Writing functions with the "raster" package

Robert J. Hijmans

April 18, 2010

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 this article you should be already somewhat familiar with the raster package. It is also helpful to have some general knowledge of S4 classes and methods.

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
filename    :
nrow        : 10
ncol        : 10
ncell       : 100
min value   : 1
max value   : 100
projection  : +proj=longlat +datum=WGS84
xmin        : -180
xmax        : 180
ymin        : -90
ymax        : 90
```

```
xres        : 36
yres        : 18
```

Now we write a simple function, f1, and use it 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
filename    :
nrow        : 10
ncol        : 10
ncell       : 100
min value   : 1
max value   : 100
projection  : +proj=longlat +datum=WGS84
xmin        : -180
xmax        : 180
ymin        : -90
ymax        : 90
xres        : 36
yres        : 18
```

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 may need to be adjusted when chaging values. For example, the returned 'x' may still point to a file (now incorectly, because the values no longer correspond). And the slots with the minimum and maximum values have not been updated. While it is OK (but normally not necessary) to directly read values of slots, you should not set them directly. The raster package has functions set values of slots as shown 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
filename    :
nrow        : 10
ncol        : 10
ncell       : 100
min value   : 6
max value   : 105
projection  : +proj=longlat +datum=WGS84
xmin        : -180
xmax        : 180
ymin        : -90
ymax        : 90
xres        : 36
yres        : 18
```

f2 is much better. 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 for very large raster files, depending on the amount of RAM your computer has and R can access, because all values are read into memory at once. Processing data in chuncks 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
+                 writeValues(out, v, r)
+         }
+         out <- writeStop(out)
+         return(out)
+ }
> s <- f3(r, 5, filename='test')
> s

class       : RasterLayer
filename    : test.grd
nrow        : 10
ncol        : 10
```

```
ncell       : 100
min value   : Inf
max value   : -Inf
projection  : +proj=longlat +datum=WGS84
xmin        : -180
xmax        : 180
ymin        : -90
ymax        : 90
xres        : 36
yres        : 18
```

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 Block by block

Row by row processing is easy to do but it can be a bit inefficient. 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(r)
+         out <- writeStart(out, filename, overwrite=TRUE)
+         for (i in 1:bs$n) {
+                 v <- getValuesBlock(x, row=bs$row[i], nrows=bs$nrows[i] )
+                 v <- v + a
+                 writeValues(out, v, bs$row[i])
+         }
+         out <- writeStop(out)
+         return(out)
+ }
> s <- f4(r, 5, filename='test.grd')
> blockSize(s)

$size
[1] 5

$row
[1] 1 6

$nrows
[1] 5 5
```

```
$n
[1] 2
```

5 Filename optional

In the above examples (functions f3 and f4) you must supply a filename. 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 bf canProcessInMemory is used. This function uses the size of the output raster to determine the total memory size needed. However, additional 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 memory need. In the examples below we use '3', indicating that we would need RAM equivalent to 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)
+         big <- canProcessInMemory(out, 3)
+         filename <- trim(filename)
+         if (big & filename == '') {
+                 filename <- rasterTmpFile()
+         }
+         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) {
+                         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 using blocks:

```
> f6 <- 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, overwrite=TRUE)
+                 todisk <- TRUE
+         } else {
+                 vv <- matrix(ncol=nrow(out), nrow=ncol(out))
+         }
+
+         bs <- blockSize(r)
+         for (i in 1:bs$n) {
+                 v <- getValuesBlock(x, row=bs$row[i], nrows=bs$nrows[i] )
+                 v <- v + a
+                 if (todisk) {
+                         writeValues(out, v, bs$row[i])
+                 } else {
+                         cols <- bs$row[i]:(bs$row[i]+bs$nrows[i]-1)
+                         vv[,cols] <- matrix(vv, nrow=out@ncols)
+                 }
+         }
+         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 optimze 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 when possible is worth the effort.

```
> f7 <- function(x, a, filename='') {
+         out <- raster(x)
+         big <- canProcessInMemory(out, 3)
+
+         if (!big) {
+                 v <- getValues(x) + a
+                 out <- setValues(out, v)
+                 if (filename != '') {
+                         out <- writeRaster(out, filename, overwrite=TRUE)
+                 }
+                 return(out)
+         }
+
+         filename <- trim(filename)
+         if (filename == '') {
+                 filename <- rasterTmpFile()
+         }
+         out <- writeStart(out, filename)
+
+         bs <- blockSize(r)
+         for (i in 1:bs$n) {
+                 v <- getValuesBlock(x, row=bs$row[i], nrows=bs$nrows[i] )
+                 v <- v + a
+                 writeValues(out, v, bs$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', 'datatype', and 'progress'. (In all cases you can also set default values with **setOptions**).

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

class       : RasterLayer
filename    : /srv/R/pkgs/raster/inst/doc/test.tif
nrow        : 10
ncol        : 10
```

```
ncell       : 100
min value   : Inf
max value   : -Inf
projection  : +proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs +towgs84=0,0,0
xmin        : -180
xmax        : 180
ymin        : -90
ymax        : 90
xres        : 36
yres        : 18
```

Note that most of the additional arguments are passed on to writeStart:
out <- writeStart(out, filename, ...)

The only exception is 'progress'. To set the type of the progress bar we use the hidden function .progress().Hidden functions are normally only used internally, but here is an exception. To access a hidden function, use the package name followed by three colons:

pb <- pbCreate(bs$n, type= raster:::.progress(...))

By providing it with a dots argument it will use the value of argument 'progress' if such an argument was provided to the main function, and the default value (normally '') if no such argument was provided.

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. Depening on how you desing 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, rater 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)
+         }
+ )

[1] "f9"

> s <- f9(r)
```