RMINC Module **mincIO**

Minc Volume Access Without Tears

Version 1.0

Jim Nikelski
Lady Davis Institute for Medical Research
McGill University
Montréal, Quebec, Canada [1]

Drafted on July 21, 2010

[1]Corresponding author. *E-mail address:* nikelski@bic.mni.mcgill.ca

I remember my first years at the Montréal Neurological Institute (MNI) Brain Imaging Centre (BIC). Those were very heady days, with various researchers, post-docs, and bioinformatics students creating the tools destined to be core of BIC development for years to come (15 years so far, and counting). My first project was a PET study in which the primary analyses were carried out by a very nice gui-interfaced tool, which permitted me to enter all of my subject and scanning condition details, and it would crunch the numbers and eventually produce some very lovely results.[1] Now, although `DOT` did all the the heavy lifting, I still found myself wanting to perform some additional processing, such as, plotting normalized blood flow over all subjects at some specific voxel, or computing group-level variance, or applying a mask to the t-statistic volume and visualizing a specific slice of interest. Although some of this work was possible using the standard command-line MINC tools, I was able to accomplish this, and a lot more, with a fairly newly developed Matlab toolbox called EMMA. And thus began my introduction to rapid application development (RAD) within a brain imaging context, as well as my love-hate relationship with EMMA.

Rapid Application Development (RAD)

Essential to RAD are the language chosen in which to do the development and the development environment. The development environment may involve an all encompassing integrated development environment (IDE), or may simply provide syntax highlighting and the ability to interactively submit snippets of code — this would seem to be a highly subjective decision. The language will often consist of a 4th generation language that does not compile down to machine code. In addition, I would suggest that in an area of research in which the researchers themselves (or their research assistants) would like the ability to do some degree of RAD for themselves, the language needs to be powerful, while retaining an intuitive syntax.

[1]For those interested, the analysis tool was called "DOT", and was the brainchild of the talented Sylvain Milot.

In the sections to follow, I will briefly review not only some RAD language options within the context of developing brain imaging software, but also the currently dominant volume formats, as both language and volume format need to align in order to offer a viable brain imaging RAD option.

Effect of Volume Format

This is not my area of expertise, but I'll make a few relevant observations anyway. To begin, I note 3 major volume formats:

Analyze/Nifti-1. I don't really want to get involved in the volume format wars, so I'll keep my comments polite. The Analyze format was a very simple format containing an ASCII header (.hdr) file and a binary image (.img) file. My grandmother would be very comfortable with this format — as a matter of fact, my grandmother could have designed this format. My dog would have done a better job. After numerous revisions in which very little changed, the powers that be (PTB) decided to use this format as the basis of the new and improved Nifti-1 format. Happily, Nifti-1 *does* correct some of the major drawbacks of Analyze, but for reasons of backwards compatibility, not nearly enough. There's a story about a farmer who put lipstick on a pig . . . which seems to be oddly relevant here.

AFNI .BRIK/.HEAD Originally developed at the Medical College of Wisconsin by Robert Cox, this volume format was (and still is, I believe) the native format used by the rather good AFNI image analysis software. The AFNI project is currently developed by the NIH (`http://afni.nimh.nih.gov/afni`). Although separate header and image files are used (like Analyze), this format is quite different. Storage of the image data as a separate file has the advantage of permitting the image data to be memory mapped; sadly, along with this advantage comes a number of disadvantages, such as binary incompatibility of image data across platforms (the endianess problem).

MINC MINC was originally written by Peter Neelin in 1992 (see `http://en.wikibooks.org/wiki/MINC/History` for his reasons for developing the MINC format). The original version of MINC was built on top of the NetCDF container format (`http://www.unidata.ucar.edu/software/netcdf`), and extended for use in medical imaging environments. The first major revision of the MINC volume format, resulted in changing the underlying container from NetCDF to HDF5 (`http://www.hdfgroup.org/HDF5`), yielding the MINC2 volume format. Unlike some of the previously mentioned formats, MINC combines image and header meta-data in the same volume. Due to the complexity of this volume structure, access to data and meta-data is usually through an application programmer interface (API), which has presented some rather serious barriers to entry for researchers who just wanted to start hacking at their data. Three APIs are currently available for MINC volumes: (1) the original MINC1 API, (2) the volumeIO API, developed to present a simpler, gentler interface, and (3) the MINC2 API, developed to permit relatively easy access to the new MINC2 volumes (only).

Personally, while I really like the MINC volume format (surprised?), I can see the allure of the other formats. Specifically, the design decision to separate image and header data, permitted (1) the creation of an extremely simple header format which was programmatically very easy to access, and (2) the memory mapping of the image data. The simplicity of this design allowed pretty much anyone to write code to access these volumes. This ease of use feature has historically not been a feature of the MINC format, and, I would suggest, has really been a huge impediment to the wider adoption of the the MINC format. So the question now is, can something be done to make the MINC volume format *much more accessible*, and thus potentially expand its use in the imaging community? Read on — yes, the answer is in the next section.

Effect of Development Language

As mentioned previously, choice of language is crucial for RAD development in that it must be very easy to use, preferably multi-platform, and it must permit easy and intuitive access to the volume format of choice. Clearly, here at the MNI, our volume format of choice is MINC. So let's see how some of the language options stack up.

C/C++ OK, C/C++ is generally *not* thought of as a RAD language, however, it should be noted that multi-platform toolkits like QT (`http://qt.nokia.com/products`) and powerful IDEs (e. g., Eclipse, QT Creator, etc.) almost seem to make developing in C++ just as easy as writing in a scripting language. In addition, all volume formats are perfectly accessible from C/C++. Having said that, unless the PTB at the BIC make an executive decision to move to QT, C/C++ is not really an option as an omnipresent RAD language.

Python Python is a fine multi-platform language, that has already seen action in the brain imaging world. Specifically, Jonathan Taylor converted fmriStat to run under Python, which was able to read MINC1 volumes without difficulty. I believe that the project is still under development under the name of "Neuroimaging in Python" (NIPY) (`http://neuroimaging.scipy.org`), however, I do not believe that it is capable of reading MINC2 volumes. Sadly, there appears to be a strong anti-Python movement at the BIC (or perhaps a pro-Perl lethargy), so it's acceptance as the primary RAD language is unlikely.

Matlab Matlab (`http://www.mathworks.com`) is the 800 pound gorilla in this area,[2] primarily due to its use in the Statistical Parametric Mapping (SPM) toolbox (`http://www.fil.ion.ucl.ac.uk/spm/`) developed at the Wellcome Trust Centre for Neuroimaging at University College London (UCL). The primary advantage of using Matlab for imaging work

[2]According to Wikipedia, gorillas range in weight from 310–440 pounds. As such, an 800 pound gorilla would be really, really huge — which makes this an appropriate and enlightening metaphor.

is that, due to its focus on linear algebra, it is particularly well suited to the manipulation of large matrices and arrays — the sorts of data structures that we in the imaging community see a lot of. Additionally, the people of the SPM project (and others) have shown that Matlab can, indeed, do the job — although being capable of something does not imply that it's the best tool for the job. It's primary disadvantages are twofold: (1) the Matlab language, while extremely well suited for linear algebra, is an extremely weak general purpose language — when compared to other scripting languages such as Perl, Python, or Ruby, the Matlab syntax is remarkably limited, and (2) of all of the above languages mentioned, this is the only one for which one needs to pay a rather hefty licence fee. While opensource alternatives exist for Matlab (`http://www.gnu.org/software/octave`), they do not address the issue that Matlab is simply not a particularly good general purpose language.

As mentioned in the introduction, the EMMA toolbox is currently (and has been for some time) the RAD tool of choice. EMMA (Extensible MATLAB Medical Analysis) is a Matlab toolbox written at the MNI BIC in the mid-1990's by Mark Wolforth, Greg Ward, and Sean Marrett, and even now serves as the primary Minc IO layer for new development, including Keith Worsley's SurfStat analysis toolbox (`http://www.stat.uchicago.edu/~worsley/surfstat/`). Unfortunately, EMMA is not only not in active development, rather its current status is more like being on life support.[3] As such, now would be a very good time to come up with an open-source RAD language that would permit easy access to MINC volumes, serve as a foundation for new development, and take over from the increasingly irrelevant, drooling, and sputtering EMMA.

[3]Thanks, Claude!.

The R Development Environment

The R development environment (`http://www.r-project.org`) shares many similarities with Matlab, in that (1) development generally occurs in an interactive environment, (2) users write scripts using a built-in scripting language, (3) user code is processed by an interpreter (not compiled), (4) the project is under active development, (5) the language can be extended via user-written *packages* (similar to Matlab *toolboxes*), and (6) generated code is multi-platform, and will run on all *nix systems, as well as Windows.

While there are many commonalities, the differences are significant. Firstly, R is open-source software, released under the GNU General Public Licence (GPL). As such, the entire project's source-code is always available for download and inspection, and the software is "free" in the sense that the user is free to modify the source code, as well as in the financial sense, i.e., the software is free of charge. Secondly, R incorporates a powerful, general purpose, functional programming language, providing the user with the power and flexibility comparable to any number of scripting languages. Finally, while Matlab is focused on linear algebra, the primary focus of the R community is on statistical processing, with contributions being made by statisticians across the globe.

Given the above-mentioned benefits, R would appear to be a strong contender for the role of Matlab/EMMA replacement — if, an R equivalent of EMMA could be written, permitting developers writing in R full read/write access to MINC volumes. In fact, work by Jason Lerch on the RMINC package provided precisely this type of functionality. Specifically, although the primary focus of Jason's RMINC package was to implement various sorts of volumetric statistics (e.g., voxel-based morphometry, deformation-based morphometry), this could only be accomplished if the package were able to read and write MINC volumes. This ability was implemented within the RMINC package by calling MINC2 API routines via a shared library using R's .C/.Call call mechanism. Unfortunately, as the primary purpose of this package was volumetric statistics, the MINC IO functions were necessarily somewhat limited, and embedded into the RMINC package, thus mixing analysis code with MINC IO functionality.

In summary, work on the RMINC package provided support for the contention that the opensource R project could be extended to provide MINC volume IO functionality, thus opening the possibility that an extended R environment (via a specialized MINC IO package) could provide us with a viable alternative to the older Matlab/EMMA model.

The **RMINC-mincIO** Module

As you may have guessed from the title of this document, it was indeed possible to expand on the work done by Jason Lerch with the RMINC package, and that thing is called mincIO. mincIO differs from and extends the original RMINC in a number of ways. Firstly, as suggested by the module name, mincIO deals exclusively with MINC IO functions. No other non-IO-related stuff is included, and it never will be. Other packages that require mincIO functionality can simply list RMINC as a package dependancy, causing the mincIO functions to load immediately prior to the dependent package loading.

mincIO also differs from RMINC in that mincIO makes use of R's S4 classes instead of the S3 classes used by base RMINC. Now this is a rather technical issue, so let it suffice to say that S4 classes permit us to define fairly classical (more or less) classes, such as one might use in C++. All of the various mincIO S4 classes can be easily instantiated and used. It's probably a good thing to get a fairly clear sense of what these objects are, and how they interact. A quick peruse of the RMINC-mincIO reference guide or online help should provide a quick overview.

This reminds me that every function, method, and class that comprises the mincIO package has been documented and is available via the online R help. The R help system can usually be easily accessed via menus (if you're using a version of R with a graphical user interface), else typing `help.start()` at the R command-line should open a browser page. From the browser page clicking on `packages` and then RMINC, should take you to the online help. In addition, all of the mincIO help pages should also be available in pdf form, so get a hold of them and enjoy countless hours of bathroom reading.

Using RMINC-mincIO

This section of the document serves as a tutorial, packed with examples of mincIO usage. Before we get on with the tutorial goodness, first a bit of implementation background.

An Overview. As was the case with RMINC, mincIO is built using the MINC2 API. What does this mean? Basically, it means that whenever mincIO accesses a MINC2 volume, mincIO repackages the passed call arguments and then passes off control to a shared library (coded in C), which does a bit more processing, calling the MINC2 API, as needed, and then returns control. The calls to the MINC2 library routines require that your system have the BIC MINC tools installed, specifically, the MINC2 library needs to be build as a dynamic library, in order to permit real-time loading of the MINC2 API routines into R's address space. So, make sure that MINC is installed on your system, and that MINC2 is built as a dynamic library. On a related note, the MINC command-line tools should also be installed and available on your `PATH`, since mincIO occasionally forks off some calls to these tools (e.g., `mincconvert` for conversion to MINC2, `mincinfo` to grab some volume attributes).

mincIO does *not* do Windows©. Now, the previous paragraph should make clear that, given that MINC needs to be installed, and given that the MINC tools are not really supported on the Windows platform, running mincIO under Windows is not really an option. This is not an R restriction, as R is multi-platform, but rather a MINC restriction. As such, mincIO has not been tested under Windows, has never compiled under Windows, and is therefore not supported under Windows. Of course, I would very much like mincIO to run under Windows, and I'm entirely open to adding code to the package that would make this possible. Volunteers?

The MINC2 API. This API was created in order to present the user[4] with a simpler and more convenient API to MINC volume access than was previously available via the original MINC API. In addition, unlike the original MINC API that was built on top of the NetCDF API, MINC2 was built using the HDF5 API. The consequence of this decision is that the MINC2 API is not capable of reading the older NetCDF-based (MINC1) volumes. Happily, the `mincconvert` MINC utility does a fabulous job of losslessly converting MINC1 to MINC2 volumes, and the mincIO package does this conversion automagically whenever it detects a MINC1 volume. Of course, this conversion does take some time, and so mincIO spits out a warning message to let you know that it's doing a conversion. Clearly, if you know that you're using MINC1 volumes, it would probably be a good idea to run `mincconvert` yourself prior to running your R code.[5]

MINC Environmental Variables. Also related to MINC2 volumes, please be aware of the compression settings, which are set via various environmental variables. While the BIC recommends using moderate compression, I've found (running on a 2.2 GHz Core 2 Duo MacBook Pro with 4GB of RAM) a devastating performance impact. This is not related to R, as the impact can be seen in other programs, such as `postf`, in which opening a relatively high-resolution 4D volume with moderate compression takes (literally) 10 times longer. So, if you need performance, do not use internal compression or chunking. I use the following settings:

```
MINC_FORCE_V2=1               # all new volumes are MINC2
MINC_COMPRESS=0               # no compression
MINC_CHUNKING=0               # no chunking
VOLUME_CACHE_THRESHOLD=-1     # no caching badness
```

[4]The "user" being a C programmer.

[5]You can tell whether you're dealing with MINC1 or MINC2 by using the `file` command-line tool. For example, type "`file myMincVolume.mnc`", and if the returned line contains "NetCDF", then it's MINC1; if it contains "HDF5", then it's MINC2. Note that the Civet pipeline currently produces MINC2 volumes (a very good thing).

What is *Sweave*? And finally, before we continue with examples, we need to discuss the *formatting* used in the following examples. This user guide was generated using the Sweave system (`http://en.wikipedia.org/wiki/Sweave`), which permits embedding of live R code into a document. Embedding code directly within a document has the advantage that the actual R code contained within the document is actually executed within R every time the document is regenerated, thus ensuring that the code within the document was correct and executable at the time that the document was generated. Neat, yes? When reading Sweave output, the lines that show "user" input are always preceded with a greater-than sign (">"), whereas the lines generated by the R interpreter are not; all code lines are shown using `a typewritter font`. That's Sweave.

Great. With that said, we now move on with the tutorial. We'll start by briefly outlining building and installing RMINC, then we'll demonstrate how one reads volume header information and explain how to extract information from S4 mincIO objects. In subsequent sections, we'll move on to explaining the 3 levels of IO granularity, and how these are used. Note that, while we will necessarily be using R code, this is *not* an R tutorial. If you need help with R, you will find an almost infinite number of introductory R documents on the net, given that an ever increasing number of stats instructors are using R as their statistics software of choice. Google is your friend.[6] Pull up your pants and tighten your belt, it's going to be a bumpy ride.

Installing RMINC

Building the RMINC Package. I need to write this after I have confirmed building in Linux.

Loading the RMINC Package. OK, just to make sure that RMINC is properly installed and available, try to load the package. So, start R and type:

[6]Since googling for `R` returns quite a bit of stuff, try using `r-project` as a search term to limit things a bit more.

```
> library(RMINC)
```

Loading RMINC causes a couple of other required packages to be loaded (e.g., lattice, grid). If you don't have these, go and install them now — I'll wait for you. Done yet? Once you've loaded the package, try displaying the package overview document.

```
> # show overview doc page
> package ? mincIO
```

Note that in the R system, preceding the function of interest with a question mark will pull up that function's documentation. I'll be showing use of the help system throughout all examples.

Getting Volume Information

Reading volume header information. OK, now begins the fun part. Let's start by getting some header information from a 3D volume by using the `mincIO.readMincInfo` function.

```
> # see doc with: "?mincIO.readMincInfo"
>
> # get information for the 305 volume
> v305PET <- "average305_PET_t1_tal_lin.mnc"
> v305PET <- file.path(volDir, v305PET)
> volInfo <-  mincIO.readMincInfo(v305PET)
```

Firstly, on a point of semantics, all mincIO functions whose names start with either `read*` or `write*` will do file access, whereas those that start with either `get*` or `put*` will not — they are usually associated with manipulating in-memory structures. As such, `mincIO.readMincInfo` is going to the file to access volume header information. Right? In the above code, we set up a full path name pointing to the volume of interest, and then call `mincIO.readMincInfo`, passing the name of the file. If this works without a hitch, the `mincIO.readMincInfo` function will return a S4 MincInfo object which we assign to variable `volInfo`. While the MincInfo object does not contain any image data, it does contain all sorts of volume header information. Wanna see?

```
> print(volInfo)

---- Volume Specific Information ----
File: /Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/average305_PET_t1_ta
Interpreted data class: REAL
Internal data type: 8-bit unsigned integer
Volume real data range:     0.000 /     0.000

Image dimensions:
       sizes steps  starts units
zspace    80  1.50  -37.50    mm
yspace   128  1.72 -126.08    mm
xspace   128  1.34  -85.76    mm
```

The display of the MincInfo object information illustrates a couple of important concepts. Firstly, instead of creating a special function to display the object information, we simply called the same `print()` function that is used throughout R to display the contents of a variable. By the way, since R automatically generates a `print()` function every time a variable's name is entered, we could have printed the contents of `volInfo` simply by entering `volInfo` and hitting return. Secondly, and on a more technical note, although the built-in R `print()` function is entered by the user, under the covers, a whack of specialized code is executed in order to produce the specially formatted display that we see. In the object oriented programming world, this attribute is called polymorphism, and it allows users to call the same function name, while having the actual work done by very different functions. We use this feature a lot in the mincIO package. In particular, we overload the standard `print()` and `plot()` functions a fair bit, permitting us to produce special minc-ified output.

Before we go on to the next function, we need to learn how to peek inside the actual contents of the various mincIO objects.[7] While using `print(volInfo)` presents us with a nicely formatted display, it does not show us everything. If you want or need to see all of the underlying information, use the R "structure" function. Let's take a look at the details of the `volInfo` object.

```
> str(volInfo)

Formal class 'MincInfo' [package "RMINC"] with 10 slots
  ..@ volumeDataClass     : int 0
  ..@ volumeDataType      : int 100
  ..@ spaceType           : chr "native____"
  ..@ nDimensions         : int 3
  ..@ dimInfo             :'data.frame':        3 obs. of  4 variables:
  .. ..$ sizes : int [1:3] 80 128 128
  .. ..$ steps : num [1:3] 1.5 1.72 1.34
  .. ..$ starts: num [1:3] -37.5 -126.1 -85.8
  .. ..$ units : chr [1:3] "mm" "mm" "mm"
  ..@ nFrames             : int 0
  ..@ timeWidths          : num 0
  ..@ timeOffsets         : num 0
  ..@ volumeIntensityRange: num [1:2] 0 0
  ..@ filename            : chr "/Users/jnikelski/R_libs/2.10//RMINC/packageData/v
```

Lots of stuff, yes? It contains all of the information seen in the display, and a bit more. Each element within the structure has a fairly intuitive name, and if you desire, any element can be extracted and used programmatically within R. For those of you who desire an in-depth knowledge of R S4 objects — have a look at the "Writing R Extensions" document that comes with R. Also, please note that there is a better (and recommended) way of extracting this information when writing your own scripts; we'll be looking at that way

[7]OK, we don't really *need* to know this, although it's useful if you have some slightly more complex needs that require you programatically to access the contents of some of the mincIO objects. Don't worry too much if stuff is still a little unclear.

of doing this next. But first, let's take a look at a few examples of doing this the hard way, since it provides a bit of insight into how the mincIO objects are actually constructed.[8]

```
> # print the filename
> print(volInfo@filename)

[1] "/Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/average305_PET_t1_tal

> # show number of dimensions
> print(volInfo@nDimensions)

[1] 3

> # get no. dims +7 and print result
> dimsPlus <- volInfo@nDimensions +7
> print(dimsPlus)

[1] 10
```

Fine. We use the "@" sign to serve as a delimiter in S4 objects. Now let's try something a bit more complex.

```
> # show number of elements
> print(volInfo@dimInfo$sizes)

[1]  80 128 128

> # get the number of axial slices
> print(volInfo@dimInfo$sizes[1])

[1] 80
```

[8]Of course, if you just want to get a job done, feel free to skip over this bit and move right on to the `mincIO.getProperty()` method.

These commands need some explanation. The storage of image data within volumes is quite flexible, permitting data to be ordered in a number of different ways. For example, image data can be stored as axial slices (e.g., zyx-order), coronal slices (e.g., yxz-order), or sagittal slices (e.g., xyz-order). This flexibility can make writing code unnecessarily complicated, since one always has to check for data orientation. In mincIO, this issue was addressed by using an extremely useful feature of the MINC2 API — the ability to set an "apparent" order to be used for IO operations, regardless of the actual order of the image data within the volume. As such, all mincIO IO operations use an *apparent* order of ZYX. That is, all volumes are read as if they were comprised of axial slices, and this is why the `print(volInfo@dimInfo$sizes)` function shown above displays the dimension information in ZYX order.[9] The final command demonstrates that, since R uses 1-relative indexing, in order to extract the number of (axial) slices, one needs to get the first element of the `volInfo@dimInfo$sizes` vector. If you're starting to get a little queasy about this internal stuff, please rest assured that one will normally never have to deal with it, since there is a better way of getting at this information. So relax. Take a deep breath. It's all going to be all right.[10]

Now for the easier and recommended way of doing the above. Specifically, the better way involves use of the `mincIO.getProperty()` set of methods, which have conveniently been defined for all mincIO objects. So let's do exactly what we just did, only using the new accessor methods.

```
> mincIO.getProperty(volInfo, "filename")       # print the filename

[1] "/Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/average305_PET_t1_tal

> mincIO.getProperty(volInfo, "nDimensions")    # show number of dimensions

[1] 3
```

[9]The Civet pipeline creates all volumes in ZYX order, thus any overhead required by the MINC2 API to reorder the volumes is bypassed.

[10]The phrase "Relax. Take a deep breath. It's all going to be all right." is a trademark of Universal Mind Control Corp.

```
> mincIO.getProperty(volInfo, "sizes")          # show number of elements

xspace yspace zspace
   128    128     80

> mincIO.getProperty(volInfo, "sizes")[3]       # get the number of axial slices

zspace
    80

> mincIO.getProperty(volInfo, "sizes")["zspace"]  # ... again, by name

zspace
    80

> mincIO.getProperty(volInfo, "starts")         # get dim start values

 xspace  yspace  zspace
 -85.76 -126.08  -37.50
```

Much easier, huh? Also, note that you don't have to worry about where in the S4 object the information exists; you just specify a single property name, and the accessor finds it for you. Additionally, note that the dimension-related information is returned in xyz-order, reflecting the standard mincIO ordering, so no need to manually reorder anything. Nice, yes?

If you would like to change some of the object's properties, try the `mincIO.setProperty()` function. It works similarly to the `mincIO.getProperty()` function, with the following differences: (1) the `mincIO` object is changed in-place, thus no return value is used, and (2) not all properties can be changed. Specifically, properties which, when changed, could result in an inconsistent `mincIO` object are not permitted to be modified. So, for example, all dimension-related properties cannot be changed, as such changes may no longer reflect the dimensionality of the object's data array. So, let's look at a couple of examples.

```
> # show structure before
> str(volInfo)
```

```
Formal class 'MincInfo' [package "RMINC"] with 10 slots
  ..@ volumeDataClass     : int 0
  ..@ volumeDataType      : int 100
  ..@ spaceType           : chr "native____"
  ..@ nDimensions         : int 3
  ..@ dimInfo             :'data.frame':        3 obs. of  4 variables:
  .. ..$ sizes : int [1:3] 80 128 128
  .. ..$ steps : num [1:3] 1.5 1.72 1.34
  .. ..$ starts: num [1:3] -37.5 -126.1 -85.8
  .. ..$ units : chr [1:3] "mm" "mm" "mm"
  ..@ nFrames             : int 0
  ..@ timeWidths          : num 0
  ..@ timeOffsets         : num 0
  ..@ volumeIntensityRange: num [1:2] 0 0
  ..@ filename            : chr "/Users/jnikelski/R_libs/2.10//RMINC/packageData/v

> # change filename
> mincIO.setProperty(volInfo, "filename", "my_bunny_goes_hop.mnc")
> # change intensity range
> mincIO.setProperty(volInfo, "volumeIntensityRange", c(0,123))
> # show structure after
> str(volInfo)

Formal class 'MincInfo' [package "RMINC"] with 10 slots
  ..@ volumeDataClass     : int 0
  ..@ volumeDataType      : int 100
  ..@ spaceType           : chr "native____"
  ..@ nDimensions         : int 3
  ..@ dimInfo             :'data.frame':        3 obs. of  4 variables:
  .. ..$ sizes : int [1:3] 80 128 128
  .. ..$ steps : num [1:3] 1.5 1.72 1.34
  .. ..$ starts: num [1:3] -37.5 -126.1 -85.8
  .. ..$ units : chr [1:3] "mm" "mm" "mm"
  ..@ nFrames             : int 0
  ..@ timeWidths          : num 0
  ..@ timeOffsets         : num 0
  ..@ volumeIntensityRange: num [1:2] 0 123
  ..@ filename            : chr "my_bunny_goes_hop.mnc"
```

What we see is a change in both the filename and the volume intensity range properties. Note that since the filename is of character type, a literal string was supplied specifying the new name; the volume intensity range value needed to be supplied a 2-element numeric vector, reflecting the underlying slot type. As would be expected, attempts to modify an unsettable property shall be greeted with a stern rebuke.

Doing Stuff with Volumes

mincIO was designed to be able to do IO at 3 different levels of granularity: (1) the volume, (2) the slice, and (3) the voxel. There are 2 primary reasons for this decision. Firstly, although doing volume-level IO is intuitive, memory constraints can prevent us from loading any given number of volumes or frames into memory at one time. I first ran into this problem when trying to load all 36 frames of an ICBM-sampled volume — the malloc needed $\approx$ 2 GB of contiguous memory, which simply wasn't available. In addition, 32-bit systems also have a somewhat limited process address space, which one might also, sooner or later, bump up against. The solution to this problem is to permit IO to occur at slice granularity; a solution that isn't particularly novel, since EMMA has been doing this from the beginning. Finally, to round things out, we have voxel-level IO, permitting the user to read the value of a particular voxel, across all frames and all volumes. Examples of slice and voxel granularity IO can be seen a little later on.

Reading 3D volumes. Let's start out looking at volume-granularity by loading a 3D volume and printing the volume header information.

```
> # see doc with: "?mincIO.readVolume"
> #
> # load the ICBM-152 volume
> vIcbm <- "icbm_avg_152_t1_tal_lin.mnc"
> vIcbm <- file.path(volDir, vIcbm)
> vol <- mincIO.readVolume(vIcbm)
> print(vol)

---- Volume Specific Information ----
File: /Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/icbm_avg_152_t1_tal_
Interpreted data class: REAL
Internal data type: 16-bit signed integer
Volume real data range:   236.802 / 439776.228

Image dimensions:
       sizes steps starts units
zspace   181     1    -72    mm
yspace   217     1   -126    mm
xspace   181     1    -90    mm

---- Volume Display-Related Properties ----
Volume type: anatomical
Colormap used for display: gray
```

So we simply create a string containing the full path and file name, and call the `mincIO.readVolume()` function, which returns a MincVolumeIO object in variable `vol`. As before, we use the standard (overloaded) `print()` function to print the variable information. To get a visual summary of the volume, just call the `plot()` function, passing the new `vol` variable, the results of which can be seen in Figure 1.

```
> plot(vol)
```

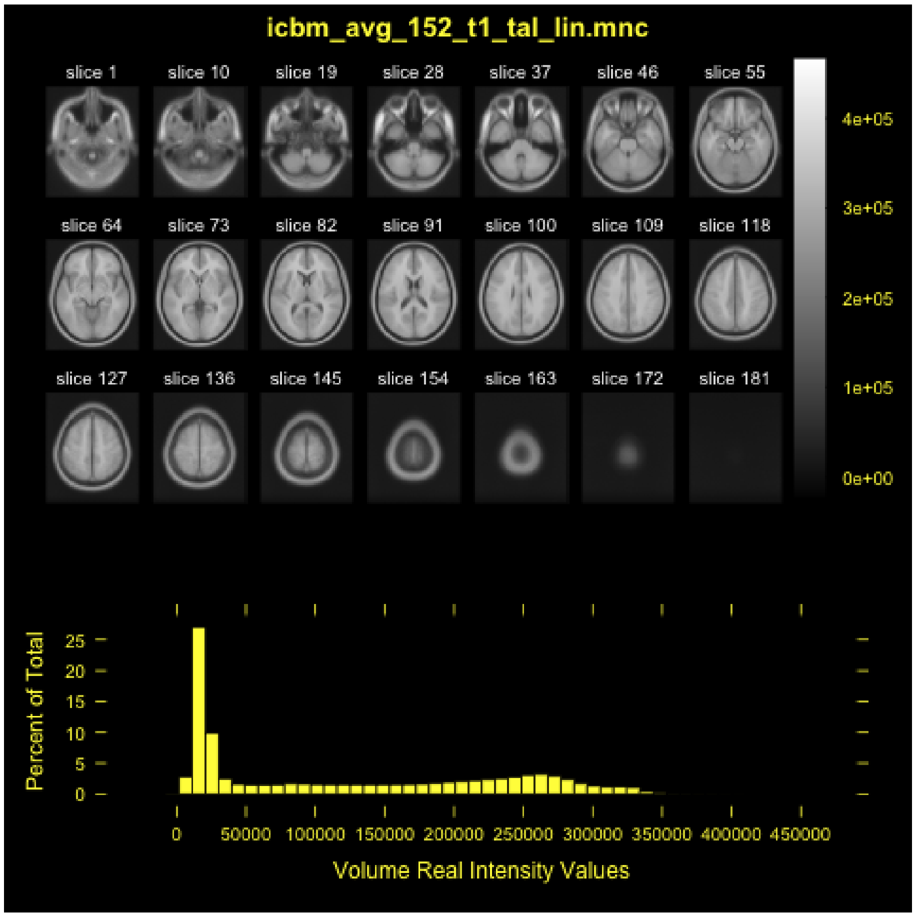

Figure 1. Summary plot of a 3-dimensional volume (ICBM-152). Slices are axial, and an intensity histogram is also provided.

Simple volume modifications. Adding or subtracting a constant to or from every value in the volume occurs as one would expect. In the example below, we zero-center the volume by subtracting the volume mean from each voxel.

```
> # add or subtract a value from every voxel in volume
> vol_adj <- vol - mean(vol)
```

Now let's try something a little more adventurous. First, a mask is created by thresholding the volume at an intensity value of 100000. Note that the resultant mask is actually a 3D array — *not* a MincVolumeIO object — this is important to note. Next, the 3D mask is applied to the original volume, yielding a masked 3D image — which is also *not* a MincVolumeIO object, as the arithmetic operations stripped the object attributes, leaving us with a simple 3D array. The final line of code converts the 3D image data back into a MincVolumeIO object by using the `mincIO.asVolume()` function, which replaces the `vol` image data with the new maskedImage 3D array, creating a new volume object (`vol2`). Of course, the `mincIO.asVolume()` function call could have been incorporated in the previous command, as demonstrated in the final line.

```
> # make a mask array (not a volume object)
> mask3D <- vol[,,]>100000
> class(mask3D)

[1] "array"

> # apply the mask element-wise to the volume
> maskedImage <- vol * mask3D
> class(maskedImage)

[1] "array"

> # recast image data into a MincVolumeIO object
> vol2 <- mincIO.asVolume(maskedImage, vol)
> class(vol2)

[1] "MincVolumeIO"
attr(,"package")
[1] "RMINC"

> # combining previous 2 lines into 1
> vol2 <- mincIO.asVolume(vol * mask3D, vol)

> plot(vol2)
```

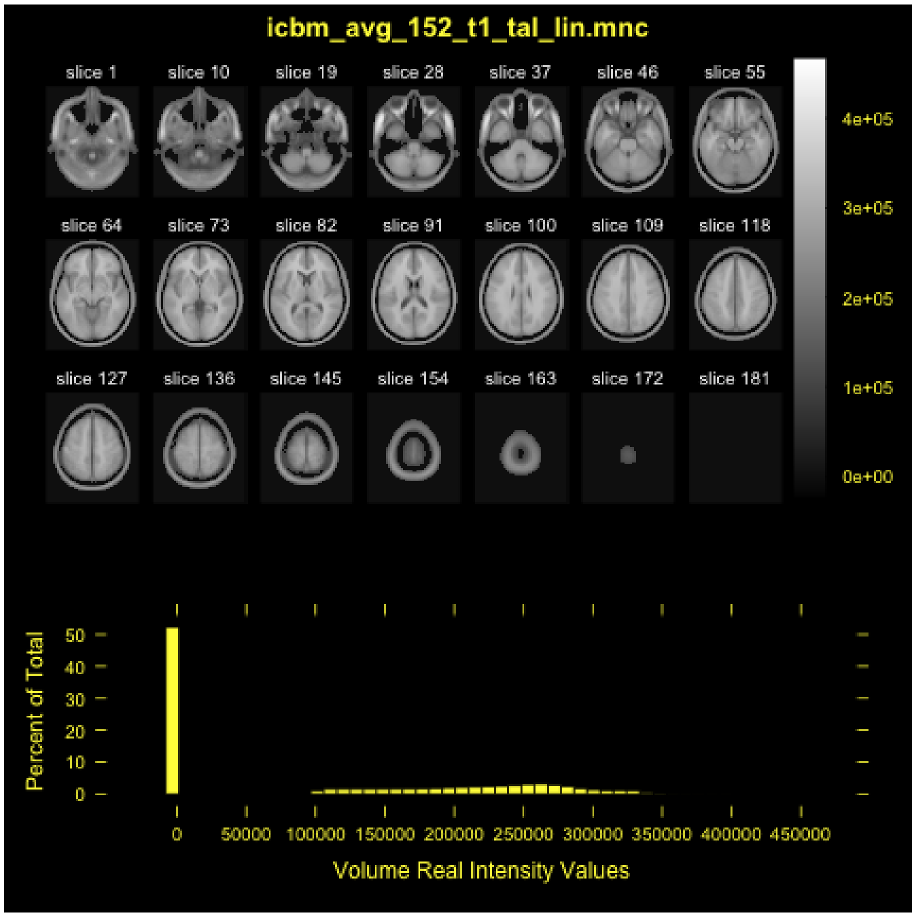

Figure 2. Summary plot of a masked 3-dimensional volume (ICBM-152). The effect of the masking at an intensity level of 100000 can clearly be seen in the histogram.

See Figure 2 for a summary plot of the newly created, masked volume data. While the `mincIO.asVolume()` function does a nice job of recasting arrays back into volumes, it would be nice if a select few arithmetic operations didn't strip off the object attributes in the first place. I believe that this is solvable by over-loading some of the more commonly used operators. Available in the next version? Stay tuned.

Volume modification using 3D indices. 3D volumes, once loaded, are treated a 3D arrays within R, and as such, elements within the array can be accessed via standard 3D indexing and slicing techniques. Before continuing with an example, we need to briefly cover a few characteristics of these arrays. Firstly, R array (and vector) indexing is 1-relative, meaning that the first element of any array or vector is element 1 (not 0, as in C/C++). Thus, after loading a 3D volume into memory, the voxel-space origin is at index [1, 1, 1] — which is left, posterior, and inferior. Secondly, on a related note, although the MINC2 API enforces an *apparent* dimension order of ZYX (resulting in axial slices), the order of indices when accessing the 3D array uses the more conventional XYZ ordering. Moreover, in an attempt to keep the interface consistent, all other mincIO functions also use the XYZ ordering. Finally, although we shall be observing how specific voxel values can be read and modified, modifying a great number of voxel values in a volume using a voxel loop is probably not a good things to do. The reason is because R uses "pass-by-value" semantics, which means that whenever we attempt to modify a data structure, behind the scenes, R actually makes a complete copy of the structure that we *think* we're modifying, and then returns it. So, modifying a single voxel value will result in a copy of the entire 3D array. It's usually better to vectorize such operations, rather than looping and making modifications voxel-by-voxel. Be careful.

Ok, now let's break by own suggestion by reading in a 3D volume and changing a number of voxel values. So, let's read the volume and change a few things.

```
> # read the ICBM-152 volume
> vol <- mincIO.readVolume(vIcbm)
> # set a voxel within the caudate (slice 82) to high intensity
> vol[85, 142, 82] <- 500000     # left caudate
> vol[95, 142, 82] <- 500000     # right caudate
```

So, there you have it. One voxel within each caudate in slice 82 has been set to a relatively high value. Of course, I could `plot()` the volume, but you wouldn't see anything anyway (believe me, I tried). So, you're going to have to trust me. No? OK, let's try a bigger manipulation. Let's draw a line through the caudate from left to right (still in slice 82).

```
> # draw a line through the caudate (slice 82)
> vol[20:160, 142, 82] <- 500000
```

Note the X-index is now specified as an integer sequence, ranging from voxel 20 (left side) to 160 (right side). Sadly, this line also does not show up on the volume summary plot — it's too thin. Let's make it a bit thicker, using a for-loop (yes, yes, I know, not efficient, but this is a didactic exercise).

```
> # draw a line through the caudate (slice 82)
> # ... and make it 5 voxels thick
> for ( ndx in 140:144) {
+ vol[20:160, ndx, 82] <- 500000
+ }
```

There you go. Can you see it? Take a gander at Figure 3.

```
> plot(vol)
```

Modifying volume hyperslabs. Now that we can set individual voxel values, let's take a look at creating a large rectangular region of interest (ROI) using a hyperslab. The ROI is comprised of a large central block of brain, anchored at the anterior commissure. The basic steps consist of: (1) reading the volume to be masked, (2) defining the hyperslab, (3) instantiating an empty volume and writing the hyperslab into it, and (4) applying the ROI mask to the input volume. The code looks a little like this — well, actually, since we're using Sweave, the code looks exactly like this.

```
> # read volume
> vol <- mincIO.readVolume(vIcbm)
> #
```

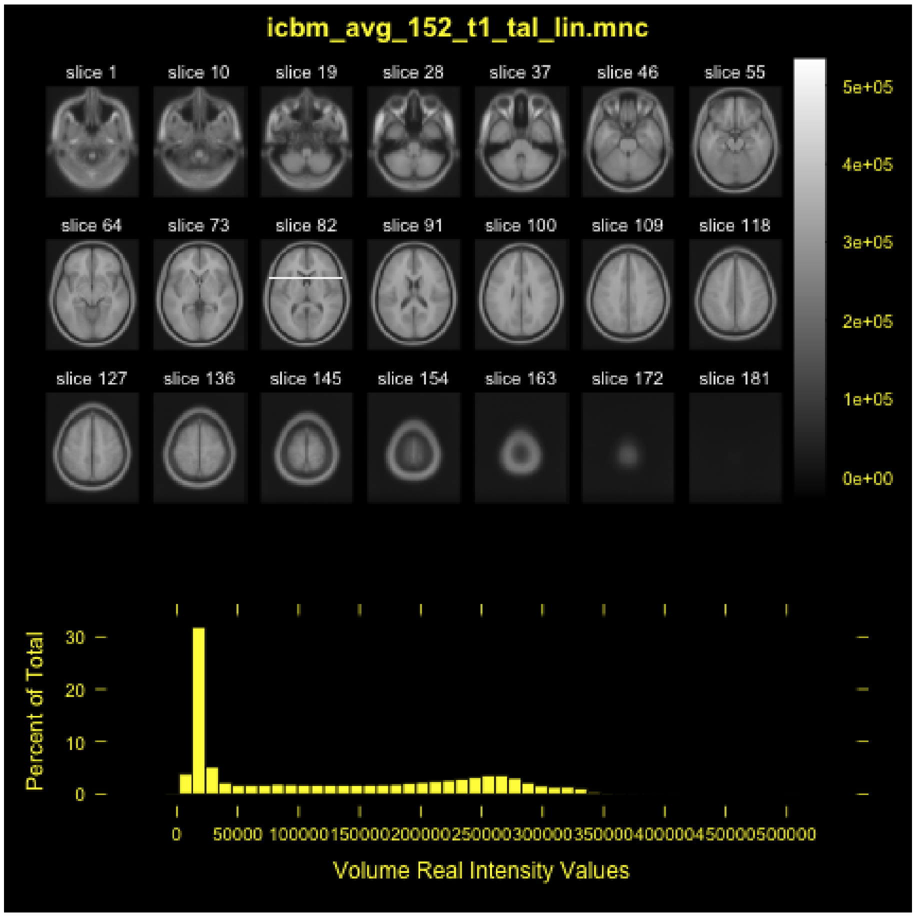

Figure 3. ICBM volume with a 5-voxel line "drawn" through the caudate at slice 82. The modification was accomplished by directly manipulating the voxel values within the 3D volume array.

```
> # define the bound of the hyperslab
> # ... X-axis start and stop voxel (+-30 mm from AC)
> rangeX <- numeric()
> rangeX[1] <- rminc.convertWorldToVoxel(vIcbm, c(0-30, 0, 0))[1]
> rangeX[2] <- rminc.convertWorldToVoxel(vIcbm, c(0+30, 0, 0))[1]
> #
> # ... Y-axis start and stop voxel (-60 mm, +30 mm from AC)
> rangeY <- numeric()
> rangeY[1] <- rminc.convertWorldToVoxel(vIcbm, c(0, 0-60, 0))[2]
> rangeY[2] <- rminc.convertWorldToVoxel(vIcbm, c(0, 0+30, 0))[2]
> #
> # ... Z-axis start and stop voxel (-20 mm, +30 mm from AC)
> rangeZ <- numeric()
> rangeZ[1] <- rminc.convertWorldToVoxel(vIcbm, c(0, 0, 0-20))[3]
> rangeZ[2] <- rminc.convertWorldToVoxel(vIcbm, c(0, 0, 0+30))[3]
> #
> # ... create a new initialized mask volume and create hyperslab mask
> maskVol <- mincIO.makeNewVolume(filename="mask_volume.mnc", likeTemplate="icbm
> maskVol[ rangeX[1] : rangeX[2],
+          rangeY[1] : rangeY[2],
+          rangeZ[1] : rangeZ[2] ] <- 1
> #
> # ... apply the mask to the T1 volume and recast as volume
> hyperVol <- mincIO.asVolume(vol * maskVol, vol)
```

Note the new `rminc.convertWorldToVoxel()` function, that takes a volume name and a vector of world coordinates (XYZ order). Perhaps in the near future, we'll be able to pass it a MincVolumeIO object, saving us some file IO, but sadly that day has not yet come. Also new is the `mincIO.makeNewVolume()` function, which creates a new (initialized to zeros) volume, with the attributes set according to a known template (ICBM, in this case). Figure 4 displays the ROI within the masked volume.

```
> plot(hyperVol)
```

Doing Stuff with Slices

While working with 3D volumes directly, as demonstrated above, is convenient and intuitive, there is also a wee bit of a performance hit, given R's pass-by-value semantics. We can minimize this performance penalty by working with individual slices rather than the entire volume while doing updates, since each

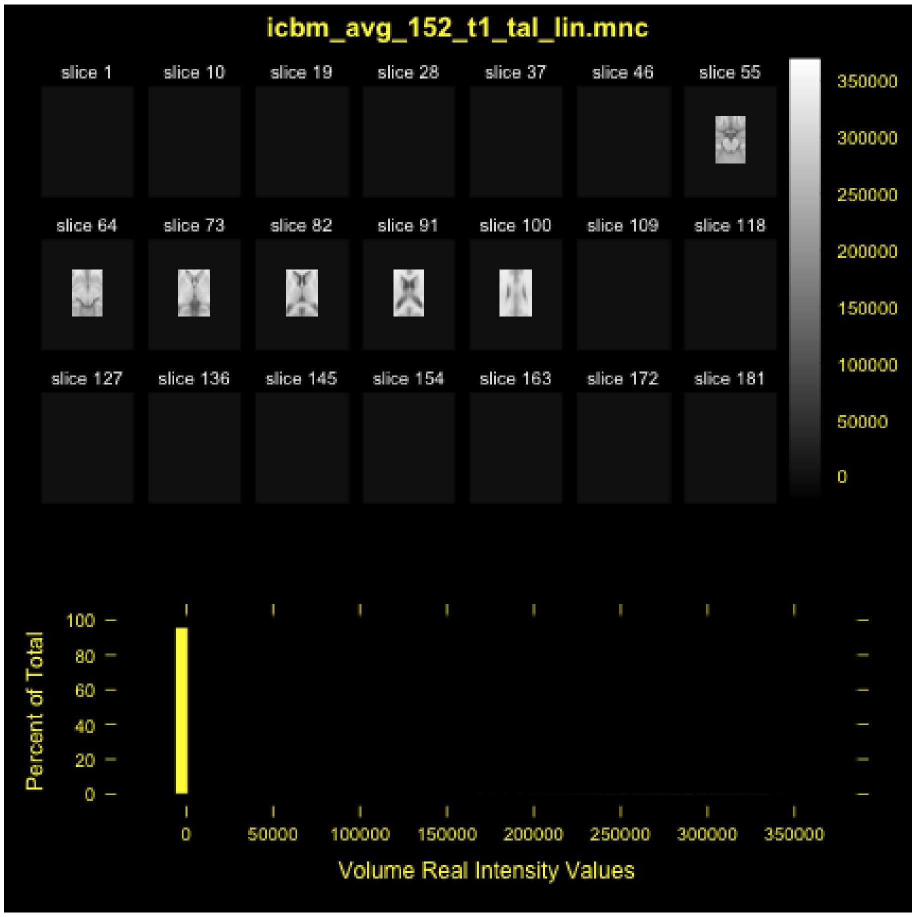

Figure 4. ICBM volume masked with a hyperslab ROI. The hyperslab mask was created relative to the anterior commissure, and ranges between the following extents: X-axis=(AC±30 mm), Y-axis=(AC-60 mm, AC+30 mm), and Z-axis=(AC-20 mm, AC+30 mm).

update will only result in the slice being copied, rather than the entire volume. To do this, mincIO provides 3 extractor functions, `mincIO.getSliceX()`, `mincIO.getSliceY()`, and `mincIO.getSliceZ()`, which serve to extract a slice of a given orientation from a 3D MincVolumeIO object. Additionally, function `mincIO.putSlice()` serves to move a slice back into a MincVolumeIO object. Let's take a look at a simple example.

```
> # see doc with: "?mincIO.getSliceZ"
> #
> vol <- mincIO.readVolume(vIcbm)        # read the ICBM volume
> s055 <- mincIO.getSliceZ(vol, 55)      # extract axial slice 55
> print(s055)                            # print summary info

---- Volume Specific Information ----
File: /Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/icbm_avg_152_t1_tal_
Interpreted data class: REAL
Internal data type: 16-bit signed integer
Volume real data range:   236.802 / 439776.228

Image dimensions:
       sizes steps starts units
zspace   181     1    -72    mm
yspace   217     1   -126    mm
xspace   181     1    -90    mm

---- Slice Specific Information ----
Slice number: 55
Slice orientation: zSlice
Min/max data values in slice: 8604.065 / 330780.2

---- Volume Display-Related Properties ----
Volume type: anatomical
Colormap used for display: gray
Aspect ratio: 1.19889502762431
```

We start simply by loading the ICBM volume, and extracting the slice at voxel coordinate 55, resulting in the creation of a Slice object (s055). Details about the slice (and the volume from which it was extracted) are displayed using the over-loaded `print()` function, and the slice is then visualized using the over-loaded `plot()` function (see Figure 5).

```
> plot(s055)                              # visualize slice 55
```

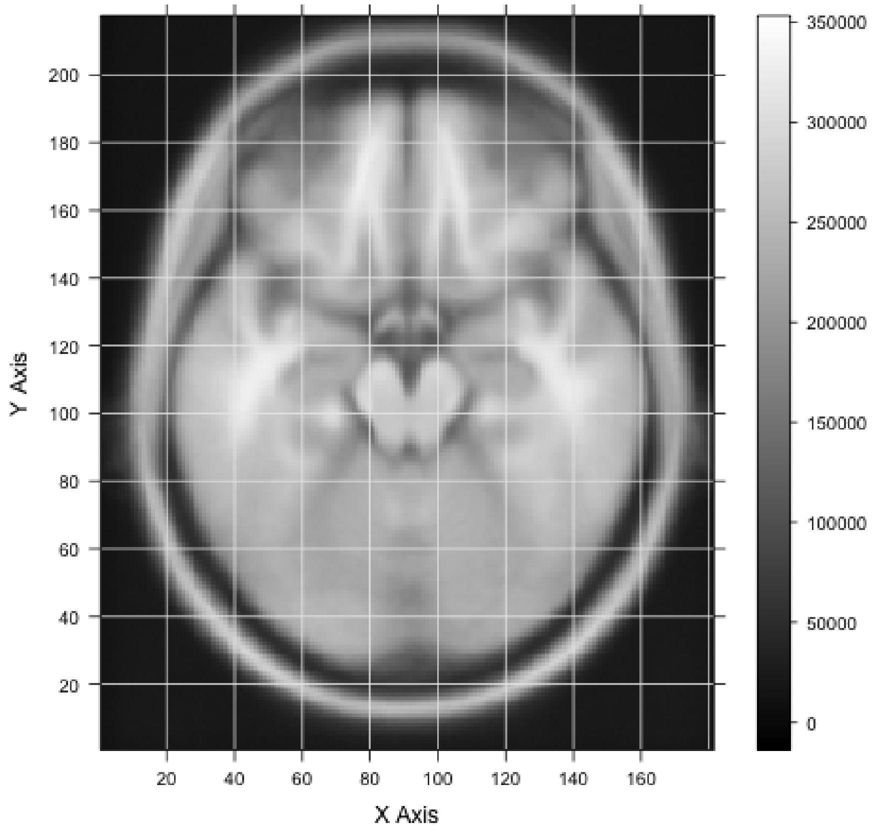

Figure 5. Axial slice 55 extracted from the ICBM volume. Default colormap is grayscale; voxel origin (voxel[1,1]) is located in the bottom-left portion of the image (posterior-left within the volume).

So now that we can see our slice, let's modify it in a visualizeable way. Note that, just as the MincVolumeIO object is a 3D array at heart, and therefore can be modified using the standard R 3D indexing, the Slice object is really a simple 2D array, and so the R matrix indexing should work just fine on these objects as well. In this example, we use matrix indexing to set a couple of square ROIs to high intensity values, and a central ROI to low intensity values.

```
> s055[20:40, 180:200] <- max(vol)   # place square in upper left
> s055[140:160, 20:40] <- max(vol)   # ... and in lower right
> s055[60:120, 80:120] <- 0          # block out brainstem

> plot(s055)                         # visualize slice 55
```

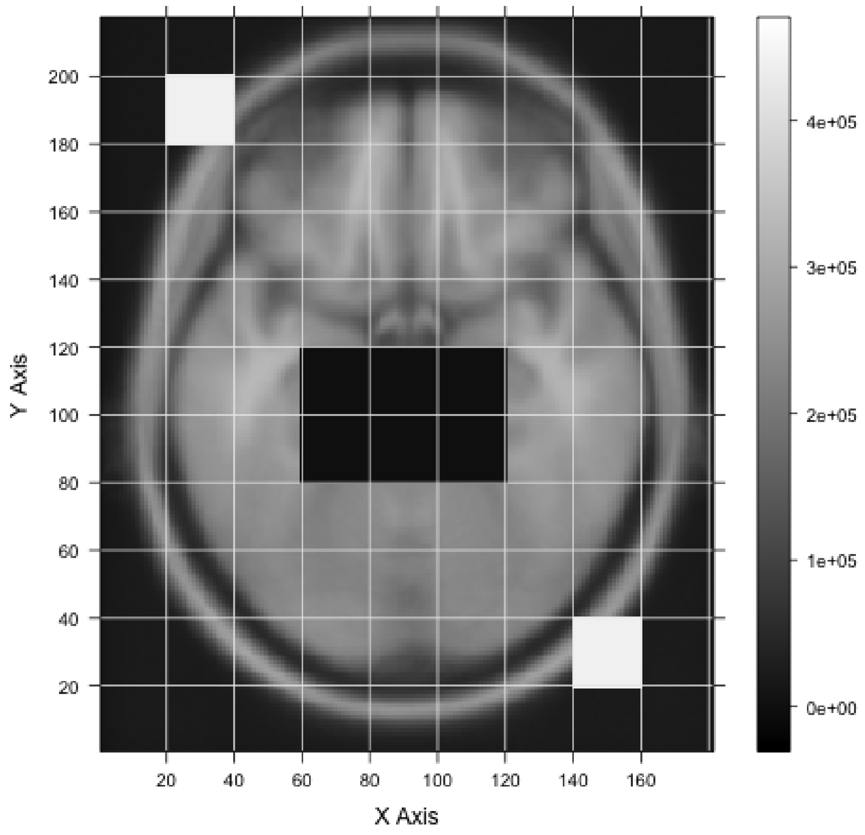

Figure 6. Display of a modified axial slice 55. Upper left and lower right blocks are set to high intensity values (white blocks), and the larger central block is set to low intensity values (black).

The results of the modifications are can be seen in Figure 6. The final step would be to move the modified slice back into the MincVolumeIO object prior to writing the modified 3D object back to disk. This is easily done via the `mincIO.putSlice()` function, and looks like the following.

```
> vol <- mincIO.putSlice(s055, vol)     # update vol with moded slice
```

Note that the only mandatory inputs to `mincIO.putSlice()` are the slice and the volume to be updated. The slice number need not be specified, as the slice already "knows" its slice number, and therefore puts itself back into the correct place within the volume. An explicit specification of the slice number (3rd argument) allows the user to write the slice back into a different position within the volume (although the dimensionality must still match). In order to prove that the updated slice *has* been written to the volume, let's take a look a the volume plot shown in Figure 7.

```
> plot(vol)                            # visualize volume
```

Doing Stuff with Slices Over Frames and Volumes

We now explore using mincIO to do IO using slice granularity. As discussed previously, slice-level IO is the default in EMMA and is particularly useful when dealing with either a large number of 3D volumes, or multi-frame 4D volumes, for which loading all of the volume data into memory would not be possible. Additionally, since slice granularity IO places the data into a 2D matrix, with each matrix column storing a slice, data manipulation across all slices using the standard R matrix manipulation functions is facilitated. In this section, we will demonstrate application of slice IO for the 2 primary use cases: (1) reading a given slice over a series of 3D volumes, and (2) reading a given slice across all frames of a 4D functional volume. We'll start with the 3D case.

Read a slice over 3D volumes. In our first example, we'll read a given slice over a number of 3D volumes. Note that the sampling must be the same across all volumes, and that slice IO reads only axial slices.[11] In our example, we read axial slice number 46 across 5 3D volumes using the `readBySlice()` function, and then print out the slice array summary information.

[11] This may change in the future if someone is able to provide me with a compelling reason.

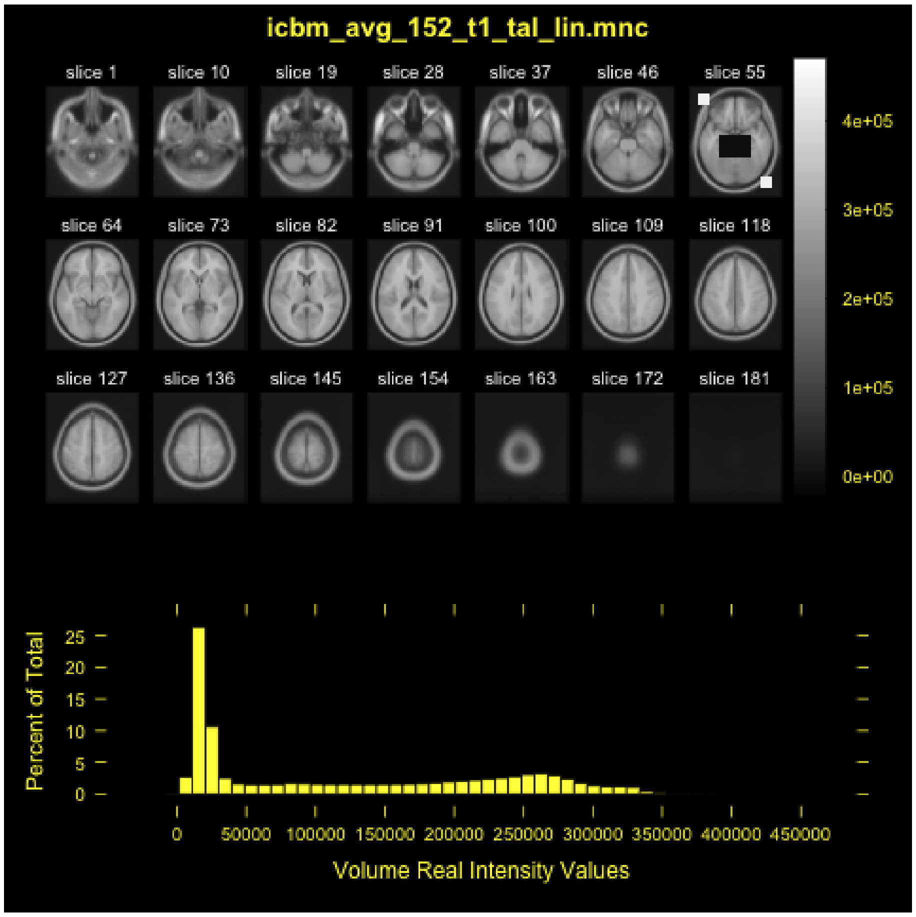

Figure 7. Display of a modified ICBM volume after inserting the changed slice 55 back into the volume using the `putSlice()` function.

```
> # see doc with: "?mincIO.readBySlice"
> #
> volumes <- c(vol1, vol2, vol3, vol4, vol5)          # set volume names
> slice_array_s46 <- mincIO.readBySlice(volumes, 46)  # load slice 46
> print(slice_array_s46)

---- Volume Specific Information ----
File: /Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/icbm_avg_152_t1_tal_
Interpreted data class: REAL
Internal data type: 16-bit signed integer
```

```
Volume real data range:     0.000 /     0.000

Image dimensions:
       sizes steps starts units
zspace   181     1    -72    mm
yspace   217     1   -126    mm
xspace   181     1    -90    mm

---- Slice Specific Information ----
Slice number: 46
Slice orientation: zSlice
Number of slices in matrix: 5
Slice origin: Slices over volumes
SliceIO matrix dimensions: [39277, 5]
Volume Names:
1.  /Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/icbm_avg_152_t1_tal_li
2.  /Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/icbm_avg_152_t1_tal_li
3.  /Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/icbm_avg_152_t1_tal_li
4.  /Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/icbm_avg_152_t1_tal_li
5.  /Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/icbm_avg_152_t1_tal_li

---- Volume Display-Related Properties ----
Volume type: anatomical
Colormap used for display: gray
```

In addition to the common volume information (common to all volumes) the `print()` function also shows some slice array details, including, slice number, matrix dimensions, and volume names.[12] As the next step, let's say that you needed to extract a particular slice from the slice array for visualization, manipulation, etc. How would one do that? Let's try the aptly named

[12]The astute reader will notice that all of the volumes are the same. Yes, I cheated by reading the same volume 5 times, since distributing mincIO with a bunch of sample volumes was really not a good thing. Corners needed to be cut.

mincIO.getSliceFromSliceArray() function.

```
> # see doc with: "?mincIO.getSliceFromSliceArray"
> #
> vol3s46 <- mincIO.getSliceFromSliceArray(slice_array_s46, 3)
> print(vol3s46)

---- Volume Specific Information ----
File: /Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/icbm_avg_152_t1_tal_
Interpreted data class: REAL
Internal data type: 16-bit signed integer
Volume real data range:     0.000 /     0.000

Image dimensions:
       sizes steps starts units
zspace   181     1    -72    mm
yspace   217     1   -126    mm
xspace   181     1    -90    mm

---- Slice Specific Information ----
Slice number: 46
Slice orientation: zSlice
Min/max data values in slice: 8831.89 / 353798.2

---- Volume Display-Related Properties ----
Volume type: anatomical
Colormap used for display: gray
Aspect ratio: 1.19889502762431
```

So there you have it. Slice 46 from the 3rd volume is now instantiated as a slice object, which we are free to either modify, move to a MincVolumeIO object, or simply visualize — as we see in Figure 8 (note the change in the colorMap property, just because we can).

```
> mincIO.setProperty(vol3s46, "colorMap", "heat.colors")
> plot(vol3s46)                        # plot extracted slice
```

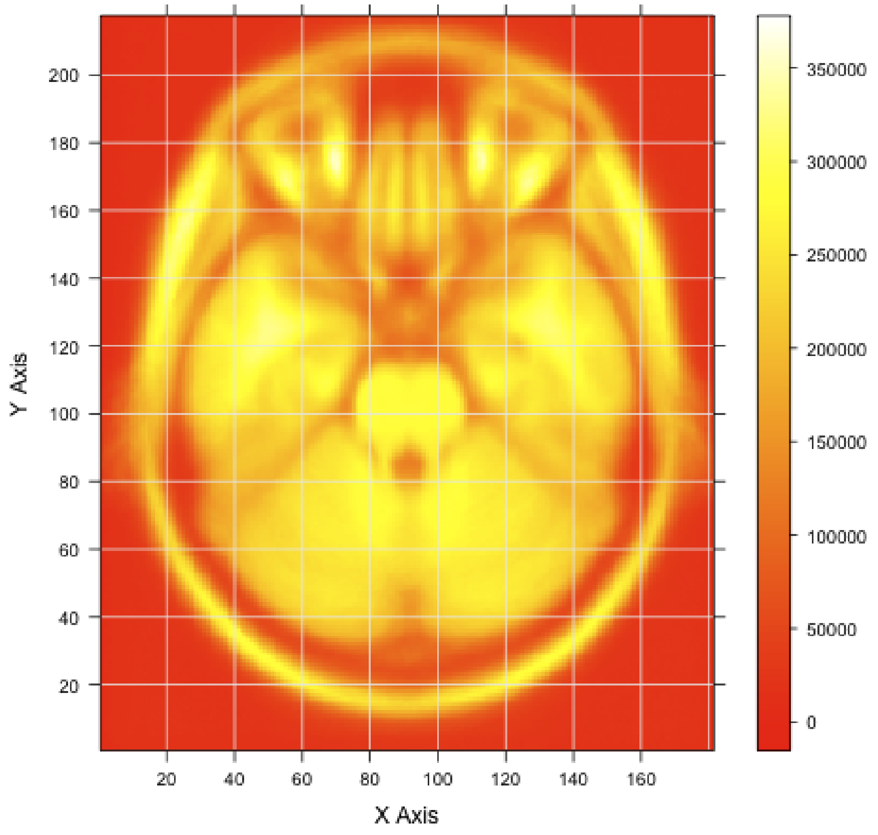

Figure 8. Slice 46 from the 3rd volume, as extracted from the slice array, displayed using the *heat.colors* color map.

Read a slice over all 4D volume frames. The second use case is very similar to the first, except that we are now reading frames within a single 4D volume. Given that the syntax is really pretty similar, let's do this quickly. Take a look at the following.

```
> slice_array_s30 <- mincIO.readBySlice(v4D, 30)     # load slice 30
> print(slice_array_s30)
```

```
---- Volume Specific Information ----
File: /Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/functional_4D.mnc
Interpreted data class: REAL
Internal data type: 8-bit unsigned integer
Volume real data range:     0.000 /     0.000

Image dimensions:
       sizes    steps   starts   units
time      34 145.4545    0.016 seconds
zspace    91   2.0000  -72.000      mm
yspace   109   2.0000 -126.000      mm
xspace    91   2.0000  -90.000      mm

----- Dynamic (Time) Information -----
          [,1]   [,2]   [,3]   [,4]   [,5]   [,6]    [,7]
Offsets  0.016 15.016 30.016 45.016 60.016 90.016 120.016
Widths  15.000 15.000 15.000 15.000 30.000 30.000  30.000
           [,8]    [,9]   [,10]   [,11]   [,12]   [,13]
Offsets 150.016 180.016 210.016 240.016 270.016 300.016
Widths   30.000  30.000  30.000  30.000  30.000  60.000
          [,14]   [,15]   [,16]   [,17]   [,18]   [,19]
Offsets 360.016 420.016 480.016 540.016 600.016 660.016
Widths   60.000  60.000  60.000  60.000  60.000  60.000
          [,20]   [,21]   [,22]    [,23]    [,24]    [,25]
Offsets 720.016 780.016 840.016 1020.016 1200.016 1500.016
Widths   60.000  60.000 180.000  180.000  300.000  300.000
           [,26]    [,27]    [,28]    [,29]    [,30]
Offsets 1800.016 2100.016 2400.016 2700.016 3000.016
Widths   300.000  300.000  300.000  300.000  300.000
           [,31]    [,32]    [,33]    [,34]
Offsets 3300.016 3600.016 4200.016 4800.016
Widths   300.000  600.000  600.000  600.000
```

```
---- Slice Specific Information ----
Slice number: 30
Slice orientation: zSlice
Number of slices in matrix: 34
Slice origin: Slices over frames
SliceIO matrix dimensions: [9919, 34]

---- Volume Display-Related Properties ----
Volume type: functional
Colormap used for display: rainbow
```

The name of a 4D volume is loaded into the variable v4D, and then function `mincIO.readBySlice()` is called, requesting that slice 30 be loaded from all frames. The `print()` function then displays similar information as previously displayed, in addition to some frame-specific information (e.g., frame offsets and widths across all frames). Extracting the slice from frame 10, we see a bit more frame-specific detail. Note that, since we are dealing with 4D volumes, the default colormap for display is "rainbow" (sort of like "spectral"), as evidenced by Figure 9.

```
> frame10s30 <- mincIO.getSliceFromSliceArray(slice_array_s30, 10)
> print(frame10s30)

---- Volume Specific Information ----
File: /Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/functional_4D.mnc
Interpreted data class: REAL
Internal data type: 8-bit unsigned integer
Volume real data range:     0.000 /     0.000

Image dimensions:
       sizes    steps   starts   units
```

```
time      34 145.4545    0.016 seconds
zspace    91   2.0000  -72.000      mm
yspace   109   2.0000 -126.000      mm
xspace    91   2.0000  -90.000      mm

----- Dynamic (Time) Information -----
          [,1]   [,2]   [,3]   [,4]   [,5]   [,6]    [,7]
Offsets  0.016 15.016 30.016 45.016 60.016 90.016 120.016
Widths  15.000 15.000 15.000 15.000 30.000 30.000  30.000
           [,8]    [,9]   [,10]   [,11]   [,12]   [,13]
Offsets 150.016 180.016 210.016 240.016 270.016 300.016
Widths   30.000  30.000  30.000  30.000  30.000  60.000
          [,14]   [,15]   [,16]   [,17]   [,18]   [,19]
Offsets 360.016 420.016 480.016 540.016 600.016 660.016
Widths   60.000  60.000  60.000  60.000  60.000  60.000
          [,20]   [,21]   [,22]    [,23]    [,24]    [,25]
Offsets 720.016 780.016 840.016 1020.016 1200.016 1500.016
Widths   60.000  60.000 180.000  180.000  300.000  300.000
           [,26]    [,27]    [,28]    [,29]    [,30]
Offsets 1800.016 2100.016 2400.016 2700.016 3000.016
Widths   300.000  300.000  300.000  300.000  300.000
           [,31]    [,32]    [,33]    [,34]
Offsets 3300.016 3600.016 4200.016 4800.016
Widths   300.000  600.000  600.000  600.000


---- Slice Specific Information ----
Slice number: 30
Slice orientation: zSlice
Min/max data values in slice: -169.3141 / 813.0569

---- Volume Display-Related Properties ----
```

```
Volume type: functional
Colormap used for display: rainbow
Aspect ratio: 1.1978021978022

> plot(frame10s30)                    # plot extracted slice
```

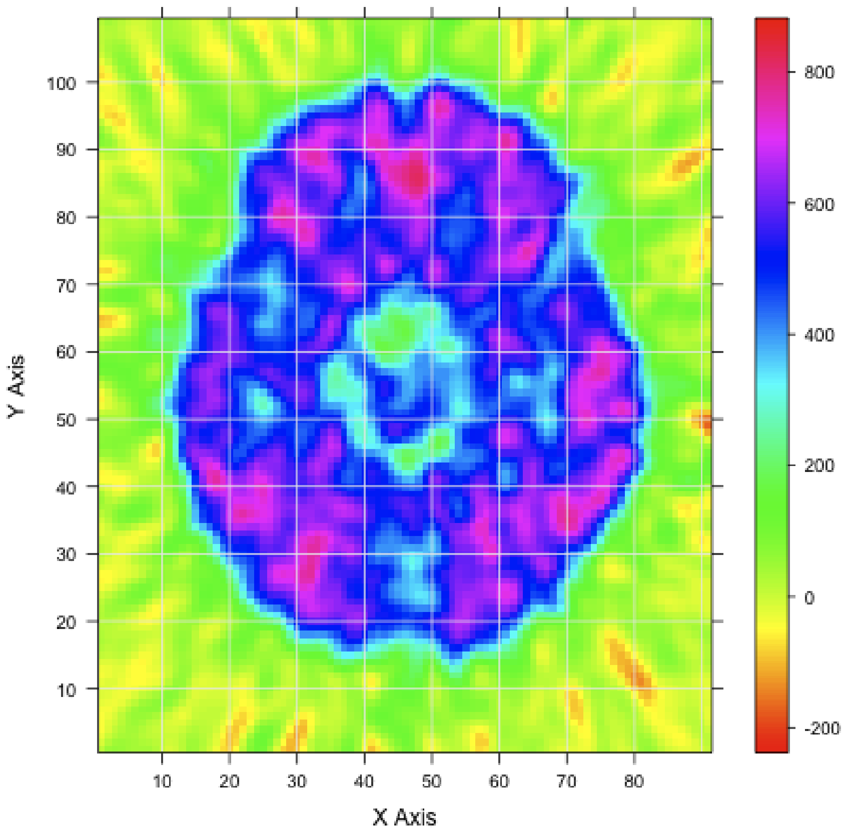

Figure 9. Slice 30 taken from frame 10 of a 4D volume, as extracted from the slice array. Note that the default colormap is “rainbow”, as 4D volumes are (by default) understood to contain functional data.

Doing Stuff with Voxels

An now we arrive at doing voxel-level IO. Voxel-level IO works with one or more 3D or 4D volumes, and extracts the real value at the specified voxel, across frames (if 4D) and across volumes. Let's start with a 3D example.

```
> volumes <- c(vIcbm, vIcbm ,vIcbm, vIcbm, vIcbm)
> #
> worldCoords <- c(0,0,0)           # get AC voxel coords
> voxCoords <- round(rminc.convertWorldToVoxel(vIcbm, worldCoords))
> print(voxCoords)

[1]  91 127  73

> #
> voxel_array <- mincIO.readByVoxel(volumes, voxCoords)
> print(voxel_array)

---- Volume Specific Information ----
File: /Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/icbm_avg_152_t1_tal_
Interpreted data class: REAL
Internal data type: 16-bit signed integer
Volume real data range:     0.000 /     0.000

Image dimensions:
       sizes steps starts units
zspace   181     1    -72    mm
yspace   217     1   -126    mm
xspace   181     1    -90    mm

---- Voxel Specific Information ----
Sampled coordinate (voxel space): [ 91, 127, 73 ]
Sampled coordinate (world space): [ 0.000000, 0.000000, 0.000000 ]
VoxelIO matrix dimensions: [5, 1]
```

```
Volume Names:
1.  /Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/icbm_avg_152_t1_tal_li
2.  /Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/icbm_avg_152_t1_tal_li
3.  /Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/icbm_avg_152_t1_tal_li
4.  /Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/icbm_avg_152_t1_tal_li
5.  /Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/icbm_avg_152_t1_tal_li

> print(voxel_array[,])

[1] 175580.5 175580.5 175580.5 175580.5 175580.5
```

Ok, it's pretty straightforward. We're endeavouring to read the real value stored at the voxel located at the mid anterior commissure across 5 volumes. Actually, all 5 volumes are the same (didactic license again), so the values all better be the same. As all access uses voxel coordinates, we first convert the stereotactic address to voxel space, and then print it out.[13] The actual read is accomplished with `mincIO.readByVoxel()`, followed by a print of the voxel array meta information, and then the actual array values themselves. Easy as π. We can do the exact same thing with 4D volumes. Take a boo.

```
> volumes <- c(v4D, v4D, v4D, v4D, v4D)
> worldCoords <- c(0,0,0)          # get AC voxel coords
> voxCoords <- round(rminc.convertWorldToVoxel(v4D, worldCoords))
> print(voxCoords)

[1] 46 64 37

> voxel_array <- mincIO.readByVoxel(volumes, voxCoords)
> print(voxel_array)
```

[13]Be aware that although the mincIO package is perfectly capable of reading and processing native-space volumes, note that the space conversion routines may not be of much use to you unless you're in stereotactic space. Specifically, in native space, one can get all sorts of unusual Start values, since the origin is machine relative; this can result in funny (e.g., negative) voxel indices being generated by `rminc.convertWorldToVoxel()`, which are usually not super useable.

```
---- Volume Specific Information ----
File: /Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/functional_4D.mnc
Interpreted data class: REAL
Internal data type: 8-bit unsigned integer
Volume real data range:     0.000 /     0.000

Image dimensions:
       sizes    steps   starts   units
time      34 145.4545    0.016 seconds
zspace    91   2.0000  -72.000      mm
yspace   109   2.0000 -126.000      mm
xspace    91   2.0000  -90.000      mm

----- Dynamic (Time) Information -----
          [,1]   [,2]   [,3]   [,4]   [,5]   [,6]    [,7]
Offsets  0.016 15.016 30.016 45.016 60.016 90.016 120.016
Widths  15.000 15.000 15.000 15.000 30.000 30.000  30.000
           [,8]    [,9]   [,10]   [,11]   [,12]   [,13]
Offsets 150.016 180.016 210.016 240.016 270.016 300.016
Widths   30.000  30.000  30.000  30.000  30.000  60.000
          [,14]   [,15]   [,16]   [,17]   [,18]   [,19]
Offsets 360.016 420.016 480.016 540.016 600.016 660.016
Widths   60.000  60.000  60.000  60.000  60.000  60.000
          [,20]   [,21]   [,22]    [,23]    [,24]    [,25]
Offsets 720.016 780.016 840.016 1020.016 1200.016 1500.016
Widths   60.000  60.000 180.000  180.000  300.000  300.000
           [,26]    [,27]    [,28]    [,29]    [,30]
Offsets 1800.016 2100.016 2400.016 2700.016 3000.016
Widths   300.000  300.000  300.000  300.000  300.000
           [,31]    [,32]    [,33]    [,34]
Offsets 3300.016 3600.016 4200.016 4800.016
Widths   300.000  600.000  600.000  600.000
```

```
---- Voxel Specific Information ----
Sampled coordinate (voxel space): [ 46, 64, 37 ]
Sampled coordinate (world space): [ 0.000000, 0.000000, 0.000000 ]
VoxelIO matrix dimensions: [5, 34]
Volume Names:
1.  /Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/functional_4D.mnc
2.  /Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/functional_4D.mnc
3.  /Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/functional_4D.mnc
4.  /Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/functional_4D.mnc
5.  /Users/jnikelski/R_libs/2.10//RMINC/packageData/volumes/functional_4D.mnc
```

```
> print(voxel_array[,])
```

```
          [,1]      [,2]     [,3]     [,4]     [,5]
[1,] -0.784392 -6.398356 8.491878 78.82688 108.5726
[2,] -0.784392 -6.398356 8.491878 78.82688 108.5726
[3,] -0.784392 -6.398356 8.491878 78.82688 108.5726
[4,] -0.784392 -6.398356 8.491878 78.82688 108.5726
[5,] -0.784392 -6.398356 8.491878 78.82688 108.5726
         [,6]     [,7]     [,8]     [,9]    [,10]    [,11]
[1,] 131.0117 157.8758 195.4130 271.4548 301.9661 200.2652
[2,] 131.0117 157.8758 195.4130 271.4548 301.9661 200.2652
[3,] 131.0117 157.8758 195.4130 271.4548 301.9661 200.2652
[4,] 131.0117 157.8758 195.4130 271.4548 301.9661 200.2652
[5,] 131.0117 157.8758 195.4130 271.4548 301.9661 200.2652
        [,12]    [,13]    [,14]    [,15]    [,16]    [,17]
[1,] 200.1842 137.9639 175.7178 187.9526 220.7555 163.9762
[2,] 200.1842 137.9639 175.7178 187.9526 220.7555 163.9762
[3,] 200.1842 137.9639 175.7178 187.9526 220.7555 163.9762
[4,] 200.1842 137.9639 175.7178 187.9526 220.7555 163.9762
[5,] 200.1842 137.9639 175.7178 187.9526 220.7555 163.9762
```

```
        [,18]    [,19]    [,20]    [,21]    [,22]    [,23]
[1,] 192.6460 182.3952 155.4194 193.0671 145.9818 153.2386
[2,] 192.6460 182.3952 155.4194 193.0671 145.9818 153.2386
[3,] 192.6460 182.3952 155.4194 193.0671 145.9818 153.2386
[4,] 192.6460 182.3952 155.4194 193.0671 145.9818 153.2386
[5,] 192.6460 182.3952 155.4194 193.0671 145.9818 153.2386
        [,24]   [,25]    [,26]    [,27]    [,28]    [,29]
[1,] 103.1237 88.7511 58.39751 85.23587 26.69102 76.98288
[2,] 103.1237 88.7511 58.39751 85.23587 26.69102 76.98288
[3,] 103.1237 88.7511 58.39751 85.23587 26.69102 76.98288
[4,] 103.1237 88.7511 58.39751 85.23587 26.69102 76.98288
[5,] 103.1237 88.7511 58.39751 85.23587 26.69102 76.98288
        [,30]    [,31]    [,32]    [,33]    [,34]
[1,] 64.33478 44.54658 51.10174 29.23720 99.99635
[2,] 64.33478 44.54658 51.10174 29.23720 99.99635
[3,] 64.33478 44.54658 51.10174 29.23720 99.99635
[4,] 64.33478 44.54658 51.10174 29.23720 99.99635
[5,] 64.33478 44.54658 51.10174 29.23720 99.99635
```

There you have it. Looks pretty much the same as the 3D case, but now the voxel array is comprised of a 5×34 matrix, with the rows reflecting the volumes, and the columns reflecting the frames. By the way, note that while the `print()` functions have been overloaded to display useful object meta-information, the `plot()` function has not. Reason? I really can't think of a useful image to display with regard to the slice and voxel matrix objects. Really, I can't. Questions? Just to finish the example, let's plot the time activity curve at the sampled voxel.

```
> timeOffsets <- mincIO.getProperty(voxel_array, "timeOffsets")
> print(timeOffsets)
```

```
 [1]    0.016   15.016   30.016   45.016   60.016   90.016
 [7]  120.016  150.016  180.016  210.016  240.016  270.016
```

```
[13]  300.016  360.016  420.016  480.016  540.016  600.016
[19]  660.016  720.016  780.016  840.016 1020.016 1200.016
[25] 1500.016 1800.016 2100.016 2400.016 2700.016 3000.016
[31] 3300.016 3600.016 4200.016 4800.016
```

```
> xyplot(voxel_array[1,] ~ round(round(timeOffsets)/60),
+ xlab='Frame Acquisition Time (Minutes)',
+ ylab=paste('Real Voxel Value at voxel [', paste(voxCoords, collapse=" "), ']',
+        panel = function(x, y) {
+            panel.grid(h=-1, v= 2)
+            panel.xyplot(x, y)
+            panel.loess(x,y, span=.5)})
```

Batch Processing

Although working within an IDE during script development has significant advantages, there comes a time in every script's life when it wants to be batched. Examples would include, for example, writing scripts to be executed at the command line, permitting the user to write code in R, but exposing an interface similar to the standard minc tools, or additionally, permitting the script to be submitted to a batch queueing system (SGE, PBS, etc.). This functionality is made available via the `Rscript` command, and permits one to treat R as any other scripting language. Let's start with a simple example.

To start, say that we simply wanted to write a script to read in a minc volume and display the volume header information. We write a script that looks similar to the following.

```
#!/usr/bin/env Rscript   \
--default-packages=utils,methods,base,stats,grDevices
# load RMINC library
suppressPackageStartupMessages(library(RMINC))
#
args <- commandArgs(TRUE)                 # read script args
```

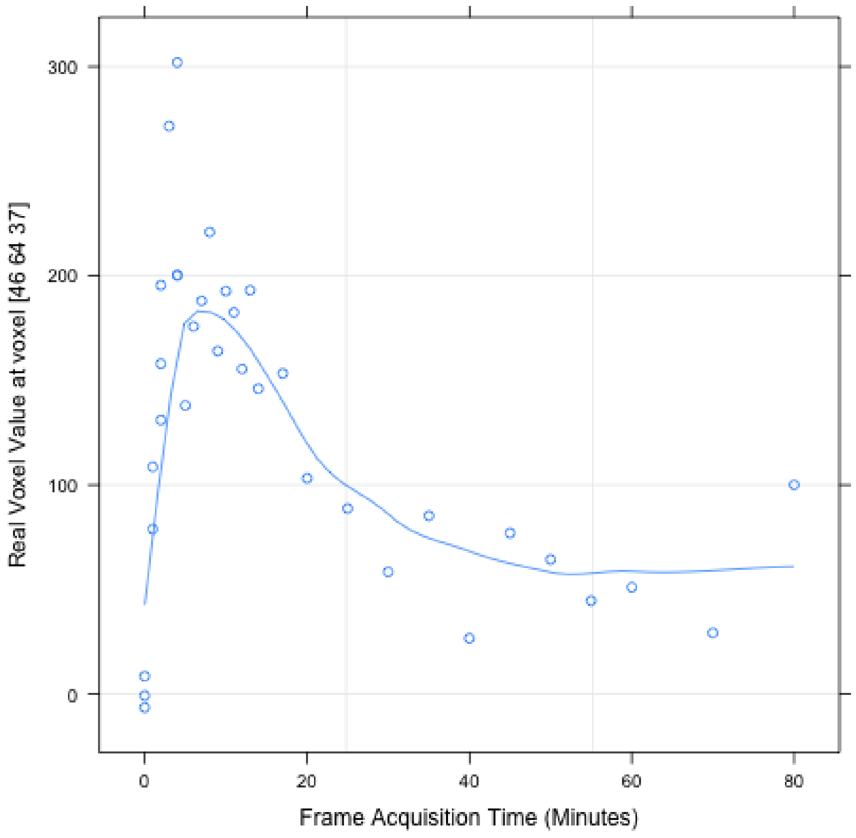

Figure 10. Time activity curve at the sampled voxel, for the first volume, taken over all 34 frames. A rather wiggly curve has been fitted to the sample points.

```
mincInfo <- mincIO.readMincInfo(args[1])  # read the volume info
print(mincInfo)                           # ... and display it
```

So we save this script to a file under an appropriately descriptive name, say “showMincInfo.Rscript”, set the execute bit via `chmod +x ...`, and then run it. Of note: (1) the `-default-packages` switch tells R which base R packages are to be loaded automatically; the list shown above is a reasonable selection, and reflects the packages loaded automatically during an interactive session, and (2) the `commandArgs()` reads all of the arguments from the command line and places them into a character vector. Those are the basics. Want to see the output?

```
>showMincInfo.Rscript average305_PET_t1_tal_lin.mnc

---- Volume Specific Information ----
File: /Users/jnikelski/tmp/rscript/average305_PET_t1_tal_lin.mnc
Interpreted data class: REAL
Internal data type: 8-bit unsigned integer
Volume real data range:     0.000 /     0.000

Image dimensions:
       sizes steps  starts units
zspace    80  1.50  -37.50    mm
yspace   128  1.72 -126.08    mm
xspace   128  1.34  -85.76    mm
```

The output appears as expected, I suppose. Clearly, everything that can be done interactively, can also be done using Rscript, including generation of plots and other images.

Summary and Future Development

So there you have a taste of the RMINC-mincIO module, and its capabilities. Clearly, I was not able to demonstrate all package functionality in this tutorial, however, the reference information integrated into the R help system will be able to provide you with a list of all functions and how to use them. A great deal of work went into writing that documentation, so please feel free to use it. With regard to learning R, remember that google is your friend — an ever-growing amount of documentation is being written and made freely available as R gains acceptance amongst college statistics instructors. While becoming an R wizard is not strictly necessary in order to use the mincIO package, is must be remembered that R is both an environment *and* a full featured language;

the more confident you are with the R language, the better use you'll be able to make use of mincIO.

Subsequent development would likely focus on using R to analyze cortical surfaces. A preliminary (proof of concept) version of a surfaceIO package already exists, and is capable of reading ASCII ".obj" files, and rendering the surfaces for inspection using the rgl package. Much work remains to be done; stay tuned.

Finally, I suppose I need to mention the Windows platform. If one of the ultimate goals of this project is make it easier to use the Minc volume format, and therefore increase its use, we need to remember that 95% of the world is using Windows. The inability of the current Minc tools to build and run on Windows is necessarily going to reduce adoption of these tools, and therefore use of the Minc volume format. It should be remembered that this restriction does not exist in the world of Matlab-Nifti, and so Windows users will naturally gravitate to that development platform. I believe that this problem should not be insurmountable, if we do the conversion in stages. I would suggest that, given that the Minc2 library is only dependent on the HDF5 library (which *does* build under windows), we start by implementing a Windows build of the Minc2 library. This has the following advantages: (1) a proof of concept build under Windows with a relatively simple, encapsulated library; lessons learned could be applied to build all of Minc for Windows, and (2) as RMINC is *only* dependent on the Minc2 library, once we have the Minc2 dlls, we would immediately be able to start using mincIO under Windows. Sadly, multi-platform building is not my strength, so we would require the services of someone else to tackle this.

Thanks for your interest in the RMINC-mincIO package. Feel free to share your ideas and critiques with others. Not with me. I don't handle criticism very well. I kid. A little.