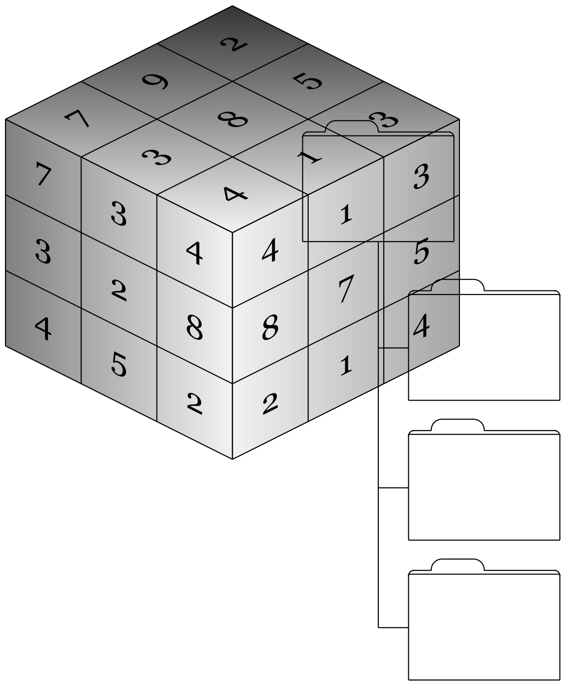

Waterloo File and Matrix Utilities

Project Waterloo classes and functions for partial i/o with huge data sets in MATLAB®

Malcolm Lidierth
Wolfson Centre for Age-Related Diseases

http://sigtool.sourceforge.net/

Revised: 14th November 2011

Acknowlegements:

MATLAB code in this document was styled using Florian Knorn's M-code LaTeXPackage.
`http://www.mathworks.com/matlabcentral/fileexchange/8015-m-code-latex-package`

The document was prepared using Pascal Brachet's TeXMaker
`http://www.xm1math.net/texmaker/`

Stefan Kottwitz for the Tikz code for the soduko cube that has been modified here. For the original see
`http://www.texample.net/tikz/examples/sudoku-3d-cube/`

Special thanks to:
AccelerEyes for providing Jacket software in support of this project
`http://www.accelereyes.com/`

Contents

The problem

What can you do when:

1. a variable is too large for the MATLAB workspace?

2. many variables are needed simultaneously, but their summed size is too large for the workspace?

3. only a subset of data are required from a large variable saved on disc

MATLAB provides few tools to help. The builtin *save* and *load* commands deal minimally with whole variables. When data are embedded in a structure, the entire structure needs to loaded: you can not use *load* to access a specific field of the structure.

In each of the examples above, using partial loading of data can substantially improve performance - lowering demands on the MATLAB workspace and, used in the right circumstances, can also dramatically increase the speed of code execution. This library provides features for partial i/o and for handling potentially huge data sets in MATLAB.

Benefits of partial i/o

To illustrate, take the example of a 3D matrix, perhaps a 128x128x8192 uint8 image stack, as shown in Figure 1. We will keep the matrix small enough to fit in the MATLAB workspace to allow comparison with using the MATLAB *load* command below.

Figure 2, illustrates how time is divided between

1. loading the image stack using the MATLAB *load* command.

2. extracting a series of consecutive frames from the loaded stack.

Loading the data set takes roughly 0.2s and accounts for most of the time regardless of how many image frames are subsequently extracted from the matrix. It is clear that attempts to improve performance will need to shift the intercept of this line downwards, i.e. to avoid loading the entire data set. Partial i/o does that.

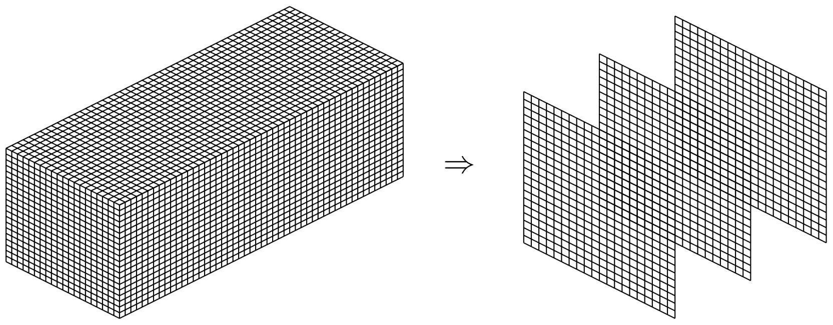

Figure 1: A large 3D matrix length (left) is stored on disc. The methods in this library allow submatrices to be loaded on-demand into the MATLAB workspace (right) without loading the entire matrix. This saves memory and increases speed as demonstrated below.

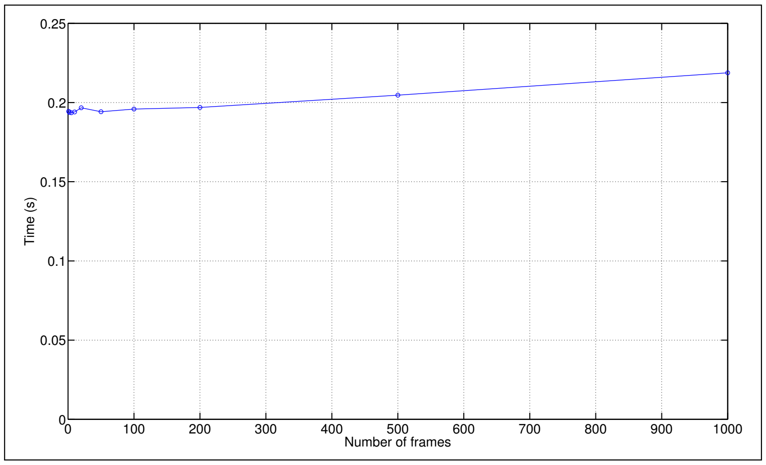

Figure 2: Using the load command to access a MAT-file containing a 128x128x8192 image stack. The tests were run extracting 1,2,5...1000 frames and the plotted points are the averages of 10 test runs for each condition.

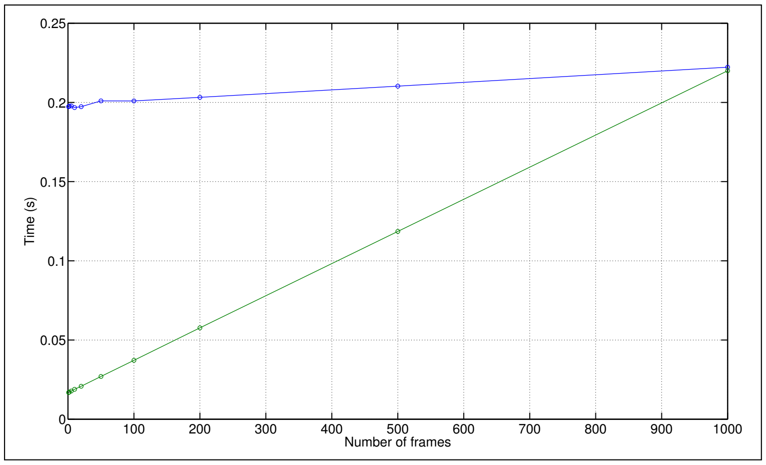

Figure 3: Comparing the speed of using the MATLAB load command and partial i/o using fread from a MAT-file containing a 128x128x8192 image stack. A variable number of consecutive images (n=1,2,5...1000) were extracted using load (○—○) or a modified ReadCubeSlice that loaded mutiple frames in a loop (○—○) .

One of The MathWorks staffers, Oren Rosen, has posted a MATLAB File Exchange contribution that illustrates the benefits of partial i/o[1]. Oren's *ReadCubeSlice* function used the *where* function from the library described here to get the required details about disc offsets, data types etc within a MAT-file. With this information, Oren was able to perform partial i/o and load only the required image from the stack using MATLAB's low-level *fread* function. Figure 3 shows the speed improvement for the tests used previously in Fig. 2. For low numbers of loaded frames this gives a performance boost of ∼10 fold.

The *ReadCubeSlice* code was designed to load only a single frame at a time: it was intended as a demonstration of principle only. For the tests shown in Fig. 3, mutiple frames were loaded in a loop. That is not optimal and performs poorly for large numbers of image frames. To get better performance, we need to reduce the slope of the line in Fig. 3 so that low-level i/o performs better even with relatively large numbers of frames being loaded. The methods of the *nmatrix* class in this library do this: they load multiple frames through a single call to *fread*. The red line in Figure 4

[1]`http://www.mathworks.co.uk/matlabcentral/fileexchange/17992`

shows the performance gain. Using a single call to fread reduces the slope again so we now have a substantial speed improvement even when loading 1000 frames at a time.

The *nmatrix* class also generalizes the approach used in *ReadCubeSlice*: while *ReadCubeSlice* allows a plane to be along the in *x*, *y* or *z* axes of a 3D matrix, nmatrix supports extraction from any dimension of an N-D matrix.

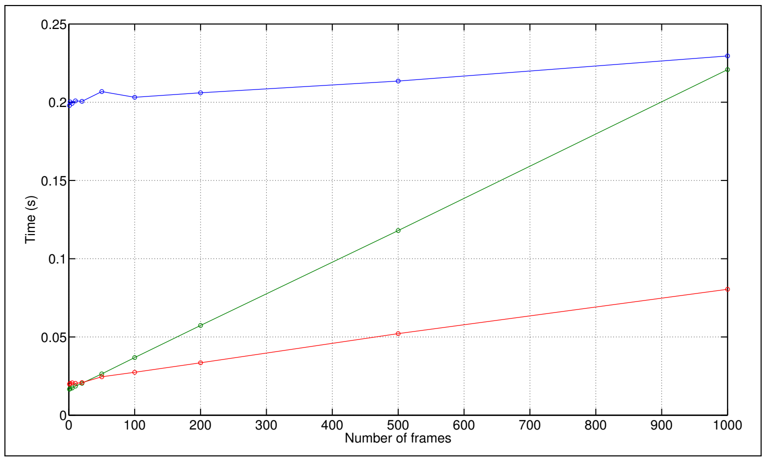

Figure 4: Comparing the speed of using the MATLAB load command and partial i/o using ReadCubeSlice or the nmatrix class from a MAT-file containing a 128x128x8192 image stack. A variable number of consecutive images (n=1,2,5...1000) were extracted using load (○—○), ReadCubeSlice (○—○), or the nmatrix class described in this document using low-level i/o (○—○) .

Partial i/o with random access

In the tests above, the data accessed were stored in a consecutive block of the file. When data are not in a consecutive block the *nmatrix* class methods switch to using a loop but this hits performance. One solution is to map the data from disc through the system's virtual memory. The *nmatrix* class supports this: you have a choice of using low-level i/o or memory mapping and you can switch dynamically. Figure 5 shows a variation of the test shown in Fig. 4. Here, the frames were chosen at random then loaded through a single call to the *nmatrix* subsref method. Note the change of x-axis range in Fig. 5: it covers loading up to 8000 frames, i.e. nearly all of those available in the file.

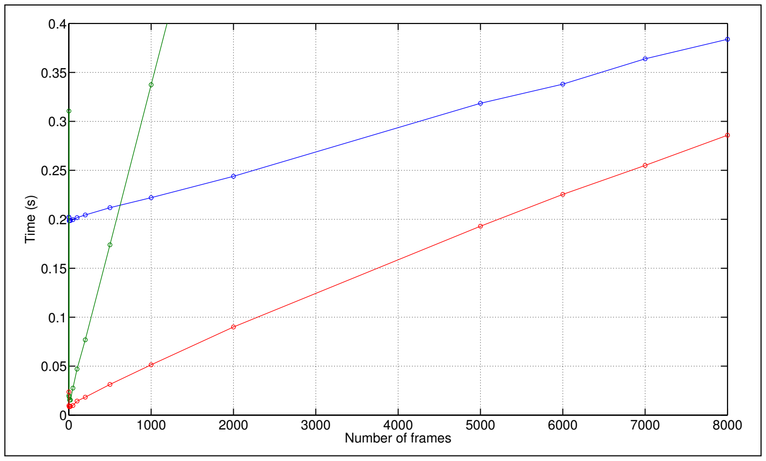

Figure 5: Repeat of the test shown in Figure 2 but with frames from the image stack chosen randomly rather than in a contiguous block. The MATLAB load command behaves much as before(○—○). The nmatrix class with low-level i/o performs poorly except for very low n (○—○). Switching to virtual memory provides a substantial performance boost (○—○).

For truly random access to the elements of the data set, the *nmatrix* class methods in low-level i/o mode resort to reading a block of data, then extracting the required elements from that. While this will often still reduce MATLAB workspace use, it will not generally be fast enough. In that case memory mapping is better: the *nmatrix* class allows both low-level access to the data on file and access via MATLAB *memmapfile* class.

Figure 6 shows that result of running a test using linear indexing into the 128x128x8192 image stack above using randomly generated linear indices. Tests returned 100 to 2x10^6 elements as indicated. The *nmatrix* class performance is dependent on the number of elements returned but invariably outperformed the *load* command by ~1.5x to ~8.5x for low *n*.

Thus far, the *nmatrix* class has been used as a simple wrapper for the virtual memory support offered in MATLAB's *memmapfile* class. If preferred, the *memmapfile* can be used directly. A *getMap* function described within provides a convenience method for obtaining *memmapfile* objects for data in MAT-files. As shown in Fig. 7, there is little performance difference between the *memmapfile* and *nmatrix* class used in virtual memory

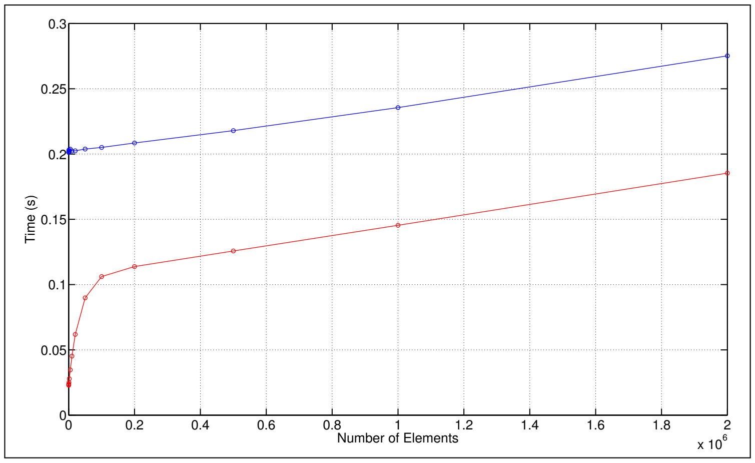

Figure 6: Linear indexing into the image stack using randomly generated indices. The returned data block had 100...2x10^6 *elements as indicated on the abscissa. As before, the blue line shows the time taken using load. The red line shows the data accessed using the nmatrix class and virtual memory.*

mode; however, the *nmatrix* class methods add additional features - the ability to switch to low-level i/o among them.

Finally, Fig. 8 compares the performance of the *nmatrix* class when using virtual memory or low-level i/o. Virtual memory always provided better performance and is therefore likely to be preferred wherever sufficient virtual memory is available.

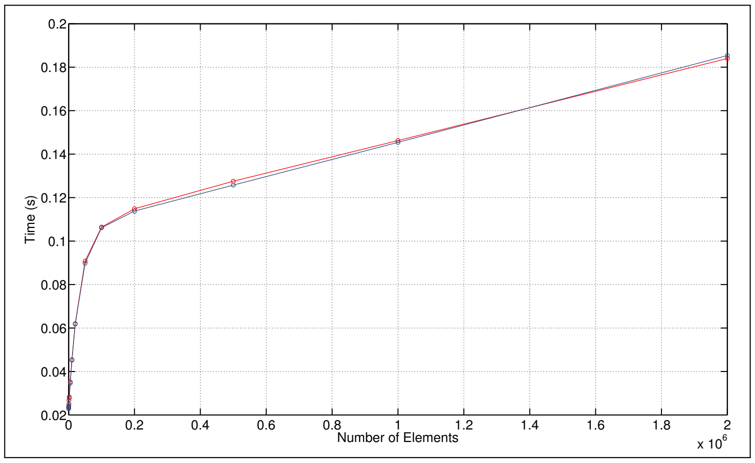

Figure 7: Comparison of nmatrix (○—○) and memmapfile (○—○) classes for inear indexing into the image stack using randomly generated indices via virtual memory.

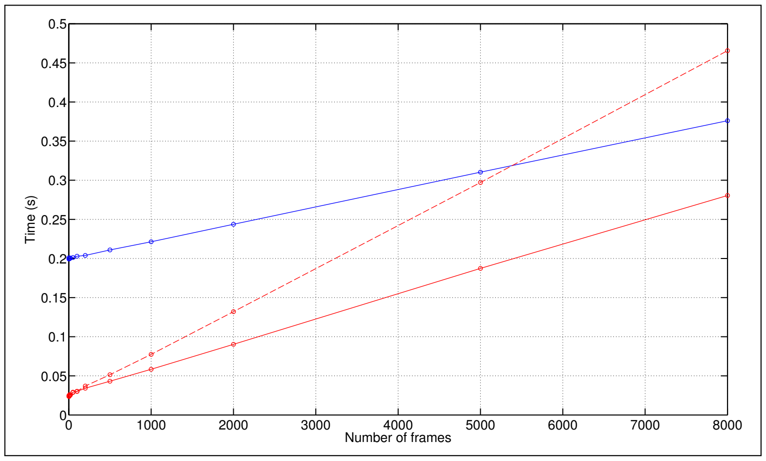

Figure 8: Comparison of nmatrix performance using virtual memory (○—○) and fread (○- - ○) for frame based access to contiguous memory blocks. For reference, the performance using the MATLAB load command is also included (○—○).

Example applications

Dealing with a single huge variable

The image stack example above used a relatively small matrix so we could compare performance with the MATLAB *load* command (an image stack small enough for the MATLAB workspace was needed). If we add more images to the stack it would make a better test of the partial i/o. For the example in Fig. 9, the stack was extended to give a matrix of 128x128x196608 (i.e. 3.2Gb) and the tests of Fig. 4 were repeated using low-level i/o and virtual memory via the *nmatrix* class. For these tests, the data were saved to a standard binary file as the version 6 MAT-file used before has a 2Gb limit. The data shown are mean ± S.D. for 10 trials in each condition.

Using virtual memory (○—○) outperformed low-level i/o (○- - ○), with the difference becoming more significant as the number of loaded frames increased. The increased scatter associated with low-level i/o is associated with the initial *fseek* performed to locate the starting point in the file (recall that *n* consecutive frames are being read with the first being chosen at random from those available).

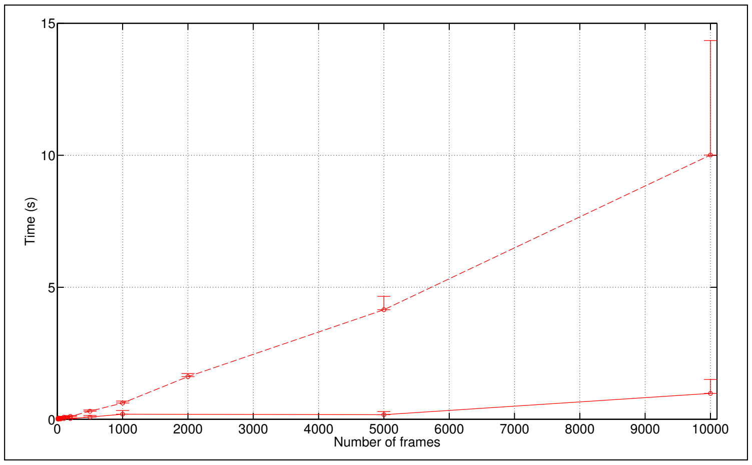

Figure 9: Comparing the time taken to load a consectutive block of images from a 3.2Gb image stack using low-level i/o (○- - ○) and virtual memory (○—○) via the nmatrix class.

The poorer performance of low-level i/o with very large data sets means that virtual memory mapping will often be preferred. If there are insufficient system resources to memory map the entire data set the *getPartialMap* function is available. *getPartialMap* allows partial i/o from an already selected part of a data set. It requires virtual memory to be allocated only for the fraction of the data set that is represented. *getPartialMap* is described fully below.

Dealing with many variables

Another case is where individual variables may or may not be too large for the MATLAB workspace but simultaneous access is needed to many of them - with the sum of their sizes being too great. Dealing with this situation was the main reason behind developing the *nmatrix* class for the sigTOOL Project.

The *nmatrix* class supports vitrual memory mapping and low-level access to data files. It also includes methods that allow virtual memory to be allocated and deallocated on-the-fly. For a data file with perhaps hundreds of signal channels, this allows virtual memory to be managed according to the available resources and, for the sigTOOL program, this is done in the background without the user needing to manage the memory actively.

The *nmatrix* class also allows the data from disc to be loaded into an *nmatrix* instance in RAM. As access to a primitive data type will always outperform disc-based access this can be used to speed up processing when system resources allow it. As the data are accessed via the *nmatrix* methods, the same code can be used regardless of whether the data are in RAM, memory mapped or accessed by low-level i/o.

Additionally, the *nmatrix* deceives MATLAB: it pretends to be a primitive data type so instances of *nmatrix* can often be passed to existing MATLAB m-files that expect a primitive data type on input: including most standard MATLAB toolbox functions.

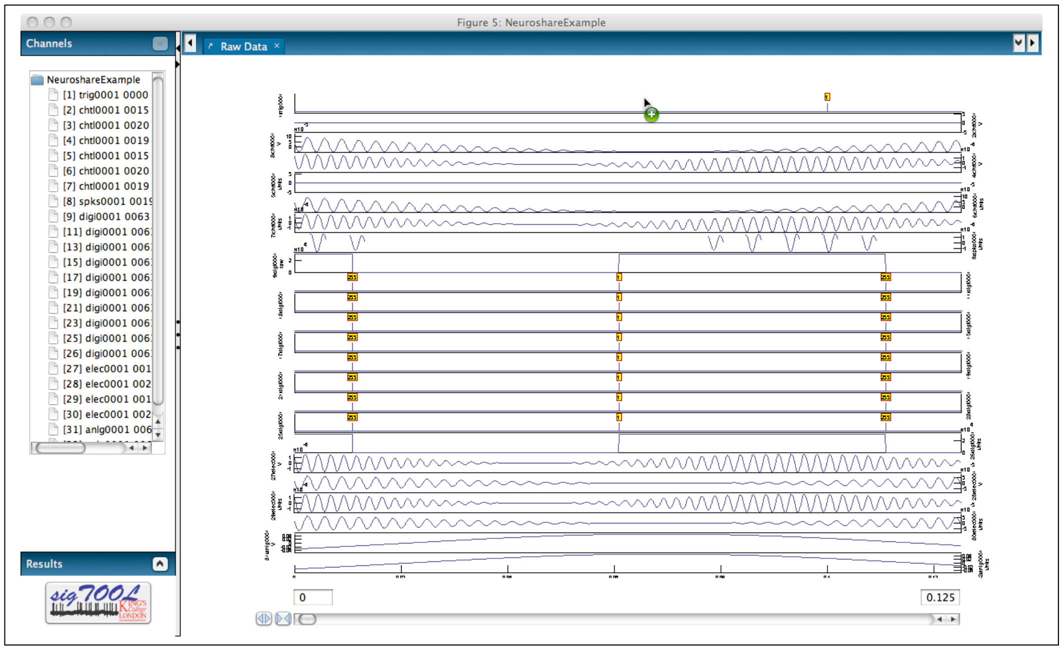

Figure 10: Multiple channels displayed in a sigTOOL data view. Data are mapped to the MATLAB workspace through virtual memory which is managed dynamically by the package. Data shown are from a sample file from MultiChannel Systems GmbH

A note on file formats

MATLAB's MAT-files do not use a single format. The file formats are classified by

1. *Level* the MATLAB version in which the basic format was introduced
2. *Version* which gives the MATLAB version required to read the file

Currently, three Levels are supported: 4, 5 and 7. Level 4 is a legacy format not described further here. The 'Version' level used in the MATLAB save command is probably more familiar to most users. These map as follows

save option	Level	Version	Comments	Documented
-v6	5	6		✓
-v7	5	7	same as -v6 but with unicode + gzip compression	✓
-v7.3	7	7.3	HDF5-based	✗

Level 5 MAT-files (Version 6 and 7 files)

The Level 5 format is a proprietary hierarchical file format fully documented by The MathWorks[2]. Tools are available to load these files in other languages e.g. Octave[3] , Python[4] and R[5].

To save a MAT-file using these formats specify the '-v6' or '-v7' option with the MATLAB *save* command. At the time of writing, MATLAB ships with Version 7 set as the default format in MATLAB desktop preferences. If you want to use this library often, you might prefer to set that to Version 6.

The use of *gzip* compression in Version 7 files is the problem for using it with this library. Compression can substantially reduce file size and, in theory at least, can speed up data access because the added overhead of decompressing the data may be small compared to the time saved in loading

[2]http://www.mathworks.co.uk/help/pdf_doc/matlab/matfile_format.pdf

[3]http://www.gnu.org/software/octave/doc/interpreter/Simple-File-I_002fO.html

[4]http://docs.scipy.org/doc/scipy/reference/generated/scipy.io.loadmat.html

[5]http://cran.r-project.org/web/packages/R.matlab/R.matlab.pdf

data from disc. For the author's code, use of the Version 7 format invariably slows down processing. Looking at MATLAB Answers[6] and the Newgroup[7] shows that others experience this too.

Level 7 MAT-files (Version 7.3 files)

Version 7.3 files were introduced in 2006. Version 7.3 files are "HDF5-based": they employ the widely used and documented Hierarchical Data Format 5 system. This means that tools from the HDF Group[8] can be used to inspect the files' structure and contents. Unfortunately, The MathWorks support staff say that the Version 7.3 format "...is not pure HDF5 however and we never claimed that these files will be readable by any HDF5 libraries".

As of MATLAB R2011b, support for partial i/o from Version 7.3 files has been added through the MATLAB *MatFile* class. On R2011b+, the *nmatrix* class included in this library can wrap a *MatFile* instance in much the same way as a *memmapfile* instance to provide support for Version 7.3 files. However, the *matfile* class does not presently support the full range of MATLAB indexing options: linear indexing and logical indexing are not supported and for N-D matrices, subscripts must be specified for all dimensions and ascend evenly. Perhaps these limitations will be removed in later releases.

The Version 7.3 format appears not to be supported in Octave, R etc. and has not been documented by MATLAB. If you are developing code for distribution you might prefer to use an earlier MAT-file version or a pure HDF5 format (see below).

HDF5 files

The design of the HDF5 format was initiated by the National Center for Supercomputing Applications in the USA[9] and its development is now supported ny the HDF Group[10]. MATLAB provides two sets of tools for writing HDF5 files:

[6]urlhttp://www.mathworks.co.uk/matlabcentral/answers/

[7]`http://www.mathworks.co.uk/matlabcentral/newsreader/`

[8]`http://www.hdfgroup.org/`

[9]`http://www.ncsa.illinois.edu/`

[10]`http://www.hdfgroup.org/`

1. a set of high-level functions h5read, h5write etc and
2. a set of low-level functions that provide access to the HDF5 API.

Examples of using the low-level functions in many languages, including MATLAB, can be found at

```
http://www.hdfgroup.org/ftp/HDF5/examples/examples-by-api/
```

The functions in this library support access to HDF5 files subject to the constraints that

1. The required data set is not chunked
2. The data are not compressed
3. The data space for the set has been fully allocated in the file

Better support for HDF5 files making use of the HDF5 low-level libraries might be included in a future version.

Custom binary file formats

The *nmatrix* class can be used with any custom binary format if you are able to provide a *memmapfile* object for the required data set in the file i.e. you know the byte offset, primitive class and shape of the saved matrix[11] .

MATLAB support for partial i/o

In MATLAB R2011b, support for partial i/o was added through a *matfile* function which returns a *matlab.io.MatFile* object. These are really only useful when dealing with MAT-files of Version 7.3 format.

Since 2005 (MATLAB 7), *memmapfile* objects have been supported in MATLAB. These allow partial i/o to the MATLAB workspace via virtual memory. MATLAB provides no tools to construct *memmapfile* objects for data stored in its own MAT-files. This library does, thus allowing easy use of virtual memory support with Version 6 MAT-files.

The syntax to access data in *matlab.io.MatFile* or *memmapfile* objects differ between each other and are different to that needed to access data

[11]For shape, remember that MATLAB uses Fortran-style, column-major ordering.

stored in an ordinary MATLAB matrix. To make full use of these features, you are likely to need to re-write your code. To avoid this, the *nmatrix* class has been developed here. An instance of this class can contain a *matlab.io.MatFile* or a *memmapfile* instance and, as described above, can also support low-level i/o. The syntax to access data in an *nmatrix* instance is the same as that to access a MATLAB matrix: there should, therefore, be no need to customise code to use them. Providing a consistent API in this way allows *matlab.io.MatFile*, *memmapfile* and standard matrices to be used as required without modifying code. To achieve this, the *nmatrix* class effectively deceives MATLAB into believing that it is a primitive data type. Note that this deception is suported only in m-code: *nmatrix* instances can not be passed as inputs to mex-files or Java code.

What this library adds

This library was designed to work with Version 6 MAT-files. Workarounds are included to allow Version 7 and 7.3 files to be used also but, in general, it will be better to transfer data to the Version 6 format. The library has three parts:

1. *The original MAT-file Utilities*

   These have been on the MATLAB FEX for some years and provide extended support for reading and writing to Version 6 MAT-files. The *where* function is part of these utilities and provides the information needed to memory map a data set. In addition, functions are provided to append data to the final variable in a Version 6 MAT-file permitting data files to be written in stages. These functions are fully described in

2. *Convenience functions*

   These functions provide support for memory mapping Version 6 MAT-files without calling the *where* function explicitly. Version 7 and 7.3 files are supported via a workaround that extracts the required data set to a temporary Version 6 file, then maps the data from that. Functions for translating between MATLAB MAT-file formats are provided also and, in general, using these to create a file in the Version 6 format will be preferred to the workarouund above.

3. *The nakhur classes*

The *nakhur* superclass is implemented by subclassing in the *nmatrix* and *adcarray* classes which extend and update the *adcarray* class that has been on the MATLAB FEX for some years. The new classes use R2008+ MATLAB class syntax and overload the MATLAB *handle* class. The nakhur classes make use of some of the MAT-file Utilities but present them in a user-friendly way. Users do not need any knowlege of the MAT-file format to use these classes.

The *nakhur* superclass provides easy-to-use constructors and a consistent API regardless of whether data are accessed through low-level i/o, memory mapping or via the *matlab.io.MatFile*. They also emulate the API of primitive matrices in MATLAB which means they should be able to slot into existing code without editing. To achieve that, standard MATLAB methods are overloaded to return non-standard results.

Convenience Functions

getMap function

v6✓ v7✓
v7.3✓ HDF5✓

getMap returns a *memmapfile* object for a variable in the specified file. Construction is easy. For example, with data in a MAT-file or an HDF5 file, just construct a *memmapfile* instance with:

```
myMap=getMap(filename, variablename);
```

where *filename* is a string giving the fully qualified file name and *variablename* is the name of the required variable in that file.

Data will always be represented through the *Data.Adc* field of the resulting *memmapfile* object.

variablename should be a string describing a vector or matrix of a primitive data type such as double, single, uint8, logical etc. (or an HDF5 equivalent). The data should be real-valued, and non-sparse.

Structures and objects are supported: just give the path to the matrix e.g.

```
myMap=getMap(filename,
    '/structname/fieldname1/fieldname2/...');
```

The file format will normally be assumed from the file extension. To force the use of a specific format, specify it at construction e.g.:

```
myMap=getMap('myfile.dat', '/x', 'mat');
```

Notes:

1. ✓Version 6 MAT-files are fully supported

2. ✓Version 7 and 7.3 MAT files are supported by copying but you must explicitly set the `usecopy_always` flag e.g

   ```
   myMap=getMap('myfile.mat', '/x', [], ...
       'usecopy_always');
   ```

   A temporary Version 6 MAT-file containing the required data set will be created and mapped[12] . The user is responsible for deleting this as required.

[12]in the folder returned by the MATLAB *tempdir* function

3. ✓HDF5-files are supported if the accessed data set is not chunked or compressed. If data can not be mapped, an exception will be thrown explaining why.

getPartialMap function

v6✓ v7✗ v7.3✗ HDF5✓ Binary✓

getPartialMap creates a *memmapfile* intance representing only part of a data set

Examples:

```
pmap=getPartialMap(filename, dataset, start, stop);
pmap=getPartialMap(memmapfileobject, start, stop);
```

1. For a vector: *start* and *stop* are the first and last elements

2. For a matrix *start* and *stop* are the linear indices into the last dimension. Thus, for a 2D matrix *start* and *stop* would be the first and last column to map. For a 4D matrix of dimension [m, n, p, q], q(start) to q(stop) would be mapped.

Data will always be represented through the *Data.Adc* field of the resulting *memmapfile* object.

Indexing into the resulting partial map begins at 1 (not *start*) so for a data set representing a 128x128x3x1000 matrix created with

```
pmap=getPartialMap(filename, dataset, 500, 25);
```

`pmap.Data.Adc(:,:,:,1)` would return the data for `dataset (:,:,:,500)` .

If required, partial maps can be wrapped in a nakhur subclass such as *nmatrix*, e.g.:

```
x=nmatrix(pmap);
```

Low-level i/o will then be supported

Partial maps can also be created using an existing *memmapfile* object.

```
x=getPartialMap(memmapfileobject, start, stop);
```

Note that the *memmapfile* constructor does not allocate virtual memory until an instance is accessed. As long as this is the case, only space for the partial map will be allocated here (and only when the partial map is accessed).

Notes:

1. ✓Version 6 MAT-files are fully supported
2. ✓Binary files are fully supported if you supply a *memmapfile* instance as input
3. ✓Version 7 and 7.3 MAT files are not directly supported but you can use a *memmapfile* object from *getMap*

```
x=getPartialMap(getMap(filename, dataset), ...
    start, stop);
```

4. ✓HDF5-files are supported if the accessed data set is not chunked or compressed. If data can not be mapped, an exception will be thrown explaining why.

copyTo functions

A set of *copyTo* functions copy existing MAT-files to files of a specified format. Using *copyToV6* is likely to be useful of you want to make much use of this library with existing data files.

Three functions are provided. All call a private *copyToVersion* function that does the work. Variables are read from the source file and written to the output file individually. These functions are *copyToV6*, *copyToV7* and *copyToV73* which copy data to Version 6, 7 and 7.3 formats respectively. The output files are written to a subfolder named 'V6', 'V7' or 'V73' as appropriate in parent folder of the source file.

Calling formats are the same for all function. For copyToV6:

1. *copyToV6(filename)* Copies the specified file
2. *copyToV6(foldername)* Copies all files in the specified folder
3. *copyToV6(cellarray)* Copies all files for each file/folder entry in the cell array

The nakhur superclass

The *nakhur* class is a superclass that has no constructors. You do not instantiate a *nakhur* instance directly. Instead you call the constructor for one of its subclasses. There are presently two of these:

1. *nmatrix* which is a general-purpose class for representing MATLAB matrices stored on disc.

2. *adcarray* which is an extension to *nmatrix* for representing data stored as integer values on disc and typically derived from an analogue-to-digital convertor, camera etc. The *adcarray* was developed for a specific purpose and is included here primarily for backwards compatability with the *adcarray* class that has been on the MATLAB FEX for some years. The *adcarray* class was developed for the sigTOOL Project. It is likely that users developing code from scratch will prefer to use the *nmatrix*.

The *nmatrix* class is versatile and will meet most users needs. Additional subclasses can be added by extending the *nakhur* superclass or the *nmatrix* subclass.

What the nakhur superclass adds

1. Consistent API

   Access to nakhur objects uses a consistent syntax regardless of whether the underlying mechanism uses low-level i/o, *memmapfile* or *matlab.io.MatFile* objects.

2. Delaying memory allocation

   Virtual memory is allocated to a *memmapfile* object only when the data property is accessed but this includes calls, for example, to *size* or *numel* on the data property. The *nakhur* wrapper prevents this by doing the work for those calls itself. With a *nakhur* object, virtual memory will be allocated only when the data contents are read.

3. Releasing memory

   The *nakhur* class provides support to *reset* an instance, releasing all virtual memory allocated to it. A new memmapfile object will be created and virtual memory will be allocated, as above, only when the data are read . The *collapse* and *expand* methods work similarly. The *collapse* function simply shrinks a memmapfile to a single element while *expand* restores it to its full dimensons. Note that the *instantiateMap* method causes a *memmapfile* to be instantiated but does not allocate virtual memory.

4. Using low-level i/o

   Using memory mapping is typically much faster than using random access file i/o but requires the allocation of virtual memory. The *nakhur* superclass supports low level i/o when required.

The properties and features of the *nakhur* superclass are discussed below in relation to the *nmatrix* class. The *nmatrix* subclass simply adds a constructor and a couple of additional methods. All properties and most methods are inherited from the *nakhur* superclass. All properties are publically accessible but setting of properties is private or protected except for the *TargetType* and *Map* properties (see below).

Note that the *nakhur* class code maintains some lookup data relating to files accessed in any MATLAB session. For low-level i/o, all *nakhur* instances accessing the same file will share the same file handle. Similarly, *nakhur* instances will share a *matlab.io.MatFile* instance for any file. The *where* function is called only once for any file in each MATLAB session - unless the system file modification time stamp alters, in which case the file will be re-analyzed.

The nmatrix class

Constructing nmatrix objects

Construct an *nmatrix* instance from a MAT-file or HDF5-file using:

```
x1=nmatrix(filename, varname)
```

If the file has a non-standard extension, specify the format in the constructor e.g.

```
x1=nmatrix(filename, varname, 'mat');
```

By default, *memmapfile* objects are used for accessing Version 6 files, as well as Version 7 files after copying. Version 7.3 files are supported via the *matfile.io.MatFile* class. To force copying instead on a Version 7.3 file, and hence not be limited by the constraints of *matfile.io.MatFile* class use:

```
x1=nmatrix(filename, varname, 'mat', ...
    'usecopy_always');
```

If you have another format e.g. a custom binary file, you can supply a *memmapfile* object directly:

```
x1=nmatrix(mapfileobject); % TargetType will ...
    default to the type on disc. No byte swapping
x1=nmatrix(mapfileobject, TargetType);% TargetType ...
    as requested. No byte swapping
x1=nmatrix(mapfileobject, , TargetType, ...
    SwapFlag);% TargetType and swapping(true/false) ...
    as requested
```

This form can also be used to create an *nmatrix* instance from a partial map created using the *getPartialMap* function. Finally, an *nmatrix* can have its data assigned directly in RAM.

```
x1=nmatrix(M);
```

where *M* is a standard MATLAB vector or matrix. In this case, the matrix *M* is placed in the *obj.Map.Data.Adc* field. This is a convenience, allowing *nakhur* objects to be used rather than writing conditional code to support both matrices and *nakhur* subclasses. When data are stored in RAM in this way they may be real, complex and/or sparse.

Properties

The viewable properties of the *nmatrix* class are shown below:

```
   nmatrix handle
Properties:
Filename: '/Users/ML/Documents/matlab.mat'
FileFormat: 'MAT-File -v6'
DataSet: '/x'
Mode: 'memmapfile'
TargetType: @uint8
Map: [1x1 struct]
GPUMode: 'off'
GPUTarget: 0
Methods, Events, Superclasses
```

Filename, *FileFormat* and *DataSet* are self-explanatory strings. Note that they may be empty, e.g. when the *nmatrix* is constructed from a *memmapfile* object the name of the data set will be unknown.

Mode identifies the data access type: *memmapfile*, *matfile.io.MatFile*, *fread* or *ram*.

TargetType is described further below. It describes a transform to apply to data accessed from disc. This will often be a simple type cast (hence the name) but *TargetType* may contain a handle to a function performing more complex transforms. *TargetType* is public and user-settable.

Map contains a structure in the example above. This contains the same fields as a *memmapfile* object. In *memmapfile* mode, this structure will be replaced by a *memmapfile* instance when required. While *Map* is a public and user-settable property to allow writing to disc as well as reading, users generally should not replace the contents of this property, only assign values to its sub-properties e.g. obj.Map.Data.Adc(10)=uint8(123);

The *GPUMode* and *GPUTarget* properties support switching between returning data to system and to GPU memory. These are described further below.

Many additional properties are hidden and do not display when an *nmatrix* is displayed. However, the *nmatrix* class has an *inspect* method that displays all these properties if required at the MATLAB command window. For details of the properties see the *nakhur* superclass code. Some of these hidden properties are mentioned below: they are hidden only for clarity as most users will not want to access them.

The TargetType property

Typically, the *TargetType* property will be used simply to cast the represented data to the required type. A *TargetType* property is needed because the data type on disc may be different to the required type. MATLAB's *save* command would automatically cast a matrix of doubles that were all valued 0-255 to the *uint8* class on disc[13] . The *load* command would then cast back. This can significantly increase speed with disc bound code.

TargetType may be a string ('double', 'single' etc) or a function handle (@double, @single). With low-level i/o, the string form will be more efficient as the casting will be done by the MATLAB builtin *fread* function.

As casting is done only on the subset of data returned by calls to *obj(...)*, an *nmatrix* instance can usefully be used to store a data type requiring few bytes when those data will need to be cast on access to a larger type.

As *TargetType* can be a function handle, it can be used to perform more complex transformations on data access. Examples:

```
% Simple
y.TargetType='single';
y.TargetType=@double;
y.TargetType=@(x)double(x)/255;
% More complex
y.TargetType=@detrend;
y.TargetType=@(x)filter(a,b,double(x));
% Copy accessed data from virtual memory to a GPU
y.TargetType=@gpuArray
% Copy accessed data from virtual memory to a distributed pool
y.TargetType=@(x)distributed(x);
```

[13]The data type on disc is available from the hidden Format property of the nmatrix: *obj.Format{1}*. The metadata in a MAT-file provide the required type so, with these files, TargetType will be set automatically by the constructor

As *nmatrix* instances can be passed as inputs to functions, the *TargetType* property effectively provides benevolent code injection to those functions.

Passing nakhur subclasses to m-files

While *memmapfile* objects and *matfile.io.MatFiles* instances can be passed as inputs to MATLAB functions, those functions will need to be specially written to access the data using the APIs for those objects. For a *memmap-file* object for example, code will need to access the *Data* property of the object. This means those objects can not simply be passed as input to most existing functions, including those in MATLAB's own toolboxes.

Objects of the *nakhur* subclass can be passed to functions and will behave as though they were matrices of the data types that they return. This is achieved by overloading standard MATLAB methods to behave in a non-standard way in the *nakhur* class definition:

nakhur subclasses are always scalar so *numel* will always return 1. As this makes MATLAB's '()' syntax for accessing arrays of *nakhur* objects redundant, the behaviour of *subsref* can be altered: for a *nakhur* object, *obj(...) accesses the data contained in the object. For an nmatrix instance with TargetType set to @double* representing data through a *memmap-file* instance (with the data in the *Map.Data.Adc* property), *data=obj(...)* is interpreted as

```
data=double(obj.Map.Data.Adc(...));
```

If TargetType were set to @(x)double(x)/255, *data=obj(...)* would equate to

```
data=double(obj.Map.Data.Adc(...))/255;
```

so might be used to cast *uint8* image data on disc to MATLAB double precision, 0-1 scaled image data.

Other methods then need to be overloaded to provide results that are consistent with this behaviour. For a *nakhur* subclass:

1. size

   Size returns results for the data, not the object. An *nmatrix* instance representing a 128 x 128 x 8192 *uint8* image stack will return [128 128 8192] for *size*.

2. is* and isa* methods

   Is and isa** methods give results for the **returned** data, not data on disc or the object. Thus with the *uint8* matrix above and *TargetType* set to *@double*, *isfloat* will return *true* while *isinteger* will return *false*.

   There is one caveat here. Recall that *TargetType* can be a function handle to any user-written code. The *is** and *isa** methods will generally test the output of this with a test input of zero cast to the type defined in *obj.Format{1}* (the format of the data on disc, see above). As MATLAB variable typing is dynamic, this test could fail if the output of the function in *TargetType* was dependent on the value of the input (e.g. it returned *double* for and inputs of 0-10, and a *uint64* otherwise). In these (unlikely?) circumstances you will need to create a custom subclass and overload the *is** and *isa** methods to suit your needs.

   A couple of new methods are added: *isnakhur* returns true for any *nakhur* subclass. *isnmatrix* returns true for an *nmatrix* or anything that extends the class such as an *adcarray*. In addition *isa(obj, string)* returns true if string is 'nakhur' and as appropriate for the name of a *nakhur* subclass. To achieve this subclasses should set the *Type* property of the object. For the *adcarray* class for example, *Type* is set to *'nakhur:nmatrix:adcarray'* indicating that the *adcarray* class extends *nmatrix* which extends *nakhur*.

3. subsref

   As noted above, standard MATLAB *obj(...)* addressing is redirected to the data property. All MATLAB indexing options are available when using *memmapfile* or low-level access to the data: *subreferencing*, *linear* and *logical* indexing etc.. Missing dimensions and trailing singleton dimensions are all supported. Thus *nakhur* subclasses behave exactly as they would if the data were represented in a primitive matrix.

The *matlab.io.MatFile* class is more limiting. Linear and logical indexing are not supported and all dimensions must be specified. In addition, indices must ascend evenly. Workarounds are included in the *nakhur* class definition, but these are often sledge-hammer style e.g. loading the whole data set, then indexing into that. Improved support for *matlab.io.MatFile* objects will require improvements to the *matlab.io.MatFile* class but that is a job for the MathWorks, not this library.

Two special cases merit mention.

(a) x() For a matrix *x=x()* is equivalent to *x=x*. With a *nakhur* instance, *x=x()* returns all of the data transformed as per the *TargetType* setting. For *y=x()*, *y* will have the dimensions of *size(x)*.

(b) x(:).
Following the logic above, for a *nakhur* object *x*, *x=x(:);* should, and does, return the data as a column vector.
There is one circumstance where this might not be helpful. It is not uncommon for MATLAB toolbox functions to force a column orientation on a vector by calling *x=x(:);*. if *x* is a *nakhur* instance, this will destroy it and replace *x* with a column vector that is potentially many Gb in size. **For consistency, this is the default behaviour**.
The default setting can be altered by calling:

```
x.setTranposeOnRowAccessEnabled(true);
```

This only affects *nakhur* instances that contain vectors. The setting has no affect on those containing matrices or N-D arrays. When the *TranposeOnRowAccessEnabled* flag is set, *x=x(:);* will set another internal flag, called *TranposeOnRowAccess*, in *x* that causes it to return a column vector when it is accessed through linear indexing regardless of the orientation of the vector it represents.

Note that

i. The data dimensions are not altered: size(x) is unchanged.
ii. x=x(:); returns the *nakhur* instance

The *TranposeOnRowAccess* can be set/cleared using *setTranposeOnRowAccess(flag)* where flag is true or false. The *TranposeOnRowAccess* is always cleared when *setTranposeOnRowAccessEnabled(false)* is called or the object is *reset* (but not when it is *collapse*d).[14]

Writing to file

Note that while the *nakhur* class provides read-only support, it is possible to write to the underlying file using the *memmapfile* or *matlab.io.MatFile* objects or the low-level file handle using custom-code. It is left to the user to manage potential share-access violations and to ensure that any data written to disc are in the correct format (including byte swapping if needed). If *TargetType* is used to transform data, the user will need to perform an inverse transform before writing to disc.

The *getIOObject* method returns the underlying i/o object given the current mode: i.e. a *memmapfile* object, *matlab.io.MatFile* object or file handle for low- level i/o. The retured instances will be shared by reference with the *nakhur* instance, and perhaps others as *matlab.io.MatFile* objects and file handles are shared between all *nakhur* instances (and subclasses) that reference the same file . For *memmapfile* objects, the instance will be unique to any *nakhur* instance.

The *isWritable* method returns *true* or *false* depending on whether the underlying wrapped object is write-enabled. For *memmapfile* and *matlab.io.MatFile* access, change the writable property by calling the *setWritable(flag)* method. Note the changing the write flag in

[14]This may seem odd. In the previous version of adcarray, calling x(:) would flip the dimensions for a row vector. This would have local scope only, but as the new version overloads handle, flipping dimensions would leak to all references of the object which is undesirable. The TranposeOnRowAccess flag is also changed by reference, but at least default behaviour can readily be restored by calling setTranposeOnRowAccessEnabled(false). Watch out for threading issues though if you alter these settings off the main MATLAB thread e.g. in callbacks.

matlab.io.MatFile mode affects all *nakhur* instances that share that *matlab.io.MatFile* instance.

For low-level i/o, the internally generated read-only file handle will need to be replaced explicitly with one that is write-enabled. For example, assuming a valid handle is already assigned:

```
obj.setFileHandle(fopen(fopen(obj.getFileHandle),'a+'));
```

The user should *fclose* the file handle when no longer required. The *nakhur* methods will automatically reassign a default file handle to the instance when its data are next accessed.

Data in RAM they are always write enabled. Calling *getIOObject* with mode set to 'ram' issues a warning and returns.

For convenience a *getByteLimits* method provides the limits of the data in file for *memmapfile* and *fread* modes. Users should ensure thay do not write beyond these limits. Limits are zero-based including for data in RAM. Limits will be returned as *[NaN NaN]* is *matfile.io.MatFile* mode.

Distributed and GPU processing

As noted earlier, the *TargetType* property can be used to distribute the returned data or place it onto a GPU by placing the relevant code in a function pointed to by a handle in *TargetType*. This is supported for the MATLAB Parallel Computing Toolbox, the Jacket package from AccerlerEyes[15] and for the freeware GPUmat package from GP-you.org[16].

With the MATLAB Parallel Computing Toolbox for example, *TargetType* to *@gpuArray* will place the returned data onto the GPU. With GPUmat, set *TargetType* to e.g. *@GPUdouble* or with Jacket to*@gdouble*.

Additional support to switch dynamically between using a GPU or system memory is provided for Jacket and GPUmat only. The support works only when the returned data type is a primitive class: *double* or *single* with GPUmat, any primitive type with Jacket.

setGPUMode

Call *setGPUMode(package)* where package can be *'off'*, *'jacket'* or *'gpumat'* to enable GPU support.

[15] `http://www.accelereyes.com/`

[16] `http://gp-you.org/`

setGPUTarget

setGPUTarget(true) and *setGPUTarget(false)* can then be used to toggle between passing data to the GPU or returning data to system memory. Note that

1. If GPU mode is *off*, *setGPUTarget(true)* has no affect and may safely be called in user-code.

2. If GPU mode is *jacket* or *gpumat*, *setGPUTarget(true/false)* will automatically toggle the *TargetType* setting to direct data to the GPU or system memory.

isGPUTarget

isGPUTarget returns *true* if GPU use is turned on, i.e. through a previous call to *setGPUTarget(true)*, and *false* otherwise.

This is likely to be most useful when data are being represented in virtual memory. A frequent limitation of GPU use is the time taken to load data and transfer them on the bus to the GPU. Using DMA to direct data from disc to the GPU will often be an optimal solution but if data are already in virtual memory because the user-application uses an *nmatrix* in *memmapfile* mode, transferring those data directly is likely to be faster still. Tests with a GeForce GTX 465, 1215 MHz, 962 MB VRAM on 64-bit Windows showed a 2.5x speed improvement using *nmatrix* and *memmapfile* to transfer an entire 819200000 byte double precision vector to the GPU even when instantiation of the *memmapfile* was included in the timing. Greater improvements would be available if only part of the data set was needed and/or the data set was already in memory courtesy of the system memory management.

Appendix 1: MAT-file Utility Functions

This Appendix describes the original MAT-file Utilities. These can be used to manipulate large data sets in a Version 6 MAT-file. They work *only* with Version 6 MAT-files.

The MAT-file utilities for writing data to a MAT-file and modifying variables mostly require that you are working with the final variable in the MAT-file i.e. the last variable saved using MATLAB's save function.

Reading data files

where function

where acts similarly to *whos* but in addition provides information about the class of the data on disc and the byte offsets into the file. This can be used to read the file using low level I/O or *memmapfile* For all variables in a file

```
s=where(filename);
```

For a specified variable

```
s=where(filename, varname);
```

For a particular field of a structure

```
s=where(filename, varname, fieldname);
```

or

```
s=where(filename, varbame, fieldname1, ...
    fieldname2 ...);
```

if you have structures within structures
Replace fieldname with propertyname for objects
`[s,swap]=where(...)` sets swap to 0, if the MAT-file endian format is the default for the platform you are using , or to 1 if byte swapping will be needed.

endian function

```
endian(filename);
```

returns 'ieee-le' for a little-endian MAT-file and 'ieee-be' for big-endian

Writing large data sets

The output of *where* can be used to help with reading subsets of data from a variable. The following routines are to assist with writing data:

MATOpen

MATOpen creates a new MAT-file, or if it exists, opens an existing MAT-file in the appropriate endian mode and returns a MATLAB file handle

```
fh=MATOpen('myfile', permission);
```

For valid strings for *permission* see MATLAB's *fopen*.

All the routines below require that the target variable is the last variable in the MAT-file. This can be checked with

```
CheckIsLastEntry(filename, varname);
```

which returns *true* or *false*. If unknown, the name of the last variable in a file can be determined with

```
GetLastEntry(filename);
```

which returns a string.

In addition, all the routines require that data is stored on disc as the same class as the target variable. MATLAB's *save* command casts data to the smallest compatible data type e.g. a *double* array with values all between 0 and 255 will be cast to *uint8*. Run

```
RestoreDiscClass(filename, varname);
```

before calling the *AppendXXXX* functions to restore the class of the data on disc to that of the target variable in memory.

The *AppendXXXX* functions can only be used with real valued variables (not complex data)

AppendVector

Appends data to the end of a row or column vector

```
AppendVector(filename, varname, vector);
```

AppendColumns

Appends columns to an existing row vector or 2D matrix.

```
AppendColumns(filename, varname, matrix);
```

Varname and matrix may be vectors or 2D matrices.

AppendMatrix

Similar to *AppendColumns* but adds data to the final dimension of an N-dimensional matrix where N is >=2. Suppose we have a 100x100x3 RGB image stored in variable img in myfile.mat. We can add a second 100x100x3 image from variable *newimg* using

```
AppendMatrix('myfile', 'img', newimg);

load myfile img
```

will then return a 100x100x6 matrix with the two images. Had *newimg* been 100x100x6, *varname* would have become 100x100x9. To organize the data into a higher dimesional matrix, use *AppendMatrix* in combination with *AddDimension* (see below)

AddDimension

Can be used with *AppendMatrix* to save data to a higher dimension. Using the example above we could use

```
AddDimension(filename','img); % Convert img to a 4D ...
    matrix                                  ...
    %(100x100x3x1)
AppendMatrix(filename, "img, newimg);% Add the 3D ...
    newimg to the                                 %4th ...
    dimension
```

Now

```
load myfile img
```

will return a 100x100x3x2 matrix, the 4th dimension is the image number

In general, for

```
AppendMatrix(filename, varname, matrix);
```

1. If varname and matrix have the same number of dimensions, matrix will be added to the highest dimension of varname (e.g. if varname points to a 100x100x9 matrix and we add a 100x100x6 matrix, we will end up with a 100x100x15 result. The element ordering on disc will be the same as if we had saved a 100x100x15 matrix in the first place. MATLAB's *load* command can be used to access the matrix.

2. If varname has 1 dimension more than matrix, matrix will be treated as a submatrix or (set of submatrices) and added to varname whose final dimension will be incremented by:

$$\frac{\text{size of ultimate dimension of matrix}}{\text{size of penultimate dimension}}$$

of *varname* which must be integer (e.g. suppose varname points to a 100x100x3x22 matrix and we add a 100x100x9 matrix, we will produce a 100x100x3x25 result – matrix is assumed to contain three 100x100x3 matrices e.g. three RGB image frames).

Note that *AddDimension* adds the dimension to the data on disc. Most MATLAB functions including *load*, *whos* etc strip away any trailing singleton dimensions so the effects of *AddDimension* will not show until the final dimension is >=2.

RestoreDiscClass

MATLAB's save command casts data to the smallest compatible data type e.g. a *double* array with values all between 0 and 255 will be cast to *uint8*. Run `RestoreDiscClass(filename, varname);` before calling the *AppendXXXX* functions to restore the class of the data on disc to that of the target variable in memory.

CheckIsLastEntry and GetLastEntry

The *AppendXXX* functions require that the target variable is the last variable in the MAT-file. This can be checked with CheckIsLastEntry(filename, ... varname); which returns *true* or *false*. If unknown, the name of the last variable in a file can be determined with GetLastEntry(filename); which returns a string.

VarRename

VarRename can be used to rename a variable in a file. The original variable name can then be re-used. This is faster than using save –append to replace a variable. *VarRename* is called in two ways:

```
bytes=VarRename(filename, varname);
```

returns the maximum length of the variable name in bytes (the number of bytes reserved on disc for the variable name)

```
res=VarRename(filename, varname, newname);
```

Replaces the variable name on disc with *newname*. This returns 0 if the rename has taken place, -1 otherwise. The length of *newname* must be less than or equal to VarRename(filename, varname);.