# Geostatistics and Image Analysis

Henning Bredel[*] Daniel Nüst[†]

March 9, 2009

R is becoming one of the major platforms to perform spatial analysis. As it is designed to analyse typical statistical data sets, fundamental problems arise when huge data, like large images with multiple bands in remote sensing, are to be analysed. Several approaches are under development to allow R to operate with large data sets. So far, no comparison has been made which of them render R useful for image analysis and how their performance is.

In this paper we give a short examplary view on the underlying problem in Rs memory handling and the most common packages for image handling. We introduce different approaches to overcome the memory limitations. In the main part we present and discuss our work with `bigmemory` and `ff`—two projects that promise improvement in these premises. We describe the process to combine packages that include image functions with `ff` and `bigmemory`. In the end we point out future work and possibilities.

**Keywords:** R-project, image analysis, remote sensing, ff, bigmemory

[*] henning.bredel@uni-muenster.de
[†] daniel.nuest@uni-muenster.de

## 1. Introduction

### 1.1. R and its limitations

"R is a language and environment for statistical computing and graphics" [5]. There are many applications using R as a toolbox for statistical analysis. No surprise: Currently more than 1,600 packages[1] are available to add functionality in a modular way. A main reason for this huge functionality is rooted in the fact, that R is free open source software[2], platform independent and has a strong user base contributing valuable content.

Besides these facts R has a few other reasons to explain its popularity: It provides a collection of many common statistical methods (tests, classification) and visualization tools out of the box, and allows quick production of statistics and well-designed plots for beginners and full control for experienced users.

The possibility to use R as toolbox makes it easy to integrate Rs functionality into other Software like Grass, QGis, etc. This kind of "software combination" adds functionality to the users workbench. Even the most powerful software cannot handle all specific kind of problems perfectly on its own.

Although R performs well with typical statistical sample data, it is problematic when dealing with huge image datasets like satellite data. The reason can be found in Rs approach to data handling, i. e. to load all data into the working memory (RAM) completely.

This is a well known problem which leads to statements one should better use other software than R to perform image analysis (see [15], [14]). Mostly this would be no big problem, since R comes from the Unix tradition where it is common to use many small re-usable tools[3]. Therefore it is possible to share functionality beyond the bounds of a software.

On the other hand, several packages try to deal with the problem of handling huge data in R in general. These approaches are interesting for remote sensing analysis, where one has to handle large images. Satellite data is large in sense of extent (number of pixels), as well as in sense of dimension (number of bands/attributes) and holds a huge set of spatially referencable information. If it would be possible to combine Rs excellence and convenience in processing "in situ" data sets with this kind of large image data, new fields in Rs (spatial) analysis could be established.

This paper presents the results of performance tests done with the `bigmemory` package and `ff` package.

### 1.2. Approaches and Solutions

On the one side one can use R as a toolbox from external software. The user can act quite flexible through that interface and still profit from R's capabilities. For an image analysis the user might switch to ILWIS [1], GRASS [7] or SAGA [13], and thus stay within the realms of free and open source software.

On the other side one can use existing R packages to perform an the analysis. There is for example `EBImage` as part of the Bioconductor [3] bundle, which uses ImageMagick for image processing. It is a good tool, but not intended to be used for georeferenced images. Other packages are namely `biOps`, `FITSio`, `rimage` and `rgdal` (this list might be not exhaustive). These packages actually use external software, and that is not what we are looking for.

A package which looks interesting and promising regarding to raster analysis within R is `Rgis` [8]. But at the time this paper is written it is still under development and no release has been published yet (Pre-Alpha).

### 1.3. Objective and motivation

This paper is made within the course *Geostatistics and Image Analysis* at the Institute for Geoinformatics (ifgi)[4] given by Edzer J. Pebesma. The topic is examined by two groups of students. One group deals with approaches to reduce the size of data, that is compression or resampling. These

[1] http://cran.r-project.org/
[2] http://www.gnu.org/copyleft/gpl.html
[3] These tools are available on other platforms as well using for example [11] or [10]

[4] http://ifgi.uni-muenster.de

bear the negative effect of information loss and require sequential processing with either a blockwise or column/row-wise method. For more information about this work please see [12].

We are interested in working with the language `R` itself. Since the authors' experience with `R` was only rudimentary before this work, we were inspired by an item on the list of possible topics in the beginning of the seminar. The technical focus and the chance to work on relatively new territory is interesting and challenging.

`R` is free software, and therefore worth to look at in general. While its capabilities in all varieties of statistical applications (econometrics, genetics, finance, . . . ) are well known for specialists in that area, a group of users started to implement image analysis methods for `R`. Moreover, there are libraries with a spatial or geostatistical emphasis. The quite early stage of these packages and a good community around `R` as a whole, would allow active participation—in case we reach good results.

Our group decided to further investigate the hardware oriented approaches, the packages `bigmemory` and `ff` in particular, because of their promising state at the time the paper is written. Based on these rather generic approaches we expect the possibility to hide non-typical data handling from the user, who can apply the existing file and data handling as well as image analysis methods without paying much attention to the file size.

Other methods on which a large dataset analysis could be based on but that are not taken into account are for example: distributed computing (`multiR`[5]) or databases (`RSQLite`[6]).

### 1.4. Overview of the paper

The remainder of the paper is structured as follows. The next part presents our work with `ff` and `bigmemory`. We describe the process combining these two packages with other packages that include image functionality and show the achieved results. Subsequently we discuss problems and assess the possibilities which the shown approaches give to apply `R` for image analysis. In the end we sum up our findings and point out directions for future work.

[5] http://e-science.lancs.ac.uk/multiR/

[6] http://cran.r-project.org/web/packages/RSQLite/index.html

## 2. The tested packages

The following versions were used.

- `R` version is 2.7.1
- `bigmemory` version is 2.3
- `ff` version is 2.0.0

### 2.1. `bigmemory`

The package was developed by Michael J. Kane and John W. Emerson. `bigmemory` uses compiled `C++` code to create, store, access and manipulate massive matrices. It offers an interface to manage these matrices in `R` and let the user work with them in "a usual way".

`bigmemory` also supports allocating the data to shared memory, so several `R` processes can process the data without keeping it redundant. "This opens the door for more parallel analysis and data mining of massive data sets" [9].

`bigmemory` keeps things simple to the user. The used datastructure is kept as `C++`-matrix in background and an ordinary pointer is returned to the user, which can be handled as ordinary `R` matrix. The interface is easy and similar to `R`s existing matrix functions. E. g. commands like `bm <- big.matrix(ncol=10,nrow=10)` and `dim(bm)` represents the same meaning as the better known functions like `m <- matrix(ncol=10,nrow=10)` and `dim(m)` respectively.

Supported types are `double`, `integer`, `short` and `char`. This makes `bigmemory` quite flexible for different kinds of data. `bigmemory` operates on RAM which makes it fast. Per contra it uses more memory while performing, but keeps matrices manageable. A next version of `bigmemory` will combine advantages of keeping things in RAM but also storing data in temporary fiels while performing.

### 2.2. `ff`

The `ff` package provides atomic data structures that are stored on disk but

> behave (almost) as if they were in RAM by transparently mapping only a section (pagesize) in main memory—the effective virtual memory consumption per `ff` object. `ff` supports atomic data types `double`, `logical`, `raw` and `integer` [...]. `ff` now has native `C`-support for vectors, matrices and arrays with flexible dimorder (major column-order, major row-order and generalizations for arrays). ff objects store raw data in binary flat files in native encoding, and complement this with metadata stored in `R` as physical and virtual attributes. [...] [4]

The authors provide a professional extension to their package that adds more data types. This explains the dual-licensing model for the package. The standart `R`- and `C`-code is licensed as GPL2[7], whereas the extended code is proprietary. `C++` code by Daniel Adler is licensed as ISC of free BSD[8]. Authors of the package are Daniel Adler, Christian Gläser, Oleg Nenadic, Jens Oehlschlägel and Walter Zucchini.

Of the addional data types, `byte` is interesting for image analysis, as most image data, meaning the pixels' values, is of that type. In our test the casting of bytes to `raw` served as a wrapper for testing the performance with byte data. According to a conversation with the authors, the behaviour should be very similar to using the actual `byte`-type of `ff`.

Objects can be copied to RAM and behave like normal `R` objects in that case. `ff`-objects can be observed in the temporal directory and persistency functions are provided. The matrix objects can be accessed using the same syntax than `R` matrices—similar to `bigmemory` mentioned above. `ff` used optimization techniques, for example a preprocessed index, for good performance on large datasets.

The authors of `ff` present a performance survey the made in [2]. Our test did not show the advantages shown in comparison with bigmemory, but some results are similar. `ff` has a notably overhead with small image data and the relative efficiency becomes better with bigger files. With notably large files (larger than RAM, $> 2$ GB) it is the only package that still is effektive, meaning that still can handle the data at all.

[7] `http://www.gnu.org/copyleft/gpl.html`
[8] `http://en.wikipedia.org/wiki/ISC_licence`

## 3. Data and Methods

Comparing `ff`s and `bigmemory`s performance requires the usage of the same methods and the same data sets. The functions of the test script (also called "scenario" in this paper) were hold quite generic. `ff` and `bigmemory` are working on their own specific matrices. The datastructures—once initialized—are handed over to each function as parameter.

### 3.1. Scenario

The script represents *one* generic sequence which can be assigned to one of both packages and one of both images to process. The workflow of the sequence is to

1. read image into data structure
2. calculate statistics on the image
3. perform matrix operations or a simple filter
4. write image back to file

### 3.2. Data

Satellite images are large data sets. Their size varies highly concerning to the level of processing they go through. Just after creation, a satellite image or an aerophoto may have a size of several GB. This is too big for practical application and unusual in normal use cases and therefore too big to be tested in this work. An image for spatial analysis may be have a size of about 600 MB. However, this would hardly be manageable, too, if such an image would not have the information stored seperately, each in its own band (e.g. RGB and several infrared channels in one band each).

For our analysis we chose two images into account: A satellite image of size 334 MB and a smaller image of about 10 MB. However, for practical reasons we only took one band of the satellite image (of size approx. 70 MB).

For a more detailed view, which data was used you may have a look at the appendix at 13 ff.

## 3.3. Data handling

`bigmemory` and `ff` perform on `C` matrices, so importing the image data into these datastructures is the first task at hand. To simplify reading the whole image into `R` and then writing it into the matrices afterwards, we used the package `rgdal`. `rgdal` lets the user take advantage of the Geospatial Data Abstraction Library (GDAL) within `R`.

`R` has a lot of memory pitfalls. With `rgdal` the user is able to act with a file pointer and subsets of the file of interest. There is no need to import the data into `R` as a whole. For satellite image data each band can be considered for its own.

However, working with one band with size of approximately 70 MB is still problematic in `R`. To load the image data into a `ff` or `bigmemory` matrix, one has to do it partitioned to avoid crashing `R`. We decided to initialize the datastructure first and then to load the image data row-by-row into it. That performed very well and didn't overrun 10 s for the 70 MB image. See the used import/export functions in listings 3 and 4. Both functions are written generically, so the same code is used for an `ff` matrix and a `bigmemory` matrix.

The packages support different data types and we mention some experiences later on in the text. It may however be pointed out, that a satellite band usually contains values from 0–255 (in each channel) and consequently a `byte` is sufficient to represent the information of a band. Data types that require more memory are limited by an overhead.

## 3.4. Operations

We implement some simple image analysis methods and one complex one by establishing kernel operations. The next sections describe them in more detail.

### 3.4.1. Simple operations

The simple operations have in common, that each performs on only one pixel. Similar to reading the image, the operations process the image row-by-row. See an example in listing 1.

Listing 1: Negate image values in a given matrix

```
negate <- function(ffOrBm) {
  ## row-by-row
  for (row in 1:nrow(ffOrBm)) {
    ffOrBm[row,] = abs(ffOrBm[row,] - 255)
  }
  return(ffOrBm)
}
```

Other algorithms are: incrementation of all pixels with a fixed value; global threshold for a single value determining foreground (value of 255) and background (value of 0) pixels. While these are classical cases in image preprocessing, they do only differ by few computations from each other and consequently have the same complexity.

### 3.4.2. Kernel operations

Kernel operations are the basis of a wide range of image analysis methods. If one kernel operation can be effectively established, a huge set of other kernels can easily be implemented (several types of filters, etc.).

We chose an Edge-sharpening kernel. It is a common and important tool in image preprocessing to enhance boundaries between different entities represented in the pixel values of an image. A common sharpening filter is the *Laplacian* filter[9].

We implemented two algorithms: First we implemented a naive brute force algorithm which takes each pixel with the corresponding kernel neighbors, calculates the new value for the pixel and writes it back. This resulted in a double loop to move the (3x3 kernel) over the whole image—not really efficient. In contrast to the other tests, we used a much smaller image (512x512 pixels) here. Otherwise it would have taken too long.

The other approach was to take advantage of `R`s matrix multiplication, which is implemented very efficient, and getting rid of having to touch

[9] For more information see [6]

each pixel explicitly in a loop. With the help of code supplied by Edzer J. Pebesma, we implemented a filter based on the + matrix operator. This implementation was unfortunately not very generic. Since `bigmemory` doesn't support matrix access with a negative index it could not be tested. Alternative ways for implementing such a filter may exist, but could unfortunately not be deeper looked into for this paper.

## 3.5. Test machine

Comparable results are accomplished by performing all tests on the same machine and on equal data sets. `R` is started from command line and other software that runs in parallel is kept to a minimum.

- 2,063 MB RAM, 2x Intel(R) Core(TM), 2 Duo CPU (T7250@2.00GHz)
- Ubuntu 8.10 Intrepid, 2.6.27-9-generic, i686 GNU/Linux

## 3.6. Measuring performance

The large size of image data makes time a factor of interest. If an image analysis (e. g. a simple filter operation) would take 5–10 minutes, no productive environment could be established. The time processing an image depends on several issues:

- efficiency of algorithms involved
- efficiency of data structures used
- memory access speed (RAM or ROM)

For the tests only memory usage in main memory and CPU time was considered and compared. We did not take into account that `ff` uses temporary files where data is stored. Because `bigmemory` doesn't use temporary files a comparison does not make sense[10].

The results are plotted in section 4.

[10]This will be a new feature of the next version of `bigmemory` and can be a topic for future evaluations.

### 3.6.1. Observing time

Standard Linux tools were used to trace the CPU time of a process. Using `top` of version 3.2.7 in batch mode is sufficient to observe particular attributes of a process while running. Listing 2 shows an example. The attribute `TIME+` shows the cumulative CPU time used by a process. The shown time of 0:00.80 means that `R` needed 800 ms to start.

For the scenario it was interesting to differentiate between several phases. `R` offers its own build-in function to measure this (with a small overhead). Using `system.time(<function>)` it was possible to distinguish between these phases:

1. initialize the datastructure
2. read/import the image data
3. perform some image statistics
4. perform image process
5. write image back to file

Results can be found in the appendix, p. 14 ff.

### 3.6.2. Observing memory

Also, `top` was used to observe memory usage of `R` when performing either with `bigmemory` or `ff`. For this, two attributes were of interest. Listing 2 shows several attributes referring to memory (taken from `man top`):

**VIRT** The total amount of virtual memory used by the task. It includes all code, data and shared libraries plus pages that have been swapped out.

**RES** The non-swapped physical memory a task has used.

**DATA** The amount of physical memory devoted to other than executable code.

Static memory usage were not of interest, so for the scenario only `DATA` was used. It shows clearly how much memory `R` must allocate when working with external data sets.

Listing 2: Memory usage (commandline)

```
henning@phoenix-work:~/workdir/R$ top -bp `pidof R` | grep henning
# PID USER       PR  NI  VIRT  RES SHR  S %CPU %MEM   TIME+    DATA COMMAND
15334 henning    20   0 23196  16m 2624 S    0  0.8  0:00.80   14m R
15334 henning    20   0 23196  16m 2624 S    0  0.8  0:00.80   14m R
15334 henning    20   0 23196  16m 2624 S    0  0.8  0:00.80   14m R
. . .
```

Listing 3: Read an image into datastructure

```
## read image into the given (pre-
## initialized) datastructure ffOrBm
ffOrBm.readImage <- function(img,ffOrBm
      ,band=1) {

  ## get extent of image
  rowlength = nrow(img)
  collength = ncol(img)

  ## read and set raster data row-by-row
  for (row in 1:rowlength) {
    ffOrBm[row,1:collength] <-
          getRasterData(img,
          band,region.dim=c(1,collength),
          offset=c(row-1,0))
  }
  ## release pointer and return ffOrBm
  GDAL.close(img)
  return(ffOrBm)
}
```

Listing 4: Write data back to file

```
## write image from the given data-
## structure ffOrBm back to image file
ffOrBm.writeImage <- function(ffOrBm,
      filename,driver,type="Byte") {

  ## create dataset where to write on
  image <- new("GDALTransientDataset",
        driver,nrow(ffOrBm),ncol(ffOrBm),
        type=type)

  ## write row-by-row
  for (row in 1:nrow(ffOrBm)) {
    putRasterData(image,ffOrBm[row,
          1:ncol(ffOrBm)],
          offset=c(row-1,0))
  }
  ## persist transient data
  saveDataset(image,filename)
  closeDataset(image)
}
```

## 4. Results

During the tests we measured the time, each package uses to perform a scenario on each of two images available (described in section 3.1). The following sections describing the time and the memory usage refer to scenarios which processed simple operations like negation and incrementation. Further operations going beyond these are described afterwards.

### 4.1. Time

The whole time both packages needed to perform on the small image differ not too much. The several phases which form a whole scenario are quite different, though.

A noteworthy point regards the use of the parameter dimorder in `ff`. As our algorithms perform from a row-wise processing of the data, but the default dimorder is column first, a big speedup could be reached early in our algorithm development phase. On initiating the `ff` object, the paramter setting `dimorder=c(2,1)` lets `ff` switch to a row-wise ordering.

Considerable differences can be noticed when performing on the large image. `ff` does a better performance here and is up to 30 s faster that `bigmemory`.

The tests of datatypes (see section A.3) result in a mentionable performance gain for `ff` of about 5 s for the used image. This can be explained by the smaller data size that has to be loaded from the hard drive. The `byte` data (casted to `raw`) takes 11 MB storage in contrast to 43 MB when using `int`. In opposition to that, `bigmemory` actually slows down if a supposedly less storage intensive data type is used. This might be caused by type casting overhead, but is not investigated further in this work.

Considering the several phases of the scenario, big differences between the statistics calculation are noticeable: `bigmemory` is very fast calculating the statistics of an image. Whereas `ff` needs more than 30 s, `bigmemory` is done with it within about 2 s (on the large image).

Nevertheless, `ff` works faster than `bigmemory` (except one case). That's quite surprising, since `ff` must read/write from files stored on harddisk. This bottleneck is payed off with a faster processing phase. The longer this phase endures, the greater the differences between `ff` and `bigmemory` will be.

### 4.2. Memory

The memory used by `ff` while processing an image varies very much. It's very dependant to a current phase of the scenario. Due to `ff` stores the image data in temporary files, only one row has to be processed (when doing a row-by-row algorithm). This is quite memory saving. Similar to the observed time, a memory overhead becomes obvious when `ff` calculates the image statistics. For that, the whole image must be loaded into memory.

`bigmemory` acts kind of greedy. Once, it has allocated the memory needed for the image it will not be de-allocated until the scenario stops and a garbage collection is performed. This behaviour leads to a quite constant memory usage, wherein no overheads can be observed—during these memory peaks `ff` uses nearly twice of the memory allocated by `bigmemory`.

### 4.3. Laplacian edge sharpening

During our tests we started to process the naive algorithm over the images. Appendix A.1 shows the resuls. Since the algorithm took much longer than an image analysis should perform (more than 5–10 minutes), we decided to perform it on a smaller image (512x512 pixels) to see, how the packages behave. The naive algorithm is definitely not fit for production use.

To get rid of touching each pixel explicitly, we took advantage of `R`s matrix multiplication in a second approach. With the help of Edzer J. Pebesma, we implemented a filter based on an efficient implementation of the + matrix operator. Unfortunately `bigmemory` wasn't able to handle negative indexes, so only `ff` could offer results here. These weren't too bad and kept far below the 5 minute limit, but took up to 1,5 GB RAM during runtime. See the results in figure 3 on page 10.

### 4.4. Comparison to `EBImage` (exemplarily)

To get a comparison to an existing approatch dealing with image data in `R` we've chosen `EBImage`. `EBImage` belongs to the Bioconductor package bundle. Whereas `EBImage` offers broad functionality doing image analysis it unfortunately lacks of supporting image analysis with a spatial context.

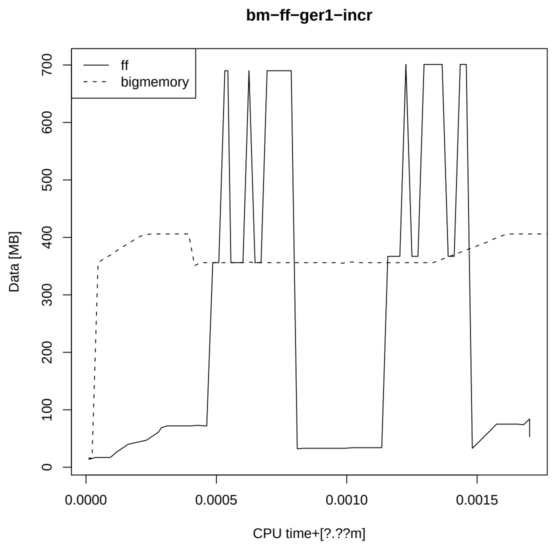
Listing 5: `R` code using `EBImage`.

```
> system.time(write(negate(readImage(
    "DEMO_ETHIOPIA_1_IMAGES_SWIR_1.tif")),
    "negate.tif"))
  user   system  elapsed
54.240   70.301  125.604
```

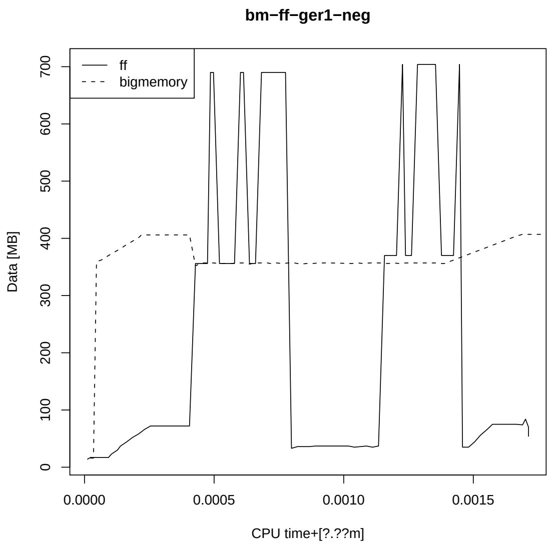
Since `.bsq` files are not supported by `EBImage`, we used the `.tif` file (size 10 MB) to perform a part of the workflow scenario (referred to example from [12]): The workflow was to read, to negate and to write the image back into a new file (see listing 5).

In comparison to our regular workflow, both `ff` and `bigmemory` were enormously faster (including statistics calculation!). In addition to this `EBImage` needed much more memory while performing (up to 440 MB instead of the 120 MB used by `ff`).

This results may arise from the fact, that the extra export/import of an image to exernal software brings out more performance loss than could be payed off from the rapidness of the external software.

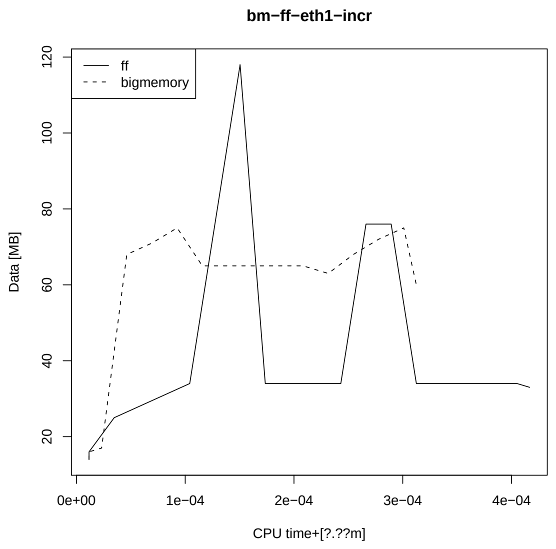


(a) CPU time from 0.000–165.554 s.

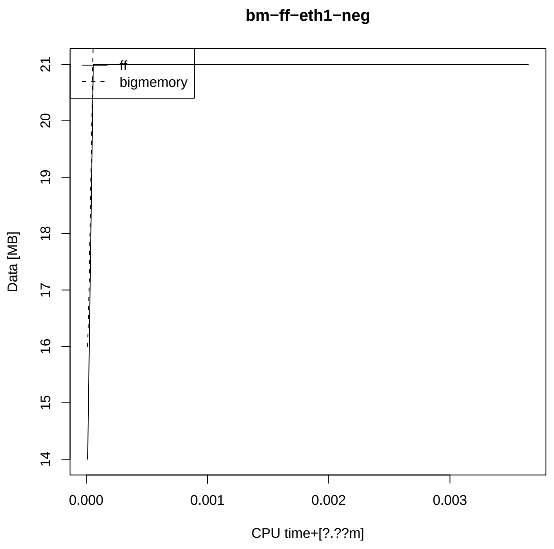


(b) CPU time from 0.000–169.798 s.

Figure 1: Calculating the 70 MB file.



(a) CPU time from 0.000–28.850 s.



(b) CPU time from 0.000–26.349 s.

Figure 2: Calculating the 10 MB file.

|  | user | system | elapsed |
|---|---|---|---|
| Init | 5.236 | 0.996 | 8.284 |
| Read | 26.734 | 0.664 | 31.328 |
| Summary | 20.265 | 4.996 | 25.373 |
| Process | 71.136 | 19.905 | 338.189 |
| Write | 19.045 | 1.504 | 25.040 |
|  |  |  | 428.214 |

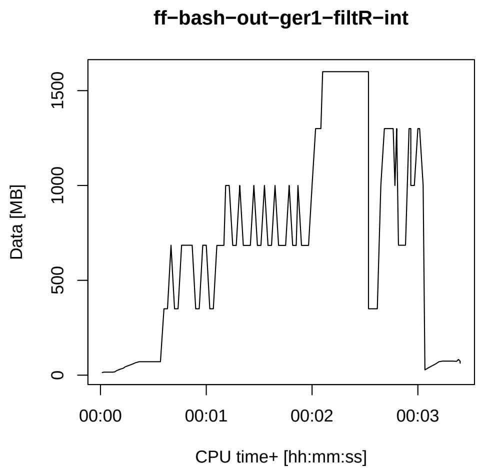


Figure 3: Laplacian filter running on ff (70 MB file)

## 5. Discussion

Quickly, the user can do lots of things wrong when he tries to load large images into R in a naive way. At the beginning of our research we also stepped into several of those pitfalls. To make R even handle large data like satellite images, we have tried the packages ff and bigmemory which can hold these kind of data in external C-matrices by giving the user R methods at hand to manipulate them. Besides the similar way of storing the data in external matrices, both packages handle the data quite different. The data plots highlight these different concepts of data handling very clearly.

Under the given circumstances, ff seems to hold back some of its potential to us. The presentation "A first glimpse into 'R.ff'" [2] mentions functions that could be of high value when working with images, namely fftile (tiling mechanisms) and ffbatch (batch processing). While image analysis would certainly gain from implementing more standard R functions for ff, especially the limited data types in the free ff versions, the *vmodes*, partly put brakes on the performance in our scenario.

It is surprising why bigmemory has performed less fast than ff did in general. By looking at the phases in particular, bigmemory does its job very well when calculating statistics but falls back when performing the image process. Both packages were tested under the same premises and due to bigmemory doesn't has to pass a bottleneck ff has to (like harddisk access), so we assume the access mechanisms of bigmemory aren't as well as in ff (ff utilizes indexes).

For ff the size of data will not be a problem. Since writing data to harddisk, the limit of a file would be theoretically given by the partition type used (in our case 2 Tebibyte on ext3). For now bigmemory holds everything in RAM, so it is limited to RAM size and swapping capabilities of the operating system. The new version of bigmemory looks more promising. It will also be able to store data as temporal files.

We could measure a good performance on initialization of the datastructures and the read/ write phases. Also the duration of simple processings, to negate and to increment, are acceptible.

Our method is flexible regarding the input and output data, which is based on the utilization of rgdal, that handles many formats. If georeferencing information is given with meta files, like in our example, a simple renaming and copying operation can preserve it. If these kind of information are tagged within the image file (e. g. geotif) one has to consider to extract this information and hold it besides the processing. After image analysis, it can be written back to a file. For operations which change the size, the orientation, etc. one has to bear in mind that the georeference information "backup" will not be taken into account.

## 6. Conclusions and future work

We began this work with an evaluation of existing packages in the general area of image analysis. In parallel we evaluated different approaches to the memory problem in R. We tried to combine these two areas to be able to use R for image analysis and lay the basis for remote sensing with large image data.

Is our approach better than the existing tools? If effectivity is the indicator, then our answer is yes. We show a way of handling of large image data exists, but its practicality still has to be proven.

A very comfortable program could handle the adapted handling of the data and the analysis invisible to the user. She just gives the original data source, the method to perfrom on that data and a vector of parameters to a function. We did not achieve a transient combination with existing image analysis packages. As to our current knowledge, the algorithms (most of which are probably well-known nowadays) would have been to implemented newly on top of the presented data structures.

Our simple yet powerful algorithm for image analysis in the spatial domain (admittedly mostly applicable in the area of preprocessing) provides a basis for advanced methods like morphological image processing [6]. It lies in the nature of pixel-oriented processing that every pixel and his neighborhood is visited, and so a considerable amount of processing can be expected.

Providing a comfortable interface to these methods, like higher level functions for dilation or erosion, was not the scope of this work. An implementation of these methods, that include a customized optimization to the underlying implementation, would provide an easily usable package for a wider audience. To achieve that level of applicability, the quality of the methods needs to be raised regarding special cases. For example filter effects at the image borders that are currently not treated can be avoided through padding the image.

Moreover automized methods that choose an appropriate technique to handle the data depending on its size and the operation are conceivable. The advanced user can take into account by himself that if the data is small enough for R to load without problems, then that approach is logically the best way to go.

Open questions that arise at the end of our work are: How do the presented methods scale, especially with several bands (multivariate analysis)? How can image data with its inherit channel structure be stored more efficiently in our storage systems? Another approach towards the used data type might also be a good enhancement, for example storing bands in a special data structure as raw types.

As far as the effeciency for a general use in image analysis is concerned, the R-based methods have to compete with widely accepted specialized tools, e. g. MATLAB® [16]. The amount work that is necessary to include external tools into R or vice versa should be weighted critically against a completely R-based approach. This applies even more to advanced image analysis methods like feature extraction and pattern or object recognition, whose algorithms most definetely can be implemented in R.

We imagine that further investigation in how Rs statistical features can be transferred to images will show the value of imagine analysis performed within a R environment—the actual processing is possible and R can provide more than just a convenient interface.

## References

[1] 52° North. *ILWIS 3.5 Open*. 2009. URL: http://52north.org/.

[2] Nenadic Zucchini Adler Oehlschlaegel. "Large atomic data in R: package 'ff'". In: 2008.

[3] Bioconductor Development Core Team. *Bioconductor, Open Source software for bioinformatics*. 2009. URL: http://www.bioconductor.org/packages/release/bioc/.

[4] Oleg Nenadic Jens Oehlschlägel Walter Zucchini Daniel Adler Christian Gläser. *CRAN – Package ff*. 2008. URL: http://cran.r-project.org/web/packages/ff/index.html.

[5] R Development Core Team. *What is R – Introduction to R*. 2009. URL: http://www.r-project.org/about.html.

[6] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2008. ISBN: 013168728X.

[7] GRASS Development Team. *Geographic Resources Analysis Support System*. 2009. URL: http://grass.itc.it/.

[8] Jacob van Etten, Robert Hijmans, Yann Chemin, Sonia Asilo. *r-forge – Package r-gis*. 2009. URL: http://r-forge.r-project.org/projects/r-gis/.

[9] Michael Kane Jay Emerson. "The R Package bigmemory: Supporting Efficient Computation and Concurrent Programming with Large Data Sets". In: 2008.

[10] MinGW Development Team. *MinGW Home*. 2009. URL: http://www.mingw.org/.

[11] Red Hat Cygwin Development Team. *Cygwin Home*. 2009. URL: http://www.cygwin.com/.

[12] Richard Redweik and Johannes Trame. "Handling large images in R". Seminar presentation. 2009.

[13] SAGA Development Team. *SAGA, System for Automated Geoscientific Anaylis*. 2009. URL: http://www.saga-gis.org.

[14] Phil Spector. *Data Manipulation with R*. ISBN 978-0-387-74730-9. New York: Springer, 2008.

[15] R Development Core Team. *R Data Import/Export*. 2006. URL: http://cran.r-project.org/doc/manuals/R-data.html.

[16] Inc. The MathWorks. *MATLAB®*. URL: http://www.mathworks.com/products/matlab/.

## A. Appendix

### A.1. Image data

We've used the following data files as input for our analysis:

#### A.1.1. Satellite scene somewhere in Germany

- `http://image2000.jrc.ec.europa.eu/`
- File `de33_194025.bsq`
- `.bsq` filesize 334 MB
- Comes with a bunch other (small) meta files

  **bqw** Worldfile

  **bsq** Data (strict seperation of the spectral bands)

  **hdr** `.bsq` specific information (nrows, ncols, pixeltype, etc.)

  **mta** Image 2000 Meta file

  **prj** Projection in WKT
- Each band has size of ca. 68 MB

#### A.1.2. Satellite scene somewhere in Ethiopia

- `http://www.orthocoverage.com/download/`
- File `DEMO_ETHIOPIA_1_IMAGES_SWIR_1.tif`
- `.tif` filesize 11 MB
- Comes with a two meta files

  **tfw** Worldfile

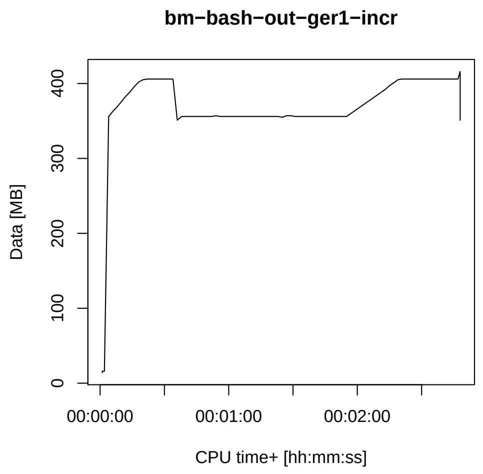
  **prj** Projection in WKT
- one band

## A.2. Simple operations

### A.2.1. `de33_194025.bsq`

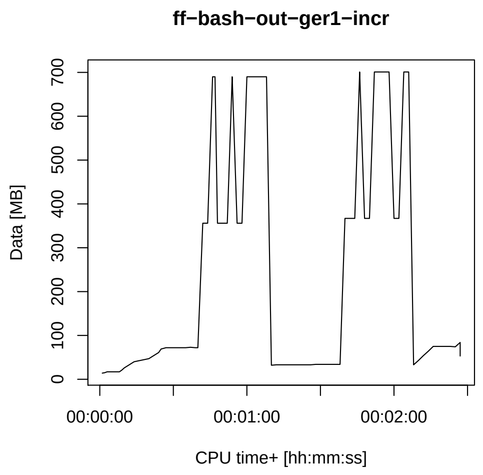
#### Increment with `bigmemory`

| | user | system | elapsed |
|---|---|---|---|
| Init | 0.180 | 0.352 | 0.533 |
| Read | 31.110 | 0.356 | 31.706 |
| Summary | 2.168 | 0.000 | 2.170 |
| Process | 74.752 | 0.316 | 75.676 |
| Write | 52.339 | 1.376 | 56.002 |
| | | | 165.554 |



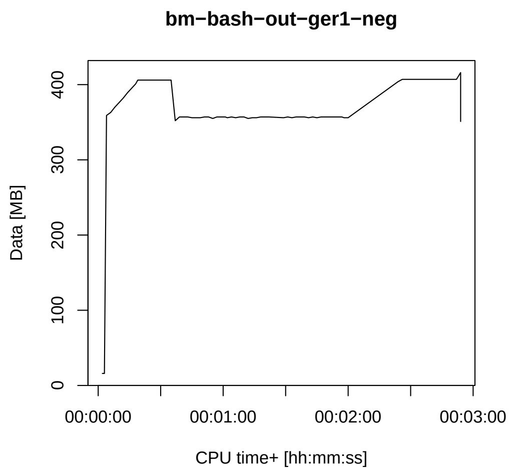
#### Increment with `ff`

| | user | system | elapsed |
|---|---|---|---|
| Init | 5.436 | 0.944 | 9.491 |
| Read | 27.106 | 0.820 | 31.144 |
| Summary | 21.217 | 6.092 | 33.548 |
| Process | 30.290 | 0.644 | 31.328 |
| Write | 19.733 | 1.524 | 22.784 |
| | | | 128.295 |



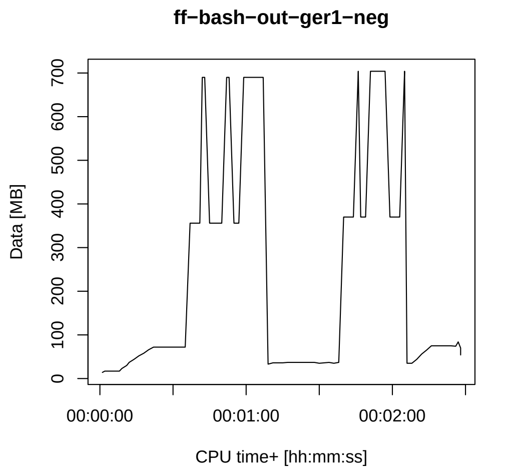
## Negate with `bigmemory`

| | user | system | elapsed |
|---|---|---|---|
| Init | 0.168 | 0.356 | 0.526 |
| Read | 31.642 | 0.416 | 33.669 |
| Summary | 2.164 | 0.004 | 2.183 |
| Process | 75.625 | 0.664 | 76.728 |
| Write | 52.299 | 1.344 | 56.692 |
| | | | 169.798 |



## Negate with `ff`

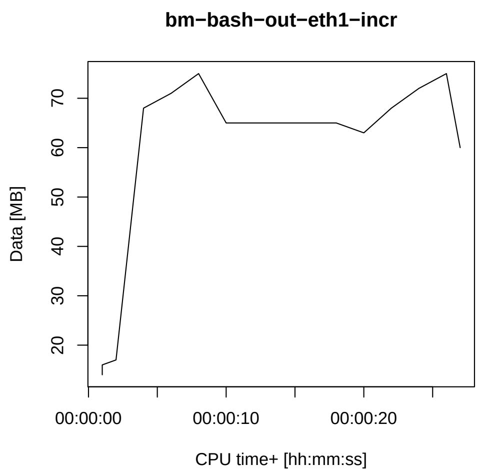
| | user | system | elapsed |
|---|---|---|---|
| Init | 5.352 | 0.992 | 8.724 |
| Read | 27.558 | 0.688 | 30.817 |
| Summary | 23.869 | 7.452 | 41.444 |
| Process | 30.682 | 0.772 | 32.560 |
| Write | 20.018 | 1.464 | 25.270 |
| | | | 138.815 |



### A.2.2. `DEMO_ETHIOPIA_1_IMAGES_SWIR_1.tif`

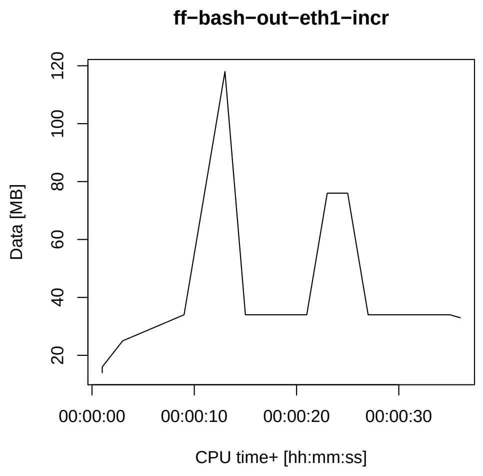
#### Increment with `bigmemory`

| | user | system | elapsed |
|---|---|---|---|
| Init | 0.040 | 0.064 | 0.102 |
| Read | 5.856 | 0.044 | 5.970 |
| Summary | 0.284 | 0.004 | 0.290 |
| Process | 10.641 | 0.040 | 10.718 |
| Write | 7.436 | 0.192 | 7.773 |
| | | | 28.850 |



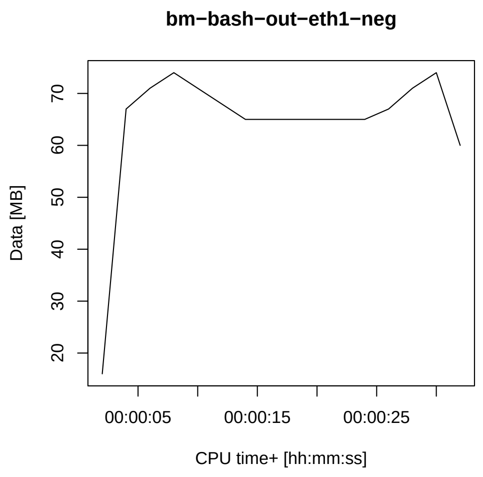
#### Increment with `ff`

| | user | system | elapsed |
|---|---|---|---|
| Init | 0.596 | 0.068 | 0.680 |
| Read | 7.036 | 0.136 | 7.329 |
| Summary | 2.804 | 0.660 | 3.528 |
| Process | 8.593 | 0.076 | 8.748 |
| Write | 5.984 | 0.180 | 6.215 |
| | | | 26.500 |



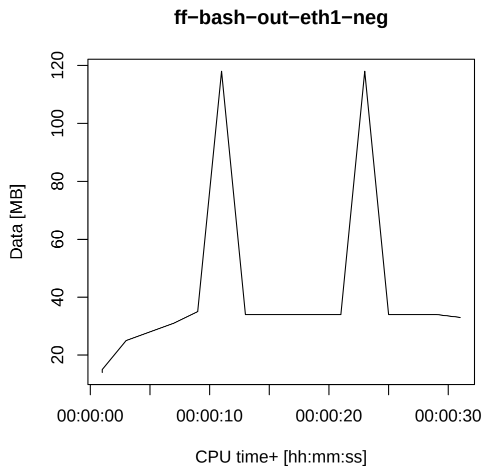
## Negate with `bigmemory`

| | user | system | elapsed |
|---|---|---|---|
| Init | 0.016 | 0.056 | 0.070 |
| Read | 5.824 | 0.024 | 5.874 |
| Summary | 0.384 | 0.000 | 0.384 |
| Process | 10.813 | 0.024 | 10.895 |
| Write | 7.368 | 0.144 | 7.573 |
| | | | 24.796 |



## Negate with `ff`

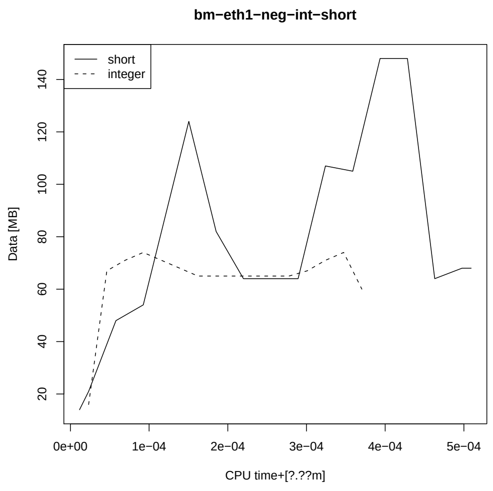
| | user | system | elapsed |
|---|---|---|---|
| Init | 0.656 | 0.048 | 0.742 |
| Read | 6.676 | 0.124 | 6.885 |
| Summary | 2.776 | 0.604 | 3.388 |
| Process | 8.593 | 0.088 | 8.729 |
| Write | 5.908 | 0.176 | 6.605 |
| | | | 26.349 |



## A.3. Comparing different datatypes on `DEMO_ETHIOPIA_1_IMAGES_SWIR_1.tif`

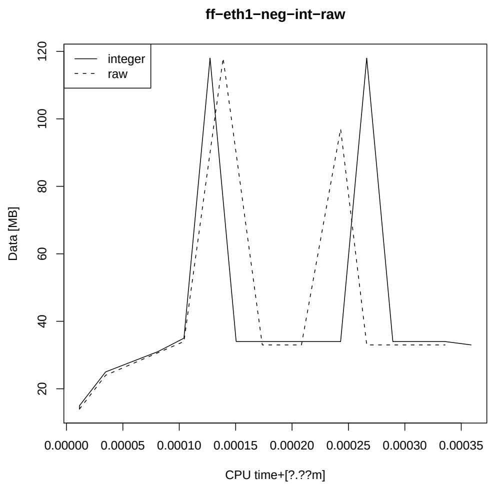
**`bigmemory` used with `integer` and `short`** Table shows data collected for `short`.

| | user | system | elapsed |
|---|---|---|---|
| Init | 0.028 | 0.016 | 0.045 |
| Read | 5.828 | 0.040 | 5.896 |
| Summary | 9.249 | 0.388 | 9.679 |
| Process | 10.044 | 0.036 | 5.896 |
| Write | 7.100 | 0.100 | 7.216 |
| | | | 28.732 |



**`ff` used with `integer` and `raw`** Table shows data collected for `raw`.

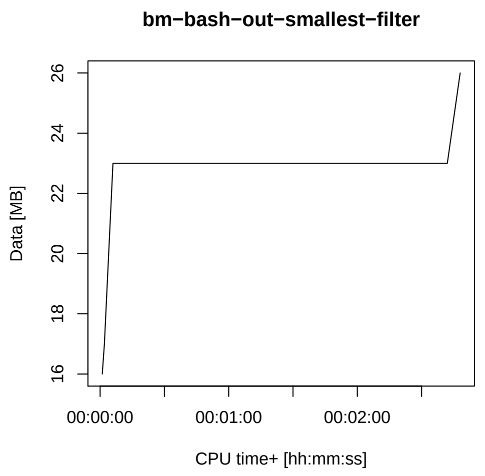
| | user | system | elapsed |
|---|---|---|---|
| Init | 0.632 | 0.016 | 0.655 |
| Read | 6.444 | 0.028 | 6.505 |
| Summary | 2.840 | 0.280 | 3.146 |
| Process | 7.961 | 0.040 | 8.138 |
| Write | 5.584 | 0.116 | 5.761 |
| | | | 24.205 |



## B. Using a filter kernel

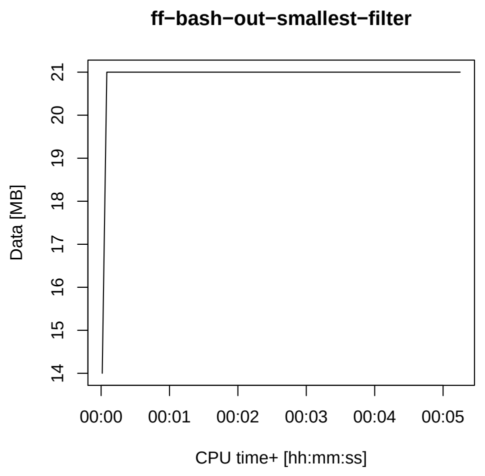
### B.1. `bigmemory`

|  | user | system | elapsed |
|---|---|---|---|
| Init | 0.004 | 0.000 | 0.005 |
| Summary | 0.256 | 0.024 | 0.287 |
| Read | 0.616 | 0.016 | 0.634 |
| Process | 159.218 | 0.084 | 160.076 |
| Write | 0.624 | 0.000 | 0.638 |
|  |  |  | 161.640 |



### B.2. `ff`

|  | user | system | elapsed |
|---|---|---|---|
| Init | 0.024 | 0.000 | 0.023 |
| Summary | 0.104 | 0.012 | 0.119 |
| Read | 0.796 | 0.016 | 0.821 |
| Process | 311.383 | 0.084 | 312.709 |
| Write | 0.864 | 0.000 | 0.866 |
|  |  |  | 314.538 |


ff−bash−out−smallest−filter
Data [MB]
00:00
00:01
00:02
00:03
00:04
00:05
CPU time+ [hh:mm:ss]