, it is always optional. If a filename is provided, values are written to disk. If no filename is provided the values are only written to disk if they cannot be stored in RAM. To determine whether the output needs to be written to disk, the function **canProcessInMemory** is used. This function uses the size of the output raster to determine the total memory size needed. Multiple copies of the values are often made when doing computations. And perhaps you are combining values from several RasterLayer objects in which case you need to be able to use much more memory than for the output RasterLayer object alone. To account for this you can supply an additional parameter, indicating the total number of copies needed. In the examples below we use '3', indicating that we would need RAM to hold

three times the size of the output RasterLayer. That seems reasonably safe in this case.

First an example for row by row processing:

```
> f5 <- function(x, a, filename='') {
+         out <- raster(x)
+         small <- canProcessInMemory(out, 3)
+         filename <- trim(filename)
+
+         if (!small & filename == '') {
+                 filename <- rasterTmpFile()
+         }
+
+         todisk <- FALSE
+         if (filename != '') {
+                 out <- writeStart(out, filename, overwrite=TRUE)
+                 todisk <- TRUE
+         } else {
+                 vv <- matrix(ncol=nrow(out), nrow=ncol(out))
+         }
+
+         for (r in 1:nrow(out)) {
+                 v <- getValues(x, r)
+                 v <- v + a
+                 if (todisk) {
+                         out <- writeValues(out, v, r)
+                 } else {
+                         vv[,r] <- v
+                 }
+         }
+         if (todisk) {
+                 out <- writeStop(out)
+         } else {
+                 out <- setValues(out, as.vector(vv))
+         }
+         return(out)
+ }
> s <- f5(r, 5)
```

Now, the same function, but looping over blocks of multiple rows, instead of a single row at a time (which can make a function very slow).

```
> f6 <- function(x, a, filename='') {
+         out <- raster(x)
+         small <- canProcessInMemory(out, 3)
+         filename <- trim(filename)
+
```

```
+         if (! small & filename == '') {
+                 filename <- rasterTmpFile()
+         }
+         if (filename != '') {
+                 out <- writeStart(out, filename, overwrite=TRUE)
+                 todisk <- TRUE
+         } else {
+                 vv <- matrix(ncol=nrow(out), nrow=ncol(out))
+                 todisk <- FALSE
+         }
+
+         bs <- blockSize(r)
+         for (i in 1:bs$n) {
+                 v <- getValues(x, row=bs$row[i], nrows=bs$nrows[i] )
+                 v <- v + a
+                 if (todisk) {
+                         out <- writeValues(out, v, bs$row[i])
+                 } else {
+                         cols <- bs$row[i]:(bs$row[i]+bs$nrows[i]-1)
+                         vv[,cols] <- matrix(v, nrow=ncol(out))
+                 }
+         }
+         if (todisk) {
+                 out <- writeStop(out)
+         } else {
+                 out <- setValues(out, as.vector(vv))
+         }
+         return(out)
+ }
> s <- f6(r, 5)
```

The next example is an alternative implementation that you might prefer if you wanted to optimize speed when values can be processed in memory. Optimizing for that situation is generally not that important as it tends to be relatively fast in any case. Moreover, while the below example is fine, this may not be an ideal approach for more complex functions as you would have to implement some parts of your algorithm twice. If you are not careful, your function might then give different results depending on whether the output must be written to disk or not. In other words, debugging and code maintenance can become more difficult. Having said that, there certainly are cases where processing chuck by chunk is inefficient, and where avoiding it can be worth the effort.

```
> f7 <- function(x, a, filename='') {
+
+         out <- raster(x)
```

```
+         filename <- trim(filename)
+
+         if (canProcessInMemory(out, 3)) {
+                 v <- getValues(x) + a
+                 out <- setValues(out, v)
+                 if (filename != '') {
+                         out <- writeRaster(out, filename, overwrite=TRUE)
+                 }
+         } else {
+                 if (filename == '') {
+                         filename <- rasterTmpFile()
+                 }
+                 out <- writeStart(out, filename)
+
+                 bs <- blockSize(r)
+                 for (i in 1:bs$n) {
+                         v <- getValues(x, row=bs$row[i], nrows=bs$nrows[i] )
+                         v <- v + a
+                         out <- writeValues(out, v, bs$row[i])
+                 }
+                 out <- writeStop(out)
+         }
+         return(out)
+ }
> s <- f7(r, 5)
```

6 A complete function

Finally, let's add some useful bells and whistles. For example, you may want to specify a file format and data type, be able to overwrite an existing file, and use a progress bar. So far, default values have been used. If you use the below function, the dots '...' allow you to change these by providing additional arguments 'overwrite', 'format', and 'datatype'. (In all cases you can also set default values with the **rasterOptions** function).

```
> f8 <- function(x, a, filename='', ...) {
+         out <- raster(x)
+         big <- ! canProcessInMemory(out, 3)
+         filename <- trim(filename)
+         if (big & filename == '') {
+                 filename <- rasterTmpFile()
+         }
+         if (filename != '') {
+                 out <- writeStart(out, filename, ...)
+                 todisk <- TRUE
```

```
+         } else {
+                 vv <- matrix(ncol=nrow(out), nrow=ncol(out))
+                 todisk <- FALSE
+         }
+
+         bs <- blockSize(x)
+         pb <- pbCreate(bs$n, ...)
+
+         if (todisk) {
+                 for (i in 1:bs$n) {
+                         v <- getValues(x, row=bs$row[i], nrows=bs$nrows[i] )
+                         v <- v + a
+                         out <- writeValues(out, v, bs$row[i])
+                         pbStep(pb, i)
+                 }
+                 out <- writeStop(out)
+         } else {
+                 for (i in 1:bs$n) {
+                         v <- getValues(x, row=bs$row[i], nrows=bs$nrows[i] )
+                         v <- v + a
+                         cols <- bs$row[i]:(bs$row[i]+bs$nrows[i]-1)
+                         vv[,cols] <- matrix(v, nrow=out@ncols)
+                         pbStep(pb, i)
+                 }
+                 out <- setValues(out, as.vector(vv))
+         }
+         pbClose(pb)
+         return(out)
+ }
> s <- f8(r, 5, filename='test', overwrite=TRUE)
> if(require(rgdal)) { # only if rgdal is installed
+         s <- f8(r, 5, filename='test.tif', format='GTiff', overwrite=TRUE)
+ }
> s

class       : RasterLayer
dimensions  : 10, 10, 100  (nrow, ncol, ncell)
resolution  : 36, 18  (x, y)
extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
data source : c:\temp\Rtmp2RnLVF\Rbuild12fc69f5496e\raster\vignettes\test.tif
names       : test
values      : 6, 105  (min, max)
```

Note that most of the additional arguments are passed on to writeStart:
`out <- writeStart(out, filename, ...)`
Only the `progress=` argument goes to `pbCreate`

7 Debugging

Typically functions are developed and tested with small (RasterLayer) objects. But you also need to test your functions for the case it needs to write values to disk. You can use a very large raster for that, but then you may need to wait a long time each time you run it. Depending on how you design your function you may be able to test alternate forks in your function by providing a file name. But this would not necessarily work for function f7. You can force functions of that type to treat the input as a very large raster by setting to option 'todisk' to TRUE as in setOptions(todisk=TRUE). If that option is set, 'canProcessInMemory' always returns FALSE. This should only be used in debugging.

8 Methods

The raster package is build with S4 classes and methods. If you are developing a package that builds on the raster package I would advise to also use S4 classes and methods. Thus, rather than using plain functions, define generic methods (where necessary) and implement them for a RasterLayer, as in the example below (the function does not do anything; replace 'return(x)' with something meaningful along the pattern explained above.

```
> if (!isGeneric("f9")) {
+         setGeneric("f9", function(x, ...)
+                 standardGeneric("f9"))
+ }

[1] "f9"

> setMethod('f9', signature(x='RasterLayer'),
+         function(x, filename='', ...) {
+                 return(x)
+         }
+ )
> s <- f9(r)
```

9 Mutli-core functions

Below is an example of an implementation of a customized raster function that can use multiple processors (cores), via the snow package. This is still very much under development, but you can try it at your own risk.

First we define a snow cluster enabled function. Below is a simple example:

```
> clusfun <- function(x, filename="", ...) {
+
+         out <- raster(x)
```

```
+
+         cl <- getCluster()
+         on.exit( returnCluster() )
+
+         nodes <- length(cl)
+
+         bs <- blockSize(x, minblocks=nodes*4)
+         pb <- pbCreate(bs$n)
+
+         # the function to be used (simple example)
+         clFun <- function(i) {
+                 v <- getValues(x, bs$row[i], bs$nrows[i])
+                 for (i in 1:length(v)) {
+                         v[i] <- v[i] / 100
+                 }
+                 return(v)
+         }
+
+         # get all nodes going
+         for (i in 1:nodes) {
+                 sendCall(cl[[i]], clFun, i, tag=i)
+     }
+
+         filename <- trim(filename)
+         if (!canProcessInMemory(out) & filename == "") {
+                 filename <- rasterTmpFile()
+         }
+
+         if (filename != "") {
+                 out <- writeStart(out, filename=filename, ... )
+         } else {
+                 vv <- matrix(ncol=nrow(out), nrow=ncol(out))
+         }
+         for (i in 1:bs$n) {
+                 # receive results from a node
+                 d <- recvOneData(cl)
+
+                 # error?
+                 if (! d$value$success) {
+                         stop('cluster error')
+                 }
+
+                 # which block is this?
+                 b <- d$value$tag
+                 cat('received block: ',b,'\n'); flush.console();
+
```

```
+                 if (filename != "") {
+                         out <- writeValues(out, d$value$value, bs$row[b])
+                 } else {
+                         cols <- bs$row[b]:(bs$row[b] + bs$nrows[b]-1)
+                         vv[,cols] <- matrix(d$value$value, nrow=out@ncols)
+                 }
+
+                 # need to send more data?
+                 ni <- nodes + i
+                 if (ni <= bs$n) {
+                         sendCall(cl[[d$node]], clFun, ni, tag=ni)
+                 }
+                 pbStep(pb)
+         }
+         if (filename != "") {
+                 out <- writeStop(out)
+         } else {
+                 out <- setValues(out, as.vector(vv))
+         }
+         pbClose(pb)
+
+         return(out)
+ }
```

To use snow with raster, you need to set up a cluster first, with 'beginCluster' (only once during a session). After that we can run the function:

```
> r <- raster()
> # beginCluster()
> r[] <- ncell(r):1
> # x <- clusfun(r)
> # endCluster()
```