Data processing in Matlab using JDATA

Jonathan M. Lilly

August 27, 2004

1 Introduction

JDATA is a set of tools for managing and manipulating large multi-component datasets. Three particular features of JDATA are described here. These are: a set of functions for simultaneously operating on multiple variables (VTOOLS); a variable-remapping scheme (`make` and `use`) which facilitates the use of uniform variable names across datasets; and functions for conveniently handling datasets whose elements are of non-uniform length (`col2mat` and `mat2col`), for example, a set of data records whose duration varies.

2 Multi-component datasets: VTOOLS

The VTOOLS module of JDATA provides a convenient way of manipulating datasets containing many variables. Let's say we have a set of variables

```
s th p sig ox lat lon
```

where `s, th, p, sig` and `ox` are all matrices of the same size, say 1000×20, and `lat` and `lon` are both row arrays of size 1×20. In this example, the rows of the matrices such as `s` correspond to measurements taken at different depths, and the columns correspond to different latitude and longitude locations.

Manipulating these variables in Matlab can be very cumbersome. For instance, to extract a set of columns we write

```
s = s(:,1:4);
th = th(:,1:4); ...
```

and then use

```
s = mean(s,2);
th = mean(th,2); ...
```

to produce a mean across columns. Thus in order to apply the same analysis to the different variables, one tends to repeat all commands many times.

Using VTOOLS, the commands in the previous paragraph may be simply written as

```
vindex(s,th,p,sig,ox,lat,lon,1:4,2);
vmean(s,th,p,sig,ox,2);
```

The first line applies the index [1 2 3 4] to the second dimension of all input variables, the second line takes the mean across the second dimension. Alternatively, we could write

```
[ms,th,mp,msig,mox]=vmean(s,th,p,sig,ox,2);
```

to make a mean set of variables while keeping the original set as well.

All VTOOLS functions perform a simple operation on an arbitrary number of input arguments. Most VTOOLS functions also follow the convention that if no input argument is specified, the input variables are overwritten[1] with the output variables. For example, there are VTOOLS for filtering, taking means, raising to a power, squeezing, and swapping one value for another (such as replacing NaNs with zeros). I find use of this package greatly reduces the amount of typing necessary to process datasets, but does not compromise the clarity of the resulting code.

The VTOOLS functions also have other desirable behavior. The functions `vfilt` and `vdiff` filter and (central) differentiate the data, respectively, without changing the number of rows. Functions for taking means, sums, or statistical moments (`vmean`, `vsum`, `vstd`, and `vmoment`) ignore NaNs, so that NaNs may be used to mark missing data.

3 Variable re-mapping: `make` and `use`

Often one needs to deal with multiple versions of the same physical quantities. That is, we have some variables such as

```
s th p sig ox lat lon
```

which occur in, say, four different regions. To compare these different regions, we could create four different sets of variables

```
s1 th1 p1 sig1 ox1 lat1 lon1
s2 th2 p2 sig2 ox2 lat2 lon2 ...
```

However, this approach tends to lead to constant and exhausting re-naming of variables. It would be much more desirable to have the four versions of the data in memory, and to swap these versions into the originally named variables

```
s th p sig ox lat lon
```

as needed. This idea may be called "variable re-mapping". Then, for instance, commands such as "`plot(s,th)`" or "`s=detrend(s);`" need not be rewritten for each version of the names.

This may be done with the JDATA functions `make` and `use`. Loading the first dataset into memory and writing

[1] This variable-overwriting feature will also work when VTOOLS are called from within an m-file function.

```
make data1 s th p sig ox lat lon
```

creates a structure `data1` with fields

```
data1.s, data1.th, ...
```

and so forth. The other datasets may then be loaded into memory, creating `data2`, etc. Then

```
use data1
```

maps all the fields of `data1` into variables of the same name, over-writing the variables

```
s th p sig ox lat lon
```

with the values of `data1.s`, etc. The original structure `data1` is not modified. Similarly, "`use data2`" over-writes the variable set with the values of the fields of `data2`, and so forth.

In practice I find it convenient to save most datasets as structures originally. In the above example,

```
save data1 data1
```

saves the structure `data1` to a `.mat` file of the same name. If this is done for all the four datasets, then

```
load data1, load data2, load data3, load data4
```

loads all the datasets into memory, and "`use data3`" maps the third dataset into the named variables.

Mapping different values into the same named variables in this way greatly reduces the amount of re-writing one has to do, since one can now run code fragments using only the original names `s, th,` etc. Furthermore, my experience has been that consistently using the same names for the same physical quantities leads to clearer code and reduces programming errors.

4 Datasets of non-uniform length

JDATA also contains some simple but powerful tools for dealing with datasets having non-uniform length. As an example, let's say we have some variables

```
lat lon u v th p
```

measured in say ten different records. For a given record, all of the variables are column arrays of the same length L, but L varies among the ten records. A good example is a collection of records from oceanographic drifters, which follow water motion and report their positions on regular basis, but whose lifetimes tend to vary.

This data may be treated as follows. First, we create long column arrays which contain the individual records, appended one after another:

```
[lat1,lon1,u1,v1,th1,p1]=vempty; %Initialize to empty sets
for i=1:N
    eval(['load data', int2str(i)]) %''load data1'', etc.
    lat1=[lat1;lat]; % Column-append successive records
    lon1=[lon1;lon];
       ⋮
    p1=[p1;p];
    id=[id;i+0*p];
end
lat=lat1;lon=lon1; ... p=p1; % Re-map to usual names
```

where the new variable `id` is a column array of the same size as the other long variables, and indicates the number data set to which each data point belongs.

Next we insert NaNs at the end of every record via

```
colbreaks(id,lat,lon,u,v,th,p).
```

The function `colbreaks` looks for discontinuities in the first input variable `id`, places NaNs in all input variables at the end of each block of contiguous values of `id`, and then overwrites the original input variables. The resulting variables all have NaNs at the same locations, at the end of each of the ten records. This is called "column" format.[2]

From column format, we can move to "matrix" format using

```
col2mat(id,lat,lon,u,v,th,p)
```

where now all variables are now matrices having each record in a separate column. Since we assumed ten separate records, all variables have ten columns. The number of rows is the maximum number of points in any record, plus one, because the final row of all matrices will contain only NaNs. In matrix format, the first element of each record is placed in the first row, and if a record has a shorter length than the maximum, remaining points at the bottom of the corresponding column are filled in with NaNs. Column format can be recovered from matrix format

```
mat2col(id,lat,lon,u,v,th,p)
```

so that we can move between these two formats as desired.

Data of non-uniform length data are quite common, and are often produced[3] or initially loaded as long columns, to which an "`id`"-type variable may easily be attached. However, many operations are more natural to perform in matrix format. For example, in matrix format one can write "`mu=vmean(u,1)`" to find the mean of `u` within each column. On the other hand, column format datasets make take up substantially less space when saved as `.mat` files. The functions `col2mat` and `mat2col`, together with `colbreaks`, let one rapidly alternate between the two formats.

[2] It is assumed that there are no other NaNs present in the data; bad or missing data points should be marked with some other value.

[3] This type of data occurs particularly in wavelet analysis ridge or modulus maxima extraction.