EMMA: Extensible MATLAB Medical Analysis User Manual

Mark Wolforth Greg Ward Sean Marrett

20 November, 1995

Contents

1 Introduction

This document is the user manual for EMMA (Extensible MATLAB Medical Analysis) developed at the Montreal Neurological Institute. This package consists of a set of MATLAB .m files and programs written in C, designed to ease the analysis of medical image data. The entire package runs under MATLAB 4.X.

This document assumes that the reader is familiar with manipulation of medical images (specifically images stored in the MINC file format)[1], and simply wishes to know how image manipulation is possible using EMMA and MATLAB.

The authors would like to thank Sean Marrett of the McConnell Brain Imaging Centre for being the driving force behind this project. He provided invaluable guidance when all seemed lost.

[1]MINC and the associated netCDF package are available via anonymous FTP from ftp.bic.mni.mcgill.ca, in **/pub/minc**.

2 Basic Concepts

To a programmer, the interface to EMMA should be very familiar. Simply open a data set contained in a MINC file by calling the **openimage** function. This returns a handle which is used in all subsequent calls to EMMA functions. The user does not need to worry about the underlying mechanisms. For example, to open an image file, read in the the first frame of the second slice, and then close the image file, you could type:

```
data_handle = openimage ('arnaud_20547.mnc');
image = getimages (data_handle, 2, 1);
closeimage (data_handle);
```

In this example, `data_handle` is a handle to the data contained in the MINC file `arnaud_20547.mnc`. After the data file has been opened with **openimage**, the name of the MINC file is not needed by the user of EMMA; only the handle for the data is used. Therefore, **getimages** is passed the handle for the data, as well as the slice number and frame number. Once you have finished working with the image, always remember to call **closeimage**. This cleans up global variables created in the MATLAB workspace.

Because EMMA uses handles, the user never needs to worry about the interface between MATLAB and the MINC file. You simply call EMMA functions with the handle, and the underlying structure takes care of the details.

However, the user *does* need to have some familiarity with MATLAB, and a good understanding of how EMMA functions deal with images. In particular, the inherent two-dimensionality of MATLAB means that a full four-dimensional MINC file is "flattened" in some way when read into MATLAB.

Generally, EMMA functions deal with images transformed from their natural two-dimensional form (well-suited to a matrix representation) to a one-dimensional vector form. For example, a single 128×128 image will become a *column* vector of dimension 16384×1. We can put several such vectors together in a matrix. For example, we can form a 16384×15 matrix from 15 slices. Thus, in an "image matrix"—which is a two-dimensional MATLAB matrix representing three-dimensional image data—the individual columns correspond to whole images. MATLAB functions that use EMMA to perform analysis should always expect image data in this format, as shown in Section 4 and the document "rCBF Analysis Using MATLAB."

Also, keep in mind that EMMA makes it deliberately difficult to deal simultaneously with the full four dimensions allowed in a MINC image variable. The reasoning for this is two-fold: first, collapsing three dimensions into two via the "image matrix" concept is fairly convenient and easy to work with; however, attempting to similarly collapse four dimensions would add another layer of complications, making both the implementation and use of EMMA functions more difficult. Second is the question of memory usage: MATLAB is capable of using up enormous amounts of memory if care is not taken, and restricting the user of EMMA to three image dimensions at a time means that it is somewhat more difficult to blindly read in massive amounts of image data with a single MATLAB command.

3 Basic EMMA MATLAB Functions

The basic MATLAB interface to EMMA is performed through the following MATLAB functions (.m files). This section is only meant to provide a brief introduction to the most important functions. See section 5.3 for full help on every EMMA function.

openimage - Prepares a MINC file for reading. **openimage** determines the image size, reads in the frame start times and lengths if applicable, and returns a handle which can be used to access the MINC file with **getimages**, **getimageinfo**, **getblooddata**, etc.

```
handle = openimage (filename)
```

getimages - Gets images from a MINC file. If the file has no frame dimension, the parameter `frames` should be empty or not provided. For files with no slices (only frames), the `slices` parameter should be made empty (a matrix of the form `[]`). Also, **getimages** can read several frames from a single slice or several slices from a single frame—but it can not read multiple slices and multiple frames simultaneously. (This restriction is also imposed for writing images, mainly to avoid complications and reduce the software's ability to easily read in far more data than can be handled by MATLAB.)

Two important features of **getimages** are the ability to re-use memory, and the ability to read partial images. Both of these features are intended to reduce the amount of memory used by MATLAB when processing images.

```
Images = getimages (handle [, slices [, frames [, old_matrix ...
                    [, start_row [, num_rows]]]]])
```

newimage - Creates a MINC file for a new set of images. Returns a handle to the newly created data set that is used in calls to other functions (such as **putimages** and **closeimage**). **newimage** has a number of optional parameters, the most important of which is the "parent file." If this is supplied, **newimage** will copy a number of important attributes (such as the patient, study, and acquisition data) from the parent to the new file, as well as using the parent file to provide the default image size and type.

```
handle = newimage (filename, dim_sizes [, parent_file [, image_type ...
                   [, valid_range [, orientation]]]]);
```

putimages - Writes entire images to a file created with **newimage**. (**Warning:** there are currently no provisions made to deny the user from writing to a MINC file opened with **openimage**.) The syntax is similar to **getimages**, except that the image data is of course an input argument to **putimages** rather than an output argument.

```
putimages (handle, images [, slices [, frames]])
```

getimageinfo - Gets information about an open image. Currently, **getimageinfo** will return the filename, number of frames or slices, image height, width, or size, the dimension sizes, frame start times, lengths, and mid-frame times. The desired information is specified as a character string, eg. 'NumFrames', 'ImageHeight', etc.

```
info = getimageinfo (handle, info_descriptor)
```

viewimage - Displays an image on the workstation screen. On a monochrome display, the **gray** colourmap will be used; otherwise, the **spectral** will be used. The image will be scaled such that the highest point in the image is always the last colour in the colourmap, and the lowest image point will be the first element of the colourmap. Also, a colourbar relating colours in the image to the values will be displayed, unless the optional **colourbar_flag** argument is set to zero.

```
viewimage (image [, colourbar_flag])
```

closeimage - Destroys the appropriate variables in the workspace.

```
closeimage (IMhandle)
```

4 Some Basic Examples: Masking and Summing

As an example of the EMMA functions presented thus far, we will explore sample code for both masking and summing/integrating images from a dynamic study. The MINC file used in this example will be one of the $H_2{}^{15}O$ CBF studies from **/usr/local/matlab/toolbox/emma/examples**; there is also a corresponding BNC (**b**lood **n**et**C**DF) file, which is necessary to get the plasma activity data. To make accessing these files easier, you may want to create symbolic links to them in a directory of your own:

```
cd <your-work-directory>
ln -s /local/matlab/toolbox/emma/examples/yates_19445.* .
```

That way, the file `/local/matlab/toolbox/emma/examples/yates_19445.mnc` can be accessed as simply `yates_19445.mnc` (as long as you are in your work directory). In the remainder of this section, it will be assumed that these symbolic links exist in the current directory.

4.1 Reading the Data

First, let us open the MINC file:

```
handle = openimage ('yates_19445.mnc');
```

Now, we may wish to find out some basic data about the MINC file. Try the following (note that we will need `nf`, the number of frames, below):

```
ns = getimageinfo (handle, 'NumSlices')
nf = getimageinfo (handle, 'NumFrames')
getimageinfo (handle, 'ImageSize')
```

Note that for the study `yates_19445`, `nf` will be 21. However, it is a good idea to always use `getimageinfo` to find the number of frames rather than assuming that it will always stay the same for a particular type of study.

Now, let us read in the frame lengths and mid-frame times:

```
FrameLengths = getimageinfo (handle, 'FrameLengths');
MidFTimes = getimageinfo (handle, 'MidFrameTimes');
```

Note that `FrameLengths` and `MidFTimes` are both column vectors. This will be important when performing matrix arithmetic with them.

Now, we read in all frames for one slice—slice 8 in this case, although you can work with whatever slice you like:

```
slice = 8;
PET = getimages (handle, slice, 1:nf);
```

Note that *this* is the step where MATLAB uses up large amounts of memory. If you execute the MATLAB `whos` command now, you will see that MATLAB's variables alone are taking up nearly 3 megabytes of memory. However, the entire MATLAB workspace generally takes up considerably more memory than is reported by `whos`, because of MATLAB's overhead and inefficient memory management scheme.

4.2 Integrating Images

We can view the matrix `PET` as a collection of 21 PET images for the same brain slice but progressing in time, or as a collection of 16,384 time-activity curves (TAC)—one for every pixel in the 128 $\times$ 128 image. (Of course, only about $\frac{1}{3}$ of the image data is actually data from inside the head. This matter will be addressed when we consider image masking below.)

For purposes of integrating images, we will consider `PET` as a collection of TACs. Thus, we are simply performing 16384 numerical integrations. This may seem a daunting task, but MATLAB's vectorized approach to mathematics makes it surprisingly easy. In fact, let us calculate a rectangular integration using the formula:

$$\int_{t_1}^{t_n} f(t)dt \approx \sum_{i=1}^{n} f(t_i)(\Delta t)_i, \tag{1}$$

where $f(t)$ is one time-activity curve and the Δt's are the frame lengths. Since we have $f(t)$ stored as a row vector—remember, each column of `PET` is the entire image for one frame, so each row of `PET` corresponds to a single pixel across time—and Δt stored as a column vector, the multiply/sum operation of equation (1) is simply a dot product. And since a matrix multiplication is nothing more than a sequence of dot products, all 16,384 TACs can be quite easily integrated with a single MATLAB command:

```
PETsummed1 = PET * FrameLengths;
```

However, rectangular integration is a very crude approach to numerical integration. Trapezoidal integration provides a slightly more refined method, and is almost as easy to implement. Furthermore, it is provided by the MATLAB function `trapz`—which, if you care to examine it (try `type trapz` in MATLAB), is essentially a one-line operation as well. (Incidentally, one of the functions provided by EMMA is **ntrapz**. **ntrapz** currently has exactly the same restrictions and capabilities as `trapz`, but is considerably faster due to being written in C using MATLAB's C interface. For the purposes of this discussion, the two functions are equivalent.)

`trapz` takes two arguments: a vector of x-points, and a vector or matrix of y points corresponding to each x-point. If the y's are given as a matrix, then each *row* corresponds to one x point.

Note that this is different from the EMMA standard, where each *column* of `PET` corresponds to one time point. Thus, we will have to transpose `PET` using the MATLAB `'` operator before passing it to `trapz`. Also, `trapz` will return the 16,384 integrals as a row vector, so we must transpose the results to get them back into the form we expect. Finally, `trapz` expects points on the x-axis rather than widths of intervals. Thus, we will pass `MidFTimes` to it, and not `FrameLengths`. The full integration is performed by the single statement

```
PETsummed2 = trapz (MidFTimes, PET')';
```

Note: This method of calculating the integral is extremely wasteful of memory due to transposing the PET images. For a full discussion of MATLAB memory issues, please consult the online document Controlling MATLAB Memory Use[2].

4.3 Viewing and Comparing the Integrated Images

Now, we wish to view the integrated images. If you are using a colour console or an monochrome X terminal, enter the following to create a MATLAB figure window and display the image obtained by rectangular integration:

```
figure;
viewimage (PETsummed1);
```

(Note that explicitly creating the window via the `figure` command is not really necessary. However, if you already had a figure window with a plot or image that you did not want to lose, MATLAB would have overwritten the existing figure when you called **viewimage**. It is generally good practice to call `figure` before creating new plots or images, unless you are sure you wish to obliterate whatever was previously in the current figure window.)

You may want to title this window:

```
title ('yates_19445, slice 8, rectangular integration');
```

Now do the same thing for `PETsummed2`. You will probably want to move one or both figure windows so that you can see both images simultaneously. Clearly, the two images are similar—but if we want a more objective comparison, we can calculate the ratio of the two images on a pixel-by-pixel basis and view this:

```
ratio = PETsummed1 ./ PETsummed2;
figure;
viewimage (ratio);
```

Note the use of the `./` operator to specify an element-by-element operation on two matrices. The resulting image shows that most of the pixels in `PETsummed1` and `PETsummed2` appear fairly close. However, because some pixels have such large values compared to 1, any small variations around 1 are swamped. One way to deal with this is simply to clip the image at certain points: for example, set all points below -10 to -10, and all points above 10 to 10. This is accomplished as follows:

[2] http://www.mni.mcgill.ca/system/mni/matlab/matlab_memory.html

```
clipneg = find (ratio < -10);
ratio (clipneg) = ones (size (clipneg)) * (-10);
clippos = find (ratio > 10);
ratio (clippos) = ones (size (clippos)) * (10);
```

If you wish to know how many points were clipped at either end, type `size (clipneg)` or `size (clippos)`.

If you now **viewimage (ratio)**, it should be clear that the overall ratio between the two integrated images is close to 1 inside the head.

4.4 Saving the Integrated Image

After performing some analysis, it is often desirable to write the results to a new MINC file. Currently, we only have a single image to write, so we will create a MINC file with 1 slice and no frames (because of the integration across frames, time is no longer a variable). We use EMMA function **newimage** as follows:

```
new_handle = newimage ('yates_summed.mnc', [0 1], ...
                       'examples/yates_19445.mnc');
```

To write our single image to slice 1 of the new MINC file, we enter:

```
putimages (new_handle, PETsummed2, 1);
```

Alternately, we could create a MINC file with a full 15 slices, and loop through the entire original study to sum every slice. MATLAB will allow us to enter a `for` loop interactively, so enter the following (note that if you already executed the above code, you should use a new filename in place of **yates_summed.mnc** or delete the old one—**newimage** will die if you try to overwrite an existing file):

```
closeimage (new_handle);
new_handle = newimage ('yates_summed.mnc', [0 ns], ...
                       'examples/yates_19445.mnc');
for slice = 1:ns
   PET = getimages (handle, slice, 1:nf);
   PETsummed = trapz (MidFTimes, PET')';
   putimages (new_handle, PETsummed, slice);
end
closeimage (new_handle);
```

4.5 Using an Integrated Image for Image Masking

First, let us clear some of the unneeded variables from the workspace:

```
clear PETsummed1 ratio clipneg clippos
```

Now, we will use `PETsummed2` for a very simple form of image masking: namely, we will use only pixels whose values in the integrated image are greater than the mean of the integrated image. For CBF studies, this usually creates a usable mask that contains only in-brain voxels. It can be done

by calculating the mean of the integrated image, and creating another image consisting entirely of 1's and 0's (a "binary" image), where a 1 indicates that the pixel is inside the head, and a zero indicates outside:

```
avg = mean (PETsummed2);
mask = PETsummed2 > avg;
```

We can view the mask as an image itself to see a silhouette of the brain at this slice:

```
figure;
viewimage (mask);
```

Or, we can apply the mask to the summed image and view it. To apply the mask to an image, simply perform an element-by-element multiply using the `.*` operator on the mask and the image; since `mask` consists entirely of ones and zeros, every point in the image will either be set to zero or unchanged. Thus,

```
PETsummed2 = PETsummed2 .* mask;
figure;
viewimage (PETsummed2);
```

will display the trapezoidally integrated image with out-of-head data set to zero.

Finally, to see just how many pixels are "on" in the mask, you can type

```
size (find (mask))
```

The fact that only a fraction of the entire image consists of image data means that masking can be used to great advantage when performing analysis that requires computations for every pixel. (Generally, this should be avoided if it is all possible to take advantage of MATLAB's vectorized structure.)

5 Underlying Structure

EMMA has a set of underlying functions written in C that provide the interface to MINC files. An EMMA user will not ordinarily require use of these functions since every interaction with MINC files should be possible using the MATLAB scripts and functions provided (.m files). However, for the sake of completeness, these functions are documented in this section.

MATLAB provides an interface to external C programs that allows dynamic linking at runtime. This interface, called CMEX, is detailed in the MATLAB *External Interface Guide*. Please consult this guide for further details. Unfortunately, due to problems with the CMEX interface, it was not possible to create any CMEX programs that wrote to MINC files. Therefore, all of the MINC creation and writing functions are written as stand-alone C programs, and all of the MINC reading functions are written as CMEX programs.

5.1 Stand-alone C Programs

The following programs are used by various EMMA functions and are not normally needed by the user.

- **miwriteimages** - Writes images to a MINC file. The images are taken from the specified temporary file, whose data is expected to be stored as doubles. The MINC file must exist, and contain an image structure (create with micreate and then micreateimage).

```
miwriteimages <MINC file> <slices> <frames> <temp file>
```

- **micreateimage** - Create a new MINC file, complete with image dimensions, dimension and dimension-width variables, image max/min variables, and (of course) the image variable. If the new file is based on a "parent" MINC file, any other variables and global attributes will be copied from the parent to the new file. As much information as *can* be copied about the image and image dimensions will also be copied, but the fact that the new file may or may not have the same orientation, dimensions, and dimension lengths as its parent complicates matters greatly.

```
micreateimage <MINC file> [option] [option] ...
```

- **miwritevar** - Writes values to a variable in a MINC file. Values are taken from the specified temporary file, whose data is expected to be stored as doubles.

```
miwritevar <MINC file> <var name> <start vector> ...
           <lengthvector> <temp file>
```

5.2 CMEX Programs

In addition to stand-alone C programs, the underlying structure also includes several CMEX programs which are dynamically linked by MATLAB. Please see the MATLAB *External Interface Guide* for an explanation of CMEX files.

- **mireadvar** - Read a hyperslab from a MINC (or any NetCDF) file into a MATLAB vector, using NetCDF-style start and count vectors. Note that the indexing of `start_vector` and `stop_vector` is zero-based.

```
mireadvar ('minc_file', 'var_name', start_vector, count_vector)
```

- **mireadimages** - Read images from a MINC file. Opens the given MINC file, and attempts to read whole or partial images from the slices and frames specified in the slices and frames vectors. If `start_row` and `num_rows` are not specified, then whole images are read. If only `start_row` is specified, a single row will be read. If both `start_rows` and `num_rows` are given, then the specified number of rows will be read (unless either is out of bounds, which will result in an error message).

```
mireadimages ('minc_file'[, slices[, frames[, options]]])
```

- **miinquire** - find out various things about a MINC file from MATLAB. Retrieves some item(s) of information about a MINC file. The first argument is always the name of the MINC file. Following the filename can come any number of "option sequences", which consist of the option (a string) followed by zero or more items (more strings). Generally speaking, the "option" tells miinquire the general class of information you're looking for (such as an attribute value or a dimension length), and the item or items that follow it give **miinquire** more details, such as the name of a dimension, variable, or attribute.

```
miinquire ('minc_file' [, 'option' [, 'item']], ...)
```

5.3 MATLAB files

CALPIX *generates the vector index of a point*

```
cp = calpix(x,y)
```

generates the vector index (1 to 16384) of the x and y coordinates of a pixel in a 128*128 image. It uses the simple formula
cp = round(x) + 128 * round(y-1)

CHECK_SF *for internal use only*

```
msg = check_sf (handle, slices, frames)
```

- **msg** = []: all is OK
- **msg** = error message if there's anything wrong

makes sure that the given slices and frames vectors are consistent with the image specified by handle. Checks for:

- both slices and frames cannot be vectors
- if image has no frames, frames vector should be empty
- if frames vector is empty, image must have no frames
- if image has no slices, slices vector should be empty
- if slices vector is empty, image must have no slices

CLOSEIMAGE *closes image data set(s)*

```
closeimage (handles)
```

Closes one or more image data sets.

DERIV *calculate the derivative and smoothed version of a function*

```
[yfit, deriv] = deriv (fit_points, data_points, y, dt)
```

This function calculates the derivative and smoothed version of a function, using the method of parabolic regressive filters, described in Sayers: "Inferring Significance from Biological Signals."

GETBLOODDATA *retrieve blood activity and sample times from a study*

```
[activity, mid_times] = getblooddata (study)
```

The study variable can be a handle to an open image, or the name of a NetCDF (MNC or BNC) file containing the blood activity data for the study. If it is a handle, getblooddata will first look in the associated MINC file (if any), and then in the associated BNC file (if any) for the blood activity variables. If just a filename (either a MINC or BNC file) is given, getblooddata will look in that file only.

The mid-sample times will be calculated from amongst the variables sample_start, sample_stop, and sample_length. The default is mid = (sample_start + sample_stop)/2; but if sample_stop is not found in the MNC or BNC file, then mid = sample_start + (sample_length/2) will be used instead.

If a file that does not exist is specified, or getblooddata cannot find the blood activity data in either the MINC or BNC file, it will print a warning message and return nothing (ie. empty matrices).

GETIMAGEINFO *retrieve helpful trivia about an open image*

```
info = getimageinfo (handle, whatinfo)
```

Get some information about an open image. handle refers to a MINC file previously opened with openimage or created with newimage. whatinfo is a string that describes what you want to know about. The possible values of this string are numerous and ever-expanding.

The first possibility is the name of one of the standard MINC image dimensions: 'time', 'zspace', 'yspace', or 'xspace'. If these are supplied, getimageinfo will return the length of that dimension from the MINC file, or 0 if the dimension does not exist. Note that requesting 'time' is equivalent to requesting 'NumFrames'; also, the three spatial dimensions also have equivalences that are somewhat more complicated. For the case of transverse images, zspace is equivalent to NumSlices, yspace to ImageHeight, and xspace to ImageWidth. See the help for newimage (or the MINC standard documentation) for details on the relationship between image orientation (transverse, sagittal, or coronal) and the MINC spatial image dimensions.

The other possibilities for whatinfo, and what they cause getimageinfo to return, are as follows:

Filename - the name of the MINC file (if applicable) as supplied to openimage or newimage; will be empty if data set has no associated MINC file.

NumFrames - number of frames in the study, 0 if non-dynamic study (equivalent to 'time')

NumSlices - number of slices in the study (0 if no slice dimension)

ImageHeight - the size of the second-fastest varying spatial dimension in the MINC file. For transverse images, this is just the length of MIyspace. Also, when an image is displayed with viewimage, the dimension that is "vertical" on your display is the image height dimension. (Assuming viewimage is working correctly.)

ImageWidth - the size of the fastest varying spatial dimension, which is MIxspace for transverse images. When an image is displayed with viewimage, the image width is the horizontal dimension on your display.

ImageSize - a two-element vector containing ImageHeight and ImageWidth (in that order). Useful for viewing non-square images, because viewimage needs to know the image size in that case.

DimSizes - a four-element vector containing NumFrames, NumSlices, ImageHeight, and ImageWidth (in that order)

FrameLengths - vector with NumFrames elements - duration of each frame in the study, in seconds. This is simply the contents of the MINC variable 'time-width'; if this variable does not exist in the MINC file, then getimageinfo will return an empty matrix.

FrameTimes - vector with NumFrames elements - start time of each frame, relative to start of study, in seconds. This comes from the MINC variable 'time'; again, if this variable is not found, then getimageinfo will return an empty matrix.

MidFrameTimes - time at the middle of each frame (calculated by FrameTimes + FrameLengths/2) in seconds

If the requested data item is invalid or the image specified by handle is not found (ie. has not been opened), then the returned data will be an empty matrix. (You can test whether this is the case with the isempty() function.)

SEE ALSO openimage, newimage, getimages

GETIMAGES *Retrieve whole or partial images from an open MINC file.*

```
images = getimages (handle [, slices [, frames [, old_matrix ...
                    [, start_row [, num_rows]]]]])
```

reads whole or partial images from the MINC file specified by handle. Either slices or frames can be a vector (to specify a set of several images), but at least one of them must be a scalar – it is not possible to read images from both different slices and different frames at the same time. (Multiple calls to getimages will be needed for this.) If the file is non-dynamic (no time dimension), then the frames argument can be omitted or empty; likewise, if there is no slice dimension, the slices argument can be omitted or empty. (But note that slices must be given if any frames are to be specified – thus, it may be necessary to supply an empty matrix for slices in the unusual case of a MINC file with frames but no slice variation.)

The default behaviour of getimages is to read whole images and return them as MATLAB column vectors with the image rows stored sequentially. If multiple images are read, then they will be returned as the columns of a matrix. For instance, if 10 128x128 images are read, then getimages will return a 16384x10 matrix; to extract a single image, use MATLAB's colon operator, as in foo (:,1) to extract all rows of column 1 of the matrix foo.

To read partial images, you can specify a starting image row in start_row; if num_rows is not supplied and start_row is, then a single row is read.

To try to conserve memory use, you can "recycle" MATLAB matrices when sequentially calling getimages to read in identically-sized blocks of image data. This is done by simply passing your image matrix to getimages as old_matrix, eg:

img = []; for slice = 1:numslices img = getimages (handle, slice, 1:numframes, img); (process img) end

This will get around MATLAB's tendency to unnecessarily allocate new blocks of memory and leave old blocks unused.

EXAMPLES (assuming handle = openimage ('some_minc_file');)

To read in the first frame of the first slice: one_image = getimages (handle, 1, 1); To read in the first 10 frames of the first slice: first_10 = getimages (handle, 1, 1:10); To read in the first 10 slices of a non-dynamic (i.e. no frames) file: first_10 = getimages (handle, 1:10);

Note that there is currently no way to write partial images – this feature is provided in the hopes of cutting down memory usage due to intermediate calculations; you should pre-allocate a matrix large enough to hold your final results, and place them there as blocks of rows from the input MINC file are processed. Then, when all rows have been processed, a whole output image can be written to the output file.

GETMASK *returns a mask that is the same size as the passed image.*

```
mask = getmask (image)
```

The mask consists of 0's and 1's, and is created interactively by the user. Currently, a threshold algorithm is used, based on the input argument image: the user selects a threshold using a slider (the default starting value is 1.8), and getmask selects all points in image greater than the mean value of the entire image multiplied by threshold the threshold. It then displays image as masked by that threshold value, so the user can refine the threshold to his/her satisfaction.

GETPIXEL *replacement for MATLAB's ginput function*

```
[x,y] = getpixel(n)
```

MATLAB's ginput function crashes if there is no X display defined. This function checks to make sure that the display exists before calling ginput. The functionality of this function is exactly the same as MATLAB's ginput.

HOTMETAL *a better hot metal color map.*

```
map = hotmetal(num_colors)
```

HOTMETAL(M) returns an M-by-3 matrix containing a "hot" colormap. HOTMETAL, by itself, is the same length as the current colormap.

For example, to reset the colormap of the current figure:

```
colormap(hotmetal)
```

See also HSV, GRAY, PINK, HOT, COOL, BONE, COPPER, FLAG, COLORMAP, RGBPLOT, SPECTRAL.

MAKETAC *Make a time-activity curve*

```
tac = maketac(x,y,pet)
```

Generate a time-activity curve from a set of data.

MIINQUIRE *find out various things about a MINC file from MATLAB*

```
info = miinquire ('minc_file' [, 'option' [, 'item']], ...)
```

miinquire has a rather involved syntax, so pay attention. The first argument is always the name of a MINC file. Following the filename can come any number of "option sequences", which consist of the option (a string) followed by zero or more items (more strings).

Any number of option sequences can be included in a single call to miinquire, as long as enough output arguments are provided (this is checked mainly as a debugging aid to the user). Generally, each option results in a single output argument.

The currently available options are:

dimlength imagesize vartype

Options that will most likely be added in the near future are:

dimnames varnames vardims varatts atttype attvalue

One inconsistence with the standalone utility mincinfo (after which miinquire is modelled) is the absence of the option "varvalues". The functionality of this available in a superior way via the CMEX mireadvar.

EXAMPLES

[ImSize, NumFrames, ImType] = ... miinquire ('foobar.mnc', 'imagesize', ... 'dimlength', 'time', ... 'vartype', 'image');

puts the four-element vector of image dimension sizes into ImSize; the length of the time dimension into the scalar NumFrames; and the type of the image variable into the string ImType.

MIREADIMAGES *Read images from specified slice(s)/frame(s) of a MINC file.*

```
images = mireadimages ('minc_file'[, slices[, frames[, options]]])
```

opens the given MINC file, and attempts to read whole images from the slices and frames specified in the slices and frames vectors. For the case of 128 x 128 images, the images are returned as the the columns of a 16384-row matrix, with the highest image dimension varying the fastest. That is, if x is the highest image dimension, each contiguous block of 128 elements will correspond to one row of the image.

To manipulate a single image as a 128x128 matrix, it is necessary to extract the desired column (image), and then reshape it to the appropriate size. For example, to load all frames of slice 5, and then extract frame 7 of the file foobar.mnc:

```
>> images = mireadimages ('foobar.mnc', 4, 0:20);
>> frame7 = images (:, 7);
>> frame7 = reshape (frame7, 128, 128);
```

Note that mireadimages expects slice and frame numbers to be zero- based! Thus, frames 0 .. 20 of the MINC file are read into columns 1 .. 21 of the matrix images.

For most dynamic analyses, it will also be necessary to extract the frame timing data. This can be done using MIREADVAR.

Currently, only one of the vectors slices or frames can contain multiple elements.

MIREADVAR *Read a hyperslab of data from any variable in a MINC file.*

```
data = mireadvars ('MINC_file', 'var_name', [, start, count[, options]])
```

Given vectors describing the starting corner (zero-based!) and edge lengths, mireadvars reads an n-dimensional hyperslab from a MINC (or NetCDF) file. The data is returned as a MATLAB vector, with the highest dimension of the variable changing fastest.

The simplest (and intended) use of mireadvars is to read an entire one-dimensional variable. For example:

time = mireadvars ('foobar.mnc', 'time');

will read the entire contents of the variable 'time' from the file foobar.mnc. (If the start and count vectors are not given, they default to reading the entire variable. Currently, if start is given, count must be given, and they must each have exactly one element per dimension.)

A more complicated example is to use mireadvar as a low-rent substitute for mireadimages. For example, to read slice 5, frame 7 (note that these are 1-based, and mireadvar expects 0-based indeces!) of foobar.mnc:

image = mireadvar ('foobar.mnc', 'image', [6 4 0 0], [1 1 128 128]);

The disadvantages of this approach are numerous. First of all, mireadimages will perform the scaling and shifting necessary to transform the image data from scaled bytes or shorts (or however it happens to be stored in the MINC file) to floating point values representing the actual physical data. Second, with mireadvar you must know the exact order of the dimensions: in the above example, "slice 5" corresponds to the 4 (note zero-based!) at position 2 of the Start vector, and

"frame 7" is the 6. Also, you must know the size of the image; mireadimages will handle anything, not just 128x128. Finally, mireadimages provides greater flexibility with respect to slice and frame selection. With mireadvar, you can only read contiguous ranges of slices and frames, and you must figure out the start and count values for each dimension yourself. Mireadimages, however, does all that work for you given just slice and frame numbers.

MIWRITEIMAGES *write images to a MINC file*

```
miwriteimages (filename, images, slices, frames)
```

writes images (in the format as returned by mireadimages) to a MINC file. The MINC file must already exist and must have room for the data. slices and frames only tell miwriteimages where to put the data in the MINC file, they are not used to select certain columns from images.

Also, the slices and frames must be valid and consistent with the MINC file, which must exist and have an image variable in it. The number of images to write (implied by the number of elements in slices or frames) must be the same as the number of columns in the matrix images. Since miwriteimages only expects to be called by putimages, none of these requirements are checked here – all that is done by putimages.

Note that there is also a standalone executable miwriteimages; this is called by miwriteimages.m via a shell escape. Neither of these programs are meant for everyday use by the end user.

NCONV *Convolution of two vectors with not necessarily unit spacing.*

```
C = NCONV(A, B, spacing) convolves vectors A and B.  The resulting
vector is length LENGTH(A)+LENGTH(B)-1.
```

This routine is a replacement for MathWorks' conv function, which implicitly assumes that A and B are sampled with unit spacing. If you are dealing with two functions that are unevenly sampled or sampled with different spacings, one or both of them must be resampled to the same evenly spaced independent variable. Then, if the spacing of the independent variable is not 1, it should be passed to nconv.

See also CONV, XCORR, DECONV, CONV2, LOOKUP.

NEWIMAGE *create a new MINC file, possibly descended from an old one*

```
handle = newimage (NewFile, DimSizes, ParentFile, ...
                   ImageType, ValidRange, DimOrder)
```

creates a new MINC file. NewFile and DimSizes must always be given, although the number of elements required in DimSizes varies depending on whether ParentFile is given (see below). All other parameter are optional, and, if they are not included or are empty, default to values sensible for PET studies at the MNI.

The optional arguments are:

ParentFile - the name of an already existing MINC file. If this is given, then a number of items are inherited from the parent file and included in the new file; note that this can possibly change the defaults of all following optional arguments.

DimSizes - a vector containing the lengths of the image dimensions. If ParentFile is not given, then all four image dimensions (in the order frames, slices, height, and width) must be specified. Either or both of frames and slices may be zero, in which case the corresponding MINC dimension (MItime for frames, and one of MIzspace, MIyspace, or MIxspace for slices) will not be created. If ParentFile is given, then only the number of frames and slices are required; if the height and width are not given, they will default to the height/width of the parent MINC file. In no case can the height or width be zero – these two dimensions must always exist in a MINC file. See below, under "DimOrder", for details on how slices, width, and height are mapped to MIzspace, MIyspace, and MIxspace for the various conventional image viewpoints.

ImageType - a string, containing a C-like type dictating how the image is to be stored. Currently, this may be one of 'byte', 'short', 'long', 'float', or 'double'; plans are afoot to add 'signed' and 'unsigned' options for the three integer types. Currently, 'byte' images will be unsigned and 'short' and 'long' images will be signed. If this option is empty or not supplied, it will default to 'byte'. NOTE: this parameter is currently ignored.

ValidRange - a two-element vector describing the range of possible values (which of course depends on ImageType). If not provided, ValidRange defaults to the maximum range of ImageType, eg. [0 255] for byte, [-32768 32767] for short, etc. NOTE: this parameter is currently ignored.

DimOrder - a string describing the orientation of the images, one of 'transverse' (the default), 'sagittal', or 'coronal'. Transverse images are the default if DimOrder is not supplied. Recall that in the MINC standard, zspace, yspace, and xspace all have definite meanings with respect to the patient: z increases from inferior to superior, x from left to right, and y from posterior to anterior. However, the concepts of slices, width, and height are relative to a set of images, and the three possible image orientations each define a mapping from slices/width/height to zspace/yspace/xspace as follows:

Orientation Slice dim Height dim Width dim transverse MIzspace MIyspace MIxspace sagittal MIxspace MIzspace MIyspace coronal MIyspace MIzspace MIxspace

---

NFRAMEINT *integrate a function across a range of intervals (frames)*

```
integrals = nframeint (ts, y, FrameStarts, FrameLengths)
```

calculates the integrals of a function (represented as a set of points in y, sampled at the time points in ts) across each of a set of frames which are given by their start times and lengths. The integral is then normalised so that nframeint returns the average value of y (as a function of ts) across each frame.

ts and y must be vectors of the same length, as must FrameStarts and FrameLengths. Normally, ts and y are a good deal longer than FrameStarts and FrameLengths in order to get reasonably accurate results. The returned variable, integrals, will be a vector of the same length of FrameStarts and FrameLengths, containing the integral of y(ts) across each frame.

Points of y to integrate for each frame are selected by finding all points of ts that are greater than the frame start time and less than the frame stop time. If possible, y is then linearly interpolated at the frame start and stop times to form a closed interval. Then, a trapezoidal integration across those points is calculated, and the integral is divided by the width of the interval across which y is known within the frame. Normally, this will simply be the length of the frame. However, it may be that the lowest value of ts is greater than the frame start or the highest value of ts is lower than the frame stop time. In these cases, y is not known outside of the frame, and cannot be resampled at the frame endpoints; so, the integration will only be performed across known points, and the integral will be divided not by the length of the frame but by the width of the interval across which y is known.

OPENIMAGE *setup appropriate variables in MATLAB for reading a MINC file*

```
handle = openimage (filename)
```

Sets up a MINC file and prepares for reading. This function creates all variables required by subsequent get/put functions such as getimages and putimages. It also reads in various data about the size and number of images on the file, all of which can be queried via getimageinfo.

The value returned by openimage is a handle to be passed to getimages, putimages, getimageinfo, etc.

PUTIMAGES *Writes whole images to an open MINC file.*

```
putimages (handle, images [, slices [, frames]])
```

writes images (a matrix with each column containing a whole image) to the MINC file specified by handle, at the slices/frames specfied by the slices/frames vectors.

Note that only one of the vectors slices or frames may have multiple elements; ie., you may not write multiple slices and multiple frames simultaneously. (This should not be a problem, since you cannot *read* multiple frames and slices simultaneously either.) If both slices and frames are present in the MINC file, then both slices and frames vectors must be supplied and be non-empty. If either of those dimensions are not present, though, then the associated vector must be either omitted or empty.

EXAMPLES

To write zeros to an entire slice (say, 21 frames of slice 7) of a full dynamic MINC file with 128x128 images [already opened with handle = newimage (...)]:

images = zeros (16384,21); putimages (handle, images, 7, 1:21);

To write random data to a single slice (7) of a non-dynamic file (again 128x128 images):

image = rand (16384, 1); putimages (handle, image, 7);

SEE ALSO newimage, openimage, getimages

RESAMPLEBLOOD *resample the blood activity in some new time domain*

```
[new_g, new_ts] = resampleblood (handle, type[, samples])
```

reads the blood activity and sample timing data from the study specified by handle, and resamples the activity data at times specified by the string type. Currently, type can be one of 'even' or 'frame'. For 'even', a new, evenly-spaced set of times will be generated and used as the resampling times. For 'frame', the mid frame times will be used. In either case, the resampled blood activity is returned as new_g, and the times used are returned as new_ts.

The optional argument samples specifies the number of samples to take. If it is not supplied, resampleblood will resample the blood data at roughly 0.5 second intervals.

SMOOTH *do a simple spatial smoothing on an image*

```
new_data = smooth (old_data)
```

Smooths a two-dimensional image by averaging over a circle with a diameter of 5 pixels.

SPECTRAL *Black-purple-blue-green-yellow-red-white color map.*

```
map = spectral(num_colors)
```

SPECTRAL(M) returns an M-by-3 matrix containing a "spectral" colormap. SPECTRAL, by itself, is the same length as the current colormap.

For example, to reset the colormap of the current figure:

colormap(spectral)

See also HSV, GRAY, PINK, HOT, COOL, BONE, COPPER, FLAG, COLORMAP, RGB-PLOT.

TEMPFILENAME *generate a unique temporary filename*

```
fname = tempfilename
```

Requires that a directory /tmp/ exists on the current machine.

VIEWIMAGE *displays a PET image from a vector or square matrix.*

```
[fig_handle, image_handle, bar_handle] = viewimage (img, colourbar_flag)
```

viewimage (img) sets the colourmap to spectral (the standard PET colourmap) and uses MATLAB's image function to display the image. Works on either SGI's or Xterminals, with colours dithered to black and white on the Xterms.

Images are scaled so its high points are white and its low points black.

viewimage (img, colourbar_flag) turns the colourbar on or off. The default is on, but by specifying colourbar flag = 0, the colourbar will be turned off.
