Working with DTAG-4 data sets

Mark Johnson
10 July 2018
markjohnson@st-andrews.ac.uk

- If you want to configure or offload a D4 tag, see: d4_install.pdf and d4_configuration.pdf.
- If you need to unpack DTG files, go to Section 1.
- If you want to work with movement (pressure, acceleration etc.) data, start at Section 2.
- If you want to work with sound data, go to Section 5.
- If you want to work with GPS data go to Section 7.
- If you just want a quick summary of how to get data into Matlab/Octave go to Section 8.
- If you want to convert existing DTAG-2 PRH data into the .nc format, go to Section 9.
- If you want to update the attributes (calibration information) in a tag, go to Section 10.
- If you need some ideas for troubleshooting the data reading and analysis steps, go to Section 11.
- If you need help on converting tag frame data to animal frame using *prhpredictor,* see the document by Lucia Martin Lopez.

Before you start

If you are working with a new dataset, pick an appropriate deployment id for your dataset. Typically you should follow the standard DTAG protocol of: 2-letter Latin or common species initials, 2-digit year, underscore, 3-digit Julian day of year, 1-letter animal of the day, e.g.:

```
depid = 'mn17_283b' ;
```

To find out the Julian day for a given date, use:

```
julian_day([2018,7,5])          % find out the julian day number of 5th July 2018
```

It is a good idea to keep to this naming convention because it will avoid that you end up naming your data the same thing as someone else has. It will also make it easier to search for data in archives. Unfortunately, we have not been very strict in the past about whether the species initials are taken from the Latin binomial (which is preferred) or the common name. So you will find DTAG data from sperm whales that is labelled 'pm' and 'sw'. Unless there is a compelling reason not to, please use the Latin binomial.

Download the latest Matlab/Octave tools for the tags from the D4 Google Drive.

You need both animaltags.zip and d3matlab.zip. The DTAG-4 uses the same tools as the DTAG-3 except in a few cases. All of the tools for both versions of the tag are in d3matlab.zip. These tools should work equally well in Octave (if you discover any that do not, please email me with details).

Unzip these files and add the resulting directories (with subfolders) to the Matlab/Octave path. You can do this with a pull-down menu option (the location of which varies annoyingly by Matlab version), or you can type *pathtool* (followed by the return key) in the command window to access the same dialog box. Remember to save the path (with the button at the bottom of the dialog box) once you have added the tag tools directories.

One more point: Looking at the size of this document, you may be asking yourself: Why are there so many processing steps with DTAG data compared to commercial tags?

The DTAG is a research tool which is in continuous development and is constantly being adapted to work with different species, measurements and conditions. In contrast, a commercial tag necessarily represents a frozen technology in order to be produced in quantity at a price that generates profits. In effect, with DTAGs you are accessing the technology at an earlier stage of development than you do when you access a commercial tag. This is reflected in the less-polished interface and the extensive data processing and checking steps required with DTAGs. However, commercial tags have their own problems when it comes to reading and checking the data: The file sizes and formats can be clumsy and you may need to work out for yourself how to get the data into a processing programme such as Matlab/Octave. Moreover, many of the data checking, calibration improvement, and metadata preparation steps in this document are just as applicable to commercial tags as they are to DTAGs. So, yes, DTAG's are a bit tricky to use and you need to be a participant in the technology, not just a user. But the benefits are that you get to use something new and different, and that you trust the results because you have invested time in learning how to check them.

1. Unpacking DTG files

Data from all versions of the DTAG are stored in .dtg files. This is a compact file format that was developed specifically for DTAG data and contains all of the data collected by the tag along with information about the configuration of the tag, as well as calibration data, and timing information. The .dtg format is a complete stand-alone archive of the tag data that is suitable for long-term storage but it depends upon having the reading software. For D4 data, the reading software is called *d4read.exe*. Make sure that you store a copy of this programme with D4 data for long-term storage. The most recent version of *d4read.exe* can be found on the D4 Google Drive.

The *d4host.zip* file at this location contains the latest *d4read* and *d4host* programmes along with documents on how to install and configure D4 devices.

To unpack data, just double click on *d4read.exe* and enter the requested information. To change the data directory, type *y* and then enter the full directory path. Unfortunately you cannot copy and paste into this window so it will help if your directory names are fairly compact! Then enter the first few letters of the name of the files in the data directory. Usually the directory will only have data from one deployment and in which case, just enter the first letter of the files. You will get a list of files to choose from. Type *a* to convert all of them or enter the numbers on the left hand side of the directory list of the files you want to convert. The programme will work through all of the files you specified, unpacking each one. It will report any errors that it encounters and give a summary of the number of undecodeable 'chunks' at the end. It is normal to have a few (even up to 50-100) bad chunks in each file expecially if it is a large. These come about from errors in the flash memory used in the tags and normally just lead to a brief glitch or outage in the sound data. As each 'chunk' of data spans only about 2000 audio samples, these glitches are very short. If you get a large number of errors in a particular file, it may be advisable to try offloading it again from the tag to see if that reduces the error count.

After unpacking, each .dtg file will have generated a number of output files. These vary according to the tag type and configuration. The most common file types are:

.bin	Snap-shot GPS data in a D4 binary file format.
.swv	Movement and other sensor data in a WAV format file.
.wav	Audio data in a WAV format file.
.snr	Sonar data in a WAV format file.
.ecg	ECG data in a WAV format file.
.tlog	Text (CSV) file containing timing data for two clock sources in the D4. Not used so far.
.vlog	Text (CSV) file containing the battery voltage as a function of time.

.wavt	Text (CSV) file containing timing information for all WAV format data sources.
.xml	XML file containing metadata defining the recording conditions and sensor parameters.

The number of these files is one reason why the .dtg file is a good choice for archiving - it is much safer to store one file than 9. The .dtg file is also a lot smaller because of compression that is used with the audio (and some other WAV format) data. The WAV format files are uncompressed whereas the .dtg files use loss-less compression (similar to using zip compression on files).

Even if a tag records continuously, the size of each individual file collected by the tag is limited to 700MB or 2^{31} audio samples, whichever is smaller, to facilitate post-processing and archiving as well as to conform with some WAV file readers. As a result, a single deployment may generate a number of files. If the recording is not interrupted by a stop condition (see *d4_configuration.exe* for an explanation of these), these files will be contiguous, i.e., there should be no data outage between adjacent files. If the tag stopped during a deployment (e.g., because you programmed a duty cycle, or you set a dry threshold or a stillness threshold), the files will not be contiguous. In either case, the data can be handled just fine by *d4read* and the Matlab/Octave tools that follow.

If you need to change the names of all the files in a recording without altering the suffixes or the three digit file numbers, use *change_fnames*. For example, if the data directory is *'e:/mn18/mn18_175d'* and the files are currently called *mn175d001* etc., and you want to change these to *mn18_175d001* etc., you would do the following in Matlab/Octave:

change_fnames('e:/mn18/mn18_175d','mn175d','mn18_175d')

Very rarely, a chunk error can occur in a data chunk that contains configuration information. If that happens, *d4read* will not know how to decode subsequent data from the sensors that were defined in that chunk and will display messages like 'data for uninitialised chunk'. This can be overcome by running a repair on the .dtg file but the tools for doing this are not yet easy to use. So if you find a problem like this email me and we can try to solve it.

A note on time: The D4 tags keep time using UTC (i.e., GMT) so all references to time in the files produced by the tags (e.g., the .xml files) are in UTC. The time clock in the tag is configured by the host computer everytime you talk to the tag so it is important that the computer is setup to the correct time otherwise the tag will get it wrong. The important thing to remember is that the computer should be set to LOCAL time and should have the correct time zone setting. That way, the tag can figure out what the UTC time is.

2. Reading DTAG-4 sensor data

Before you start, decide on a deployment id (see the 'Before you start' section above) and put this in a variable called *depid*. You also need to specify the directory where the raw data is, e.g.:

recdir = 'E:\hs17\hs17_283b' ;

This needs to be the full path including drive letter. In Windows, if you have a window open showing the data files, you should be able to copy the directory path from the navigation panel at the top of the window.

The raw sensor data from a DTAG deployment resides in a number of .swv files. To get the entire sensor dataset, each file needs to be read in and checked. There are tools for doing this automatically but, unless your deployment was very short, the size of the resulting sensor dataset can be so big that it will either not fit in memory or will at least slow down your computer drastically. To get around this, the swv reading tool allows you to decimate (i.e., reduce the sampling rate) of the sensor data and/or select a subset of the sensor channels to read.

To choose these options you first need to find out what sensors were recorded in a deployment and what the original sampling rates were. Do this using:

d3readswv(recdir,depid);   % just print a list of sensors and sampling rates

In most cases, there will be more than one sampling rate across the sensors, for example, the accelerometers will have been sampled at a higher rate than the pressure sensor or the magnetometers. There is a good reason for this: acceleration signals contain high frequency information which may be useful for inferring behaviour whereas pressure and magnetometers tend not to, at least to the same extent. This adds an extra complexity in choosing a decimation factor - do you want to apply the same decimation factor to all sensor channels or use different factors on different channels? Both are possible.

Here are some examples for how to read in the sensor data that may be useful. Of course, none of these actions change the data in the original *.swv* files, so you can read the same data in multiple ways. If you just want to get started with DTAG data without learning all of the different methods for reading it in, just do the first step below and then go to the paragraph following option 7.

1. Read in all the sensor channels and decimate them to a common 5 Hz sampling rate. This is useful to get an overview of a deployment, i.e., a dive profile and a general idea of the orientation of the animal and its direction of movement. The full 5 Hz dataset should fit in memory even for a very long (2 month) deployment.

X = d3readswv_commonfs(recdir,depid,5) ;

2. For smaller animals, a 5 Hz sampling rate may be too slow to capture rapidly changing behaviours. Decimating to a common 25 Hz sampling rate may be better in this case but the resulting dataset may only fit in memory for shorter deployments (upto 1-2 weeks).

X = d3readswv_commonfs(recdir,depid,25) ;

3. If you want to retain some high frequency information in the accelerometer signals but reduce the total size of the dataset so that it fits in memory, you can choose a decimation factor to apply to all channels. For example, to read all the sensor channels and decimate each one by a factor of 10, do this:

X = d3readswv(recdir,depid,10) ;

If the original accelerometer and pressure sampling rates were 200 Hz and 50 Hz, they will now be decimated to sampling rates of 20 Hz and 5 Hz, respectively.

4. You can select a subset of channels to read in and decimate these. For example, to just read the pressure sensor channel(s) and decimate these by a factor of 10, do this:

X = d3readswv(recdir,depid,10,'pres') ;

This is useful for quickly reading in the dive or altitude profile, e.g., to find out when the tag was actually attached to an animal.

5. You might also want to read in just the accelerometer channels and process these in a different way (and at a different sampling rate) than the other sensors. For example, to read the accelerometer data without decimation do this:

X = d3readswv(recdir,depid,[],'acc') ;

6. If you really want to read in the entire sensor data at full sampling rate (this will only be feasible for short deployments), do the following:

X = d3readswv(recdir,depid) ;

7. If you want to read in a section of the sensor data at full sampling rate from times t1 to t2 (in seconds with respect to the start time of the deployment), do the following:

X = d3getswv([t1,t2],recdir,depid) ;

All of these methods for reading in sensor data will result in a structure, X, that contains three fields: *X.x*, *X.fs*, and *X.cn*.

X.x is the sensor data. Because each sensor can have a different sampling rate, the length of the data for each sensor over the deployment can be different. Accordingly the sensor data is stored in a cell array with one sensor channel per cell to allow for different sized vectors. You access members of a cell array in Matlab/Octave using the curly parenthesis *{}* rather than the usual round brackets *()*. For example, to access the 8th sensor channel and make it into a variable *p*, you do:

p = X.x{8};

To access the first three sensor channels (assuming they are all the same size and so can be put together into a matrix, *A*), you do:

A = [X.x{1:3}];

If you used the *'info'* option with *d3readswv*, *X.x* will be empty because this kind of read is just getting the sensor information not the data.

X.fs is sampling rate of each sensor channel in Hz (i.e., samples per second). *X.fs* is a vector with a number each sensor channel. To access the 8th element of *X.fs*, you would do *X.fs(8)*.

X.cn is the identification number of each sensor channel. These are the id numbers that DTAGs use to figure out what kind of data is in each sensor channel. *X.cn* is a vector with an id number for each sensor channel and is usually used by other programmes rather than directly by yourself. For example, use *d3channames(X)* to find out which channels are present in *X* and in what order. From this you may find out that the pressure sensor data is, for example, in channel 8. In this case you have all the information you need to plot the raw dive profile:

plott(X.x{8},X.fs(8))

If you do this, you may find that the plot doesn't make sense compared to what you expect the animal to be doing. Depending on the tag, the depth numbers may be way too small. This is because most of the data that comes off the tag is not calibrated. Calibration values need to be applied to each channel of X.x to convert the data into familiar units and this is the topic of Section 4. But first you need to organise the metadata for the deployment using Section 3.

Off-animal data

Tags can detach from animals during deployment but continue recording. This is usually obvious in the dive profile - it becomes flat apart from random brief movements due to waves passing. If the duration of the off-animal data is short, it won't do any harm. However, if there is a lot of it, it will take up unnecessary space in the processed sensor data files and it may also cause problems for the automatic calibration algorithms. You can remove off-animal time in each sensor using the animaltags tools *crop* and *crop_to*. But you can also simply remove excess *.dtg* files from the deployment before you read in the data. You shouldn't delete these unless you are sure they contain nothing of interest - even ambient sound levels recorded by a tag floating at the surface could be useful. Better is to put them into a sub-directory called *off_animal* in the deployment directory. Put both the *.dtg* files and all of the associated unpacked files in this sub-directory. To figure out if a particular *.dtg* file was recorded off the animal, do the following:

```
X = d3readswv(recdir,prefix,10,'pres',N)       % replace N by the number of file you want to test
plott(X.x{2})                         % the pressure is in the second cell, temperature is in the first cell
```

If no dives are evident in this and subsequent *.dtg* files, they are most likely off animal and should be moved to the off_animal sub-directory. When you have done this, run:

```
clear_cues(depid)
```

This forces the software to re-assess how many *.dtg* files there are the next time you run *d3readswv* or any of the other reader functions.

3. Preparing metadata

It is really important to document data. This will be useful for you to help keep track of where data came from and how it was processed. It will also save you a lot of time when you share data with other people, or add data to public archives. There are growing efforts to define standards for the metadata that should go along with tag data. The format adopted by the animaltags project has two components:
(i) The use of the NetCDF file format for data. This is an international standard file format for scientific data which makes it easy to add both general metadata (i.e., that applies to all the data from a deployment) and data-specific information (i.e., that describes a particular kind or section of data).
(ii) The definition of standard metadata structures for general and data-specific information. These structures are extensible - you can add whatever additional information you want to them - but the basic structures provide a guide as to what should be the minimum content.
There are some immediate advantages to using this format:
- The Matlab/Octave/R animaltags tools, and increasingly the DTAG tools, directly support this format and make it easy to collect metadata as you perform calibration and processing steps on the data.
- NetCDF files can be exchanged without problems between Matlab, Octave, and R on Windows and Mac.
- Mail servers seem to be happy to deliver NetCDF files but will refuse Matlab .mat files.
- Journals that require datasets accompanying papers should accept this format.

Once you get used to using the NetCDF format, it will save you a lot of time and make your data products more professional. But it might seem a bit more time consuming to begin with. To make it more streamlined, take some time to define the researchers in your group and the species that your group works with. This information is collected in two text files: *researchers.csv* and *species.csv*. You will find templates for these in the *user* directory in the animaltag tools. These are files that you need to edit and maintain yourself. You should move them to another directory on your Matlab/Octave path so that they don't get overwritten when you download a new copy of the animaltag tools. You can edit these files in any text editor and it is obvious from the first line in each file as to what data needs to be entered. These files are simply used as a quick way for the Matlab/Octave tools to preset some of the fields in the metadata structure that will be made for each deployment. The first column of each file contains the shortcuts that the tools will use to access specific lines from the files, for example, your initials in the first column of *researchers.csv* will be used to find your affiliation, contact information and data use preferences.

General metadata

The general metadata for a deployment is called the *info* structure. The following line makes an info structure for a deployment of a D4 tag by a person with initials 'xx'. Replace 'xx' with your initials or the initials of whoever is the 'data owner'. Information on that person must be listed in *researchers.csv*. The first two letters of depid are used to lookup the species in *species.csv*. If more than one species matches these two letters, you will be asked to choose which one you mean.

```
info = make_info(depid,'D4',depid(1:2),'xx')
```

The fields in *info* will be preset to some relevant metadata. You can edit these fields to give extra information, e.g.,

```
info.dephist_deploy_method = 'glue' ;     % or 'suction cups' if on a whale
info.dephist_deploy_locality = 'Husum, Germany' ;
info.project_name = 'UWE' ;
```

It may be easiest to make a script for each field trip that will generate an info structure and add all the extra fields that will remain the same throughout the trip, e.g.:

```
info = make_info(depid,'D4',depid(1:2),'sp');
info.project_name = 'ONR Tag Design';
info.project_datetime_start = '2018/06/04';
info.project_datetime_end = '2018/06/10';
info.dephist_deploy_locality = 'Cape Cod, Massachusetts';
info.dephist_deploy_method = 'suction cup';
info.dephist_deploy_location_lat = 41.7;           % the approx latitude of the tag deployment
info.dephist_deploy_location_lon = -69.9;          % the approx longitude of the tag deployment
```

To make a script, think of a suitable name, e.g., *capecod18_info.m* and then do:

```
edit capecod18_info.m
```

Select 'yes' if a window asks you if you want to create a new file. Copy in the above lines to the edit window that opens up. Edit the lines to reflect your information, then save the file. Now can then generate a personalised *info* structure by just typing *capecod18_info* at the Matlab/Octave command line.

Data-specific metadata

As well as the general metadata for a deployment, some sensor-specific metadata is needed with each type of sensor data as well, for example defining its units, axes and sampling rate. Sensor data and metadata are combined together in a sensor structure that, like *info*, has a basic set of fields but to which you can add more fields. *sens_struct* is a tool for making a sensor structure that is appropriate for each data type. For many D4 devices, the temperature data is in channel 7 of the sensor cell array that is returned by *d3readswv* or *d3readswv_commonfs*. So to make a temperature sensor structure from the data in this sensor channel, you do:

T = sens_struct(X.x{7},X.fs(7),depid,'temp');

For depth data, assuming that the pressure sensor is in channel 8 of X.x, you do:

P = sens_struct(X.x{8},X.fs(8),depid,'press') ;

For acceleration data and magnetometer data which are usually in channels 1-3 and 4-6, respectively, you do:

A = sens_struct([X.x{1:3}],X.fs(1),depid,'acc') ;
M = sens_struct([X.x{4:6}],X.fs(4),depid,'mag') ;

Note that the order of the sensor channels may change if your tag was recording a different set of sensors (e.g., ECG, light). Use *d3channames* as described above to check.

To find out what sensor types are supported by *sens_struct*, just type *sens_struct* and then the return key in the command window. To tell *sens_struct* that your data corresponds to a particular sensor type, you give the first few letters of that sensor type in the last argument to *sens_struct*, e.g., *'acc'* for acceleration.

You can find out what is in any of these structures by typing its name and the return key in the command window. Do this for P and you will see that it claims that the units are in mH2O (i.e., metres of depth) which is a complete lie! The sensor structure fields are expecting that you will now calibrate the data, i.e., convert it from the raw tag units to the units that are listed in the structures. The advantage of making the sensor structures before applying the calibrations is that now the calibration constants will get added to the sensor structures so that all the processing steps you apply to each sensor will be recorded.

4. Getting, applying and improving calibration information

For D4 tags, the calibration information is stored on the tag itself in a set of information called the 'attributes' of the tag. The attributes include the capabilities of the tag (e.g., whether it includes a GPS) and the calibration information for each sensor and each setting, e.g., for the accelerometer in each of the full-scale modes: 8g, 4g, and 2g. When you offload data from the tag, the attributes are automatically added to each of the xml files for the deployment. If the attributes have been setup correctly in the tag, they should be a good starting point for calibrating the tag data.

Get the calibration constants for this tag deployment:

CAL = d4findcal(recdir,depid) ;

This will read the attributes of the tag and automatically pick the calibration constants associated with the configuration settings that were used for the deployment. The *CAL* structure contains a field for each of the main sensors in the tag (ECG, light and sonar are not yet included and we have different steps for these sensors).

The *CAL* information has to be applied to each sensor channel separately and this is followed by some checks and adjustments that are specific to each sensor type. Follow the order below:

1. Temperature
The temperature measurement is internal to the tag and is not very accurate so it cannot be used as an ambient temperature measurement without careful calibration (and even then it will lag behind the true ambient temperature by many seconds). It is however useful for compensating temperature effects in the other sensors. This is why it has to be checked first. The temperature sensor calibration may be several degrees off from the true temperature although the scale factor will be about right. For most things this doesn't matter but if you want more precise temperature you can measure the temperature offset of the sensor in the tag (e.g., by letting the tag record at a known temperature) and add it to the calibration using:

CAL.TEMP.poly(2) = CAL.TEMP.poly(2) + temp_offset ;   % add your own temperature offset

Either way, you apply the calibration to the temperature sensor by doing:

Tc = do_cal(T,CAL.TEMP);

The data in Tc will now be roughly in the correct units (degrees C).

2. Pressure
Pressure sensors tend to drift with temperature so the pressure calibration needs the temperature data in order to correct this drift.

Pc = do_cal(P,CAL.PRESS,'T',Tc) ;   % apply cal to the pressure using the calibrated temperature

Plot the resulting pressure data to check it:

plott(Pc)

For aquatic animals, the pressure should be close to 0 when the animal surfaces (surfacings are usually obvious in the pressure plot as times when the pressure reaches the fairly flat line that joins the highest points). The zero-pressure offset of the sensor is often a little off due to both errors in the pressure sensor and temperature sensitivity. There are two tools that you can try to fix the pressure data. *fix_pressure* tries to correct temperature sensitivity and offset assuming that these are constant throughout the deployment. It is called like this:

[Pf,pc] = fix_pressure(Pc,Tc);   % Pf is the hopefully 'fixed' pressure sensor structure

The tool will tell you if it couldn't make a temperature correction, in which case try the next option below. If the tool seemed to work, do *plott(Pf)* to check that the top of the dive profile is flatter and closer to 0. If you are content with the result, you need to add the corrections that *fix_pressure* made to the calibration structure in case you want to do the calibration again (e.g., at a different sampling rate):

CAL.PRESS.poly(2)=pc.poly(2);   % update the CAL for the pressure sensor offset
CAL.PRESS.tcomp=pc.tcomp;   % and the temperature compensation

If *Pf* still doesn't look correct when the animal surfaces and the '0' pressure seems to be changing over time, try the other tool:

Pf = fix_offset_pressure(Pc,300,400);

The two numbers 300, and 400 control how *fix_offset_pressure* searches for potential surfacing intervals and these need to be adjusted to match the behaviour of your animal. The first number is the search interval in seconds that is used to find surfacings. This should be chosen to be a little more than the usual dive duration. The second number controls how quickly the zero-depth offset is allowed to change. This should normally be several times larger than the first number (i.e., preventing the offset from changing quickly) unless the plot of Pc suggests that the depth sensor is making steps in its zero offset (this could happen if the sensor gets hit or gets some sand wedged on to the diaphragm). Plot Pf and adjust the two numbers in *fix_offset_pressure* up or down as required to make the surfacings look reasonable.

3. Saving the results

Having already invested some effort into calibrating the sensor data, it is a good idea to start saving the results in case Matlab/Octave crashes. There are two things to save: the CAL structure and the calibrated sensor structures.

Save the CAL structure to a calibration file for this deployment as follows:

save([depid,'cal.mat'],'CAL')

This will make a file called the same as your deployment id but with a *'cal.mat'* ending in your current working directory. You can retrieve this file later using: *CAL=d4findcal(depid);*

To save the sensor data, first make a name for the *.nc* file that you are going to create. This should start with the deployment id and then indicate what is in the file and the sampling rate. There are no rules for this but some ideas are here:

trk = position data from GPS
sens5 = sensor data at 5 Hz rate
p = depth data only
pAM25 = pressure, acceleration, magnetometer at 25 Hz
aud = clips of audio

Following the name with the sampling rate in Hz is a good idea and it allows you to have multiple versions of the data with different sampling rates. If you decide to have an ending *'sens5'*, you can make the full *.nc* file name like this:

ncname = [depid,'sens5']

Use this to make a *.nc* file with the *info* structure that you made in Section 3, and the sensor data that you have calibrated so far. You can add more data to this file later using *add_nc*. It is important to put *info* into every *.nc* file you make for this deployment because it ensures that each file is self-contained with all of the deployment metadata.

save_nc(ncname,info,Tc,Pf) ;

Note: Use Pc in the line above if the initial calibration was fine and you didn't need to run the correction tools.

4. Acceleration

The accelerometers in D4 tags tend to vary a little in sensitivity and offset but don't seem to be very temperature sensitive. Fortunately, the constant gravitional field strength makes it straightforward to infer the sensitivity and offset of each sensor axis of an accelerometer. *auto_cal_acc* takes the existing calibration and automatically tries to improve it based on the data. This function first gives you the option of selecting a sub-section of the full data: If the deployment includes a lot of time when the animal is at the surface or hauled out, or when the tag is off the animal, this doesn't provide useful information for the automatic calibration procedure. Use the mouse and the 's' and 'e' keys on the plot to select a data section that represents relatively active behaviour (this should encompass a lot of the deployment - if you just select one dive, the calibration may only be good for that dive). When you have selected the data on which to focus, press the 'q' key and *auto_cal_acc* will try to improve the calibration reporting the resulting improvement in terms of how far the accelerometer data is from the gravitational field. A small percentage is better and values around 2% or lower are very acceptable.

[AA,ac] = auto_cal_acc(A,CAL.ACC) ;

Use *plot(check_AM(AA))* to make sure that the calibration looks good. Well-calibrated accelerometer data should give a fluffy but flat line in this plot. The fluffiness comes from specific acceleration, i.e., moments when the animal is accelerating or decelerating. If you zoom into times when there isn't much specific acceleration, the plot should be pretty close to a flat line at 9.8 m/s^2. If this looks good, save the improved calibration:

CAL.ACC = ac ;

The calibrated acceleration, *AA*, is not yet correct because it doesn't account for how the sensor is oriented in the tag. *auto_cal_acc* works in the sensor frame (i.e., with respect to how the manufacturer of the sensors defines its x, y and z axes), not the tag frame, because it is trying to figure out how to correct each axis of the sensor. To apply the calibration and convert the sensor axes to the tag axes do this:

Ac = do_cal(A,CAL.ACC) ;   % Ac is now in the tag frame

Finally, add Ac to the .nc file and update the calibration file with the new calibrations:

add_nc(ncname,Ac) ;
save([depid,'cal.mat'],'CAL')

5. Magnetometer
The magnetometers used in D4 tags are sensitive to temperature and can drift over time. Magnetic field measurements can also be affected by metallic parts in the tag either warping the flux lines (a so-called 'soft iron' effect) or having a small magnetic field of their own (called a 'hard-iron' effect) due, for example, to being near a permanent magnet at some time. This combination of factors makes magnetometers more prone to error than accelerometers and makes data-driven calibration essential: The calibration constants you get when you calibrate an accelerometer by doing the flips might be good for several deployments, but they will likely only serve as a starting point for calibrating the magnetometer.

Fortunately, as with the accelerometer, the locally-constant magnetic field strength makes it straightforward to infer the sensitivity, offset and temperature sensitivity of each sensor axis of a magnetometer. This is no longer so simple if the magnetic field strength changes over the course of the deployment because the animal travels a long distance (see below for more on how to deal with magnetometer data in long deployments). For short (e.g., < 2 day) deployments, you need to first determine the magnetic field strength for the area in which the animal was tagged. The NOAA NCEI Geomagnetic Calculators are a good resource for this. The field strength needs to be in micro-Teslas (μT) and should be a value between 20 and 80. You don't need a very accurate value for magnetometer calibration so the magnetic field map included in the docs directory of the *d3matlab* tools may be good enough.

auto_cal_mag takes the existing calibration and automatically tries to improve it based on the magnetometer data, the local field strength, and the tag temperature (the *Tc* that you calibrated earlier). As with *auto_cal_acc*, you have the option of selecting a sub-section of the full data to exclude time when the animal is at the surface or hauled out, or when the tag is off the animal. Use the mouse and the 's' and 'e' keys on the plot to select an extensive data section that represents relatively active behaviour. When you have selected, press the 'q' key. *auto_cal_mag* will then try to improve the calibration and will report the resulting improvement in terms of how far the magnetometer data deviates from the local field strength. A small percentage is better and values around 2% or lower are good.

[MM,mc] = auto_cal_mag(M,CAL.MAG,field,Tc) ;

Use *plot(check_AM(MM))* to make sure that the calibration looks good. Well-calibrated magnetometer data should give a flat line in this plot with a little fluffiness. The fluffiness comes from rapid changes in orientation. If you zoom in, the plot should be a fairly flat line at the field strength in μT with occasional small deviations. If this looks good, save the improved calibration:

CAL.MAG = mc ;

As with acceleration, the calibrated magnetometer data, *MM*, is not yet correct because it doesn't account for how the sensor is oriented in the tag: *auto_cal_mag* works in the sensor frame not the tag frame. To apply the calibration and convert the sensor axes to the tag axes do this:

Mc = do_cal(M,CAL.MAG) ;   % Ac is now in the tag frame

Finally, add Mc to the .nc file and save the updated calibration file:

add_nc(ncname,Mc) ;
save([depid,'cal.mat'],'CAL')

More on calibrating the magnetometer in long-duration deployments
TBD

Final note on calibration:
The quality of the calibration information in the tag depends on you. Before using a tag in the field, you should do the usual flips to check the accelerometers and magnetometers. You should also lower the tag into water for a few metres using a measured string to check the pressure sensor. These tests should all be done with the configuration settings (e.g., depth and accelerometer ranges) that you are planning to use in the field. Check the data from these tests using the steps here to make sure that you get reasonable answers. If you don't either edit the attributes (see Section 9) or contact me.

5. Working with sound data

Run the audit tool:

R = *d3audit(recdir,depid,depid,start);*

Read in a section of data:

[x,fs] = d3wavread([t1,t2],recdir,depid) ;

6. Summarizing high rate acceleration and sound data

Generate an RMS jerk vector with a sampling rate of e.g., 5 Hz. This takes some time to run because it reads the entire high-rate accelerometer data.

J = d3rmsjerk(recdir,depid,CAL.ACC.poly,5);
add_nc(ncname,J) ;

Computing long-term spectral averages of audio data:

[SL,f,t] = d3wavSL(recdir,depid,[],1024,30) ;

Flow noise:
TBD

7. GPS grab processing

Some D4 tags include a Snapshot GPS sensor. This type of GPS gathers information from the GPS satellites when the animal is at the surface but does not try to decode the position. This makes it both very fast and low power. A normal GPS can take 5-200 s to get a position depending on how long it is since the last one. This is way too long for many marine animals that only come to the surface briefly. In comparison, the D4 GPS takes a snapshot in less than 0.1 s no matter how long since the last one. But the D4 GPS stores a data packet of 64 kBytes for every snapshot whereas a normal GPS could store a position in 16 Bytes. The snapshot GPS therefore uses 2-3 orders of magnitude less power but 3-4 orders of magnitude more memory because all of the analysis steps that reduce the size of the data are done after you recover the tag.

Processing the snapshot data after a deployment to resolve position involves three steps: first the snapshot data packets must be searched to find which GPS satellites were present and to determine the arrival times of their coded signals. This results in what are called 'pseudo-ranges' from each satellite. By themselves, these do not tell you the position but they do give you a good indication of whether a position will be decodeable. At least 4 satellites must be received with good signal-to-noise ratio to decode a position but 5 or more satellites will give a more reliable position. The second step requires estimating a starting position and clock offset for the tag. This may be easy if you know where the animal was tagged and you configured the tag on a computer that had the correct time and time-zone. But if the tag had a delayed start or your computer did not have the correct time, then you may need to work harder. The third processing step involves calculating the animal's position from the pseudoranges and the positions of the satellites (these are called the ephemeri). In a normal GPS, the ephemeri are decoded from the messages sent by each satellite and this is what makes a GPS take so long to acquire a position. The ephemeri are also published on the web (with a day or more of delay) so the snapshot GPS software tools download them directly from the internet. As a consequence, if you are in the field with no internet connection, you will only be able to do the first step in the processing.

1. Pseudo-ranges:
The data captured by the snapshot GPS is stored in *.bin* files after unpacking the *.dtg* files using *d4read* (see Section 1). There will be a *.bin* file associated with every *.dtg* file if a GPS capture occured during the interval spanned by that *.dtg* file. This depends on the animal's behaviour, the configuration settings of the tag, and the location of the tag on the animal. If the tag does not come out of the water, no GPS grabs will be taken and no *.bin* file will be made.

Processing the *.bin* files to calculate pseudo-ranges is fully automatic but is time consuming. It can take a day or more to process an entire deployment depending on the number of captures and the speed of your computer. The instruction to do this is:

d3preprocgps(recdir,depid) ;

d3preprocgps analyses each *.bin* file in turn and generates a much smaller file (called the same as the *.bin* file but with an ending '*gps.mat*') containing the decoded pseudo-ranges. This tool also reports and graphs information on the satellites found in the data and their signal-to-noise ratios. You can just let the tool run but if you want to get an idea of whether there are good captures in the data (i.e., with 4 or more satellites that may yield a position), have a look at the results on the screen for each capture. The first column is the satellite number. There are 32 GPS satellites, called Space Vehicles (SV) in NASA-speak, and these are numbered 1..32. The second column is the received power of each satellite. The received power has to be greater than about 200 for that SV to be usable. So you are looking for captures that have preferably at least 5 SVs with power > 200.

When *d3preprocgps* is finished, gather the results from all of the '*gps.mat'* files into a structure which will be used in the next steps:

OBS = d3combineobs(recdir,depid) ;

2. Work out starting estimates:
The GPS position decoding algorithm requires fairly good estimates of the start position of the tag and its clock offset with respect to GPS time. The position needs to be accurate to within about +/- 0.5 degree in latitude and longitude, and the clock offset needs to be correct within about +/- 20s. You may already have accurate enough starting estimates if you know the position where the animal was tagged, the tag was configured to start soon after (so the animal couldn't have gone far), and you configured the tag on a computer with the correct time and time-zone. In which case, you define the starting position as:

```
pos = [your_latitude, your_longitude] ;
tc = 0 ;                                % no clock offset
```

and proceed to step 3. Latitude and longitude are always in decimal degrees and are negative for southern hemisphere latitudes and for longitudes west of Greenwich. So, for example, Chatham on Cape Cod is at *pos=[41.7,-69.9]*. One decimal point of accuracy is fine for the starting position.

Most likely you know the rough start position (e.g., within 0.5 degree) but are not sure about the clock offset. In which case, do this:

[tc,rerr] = gps_timesearch(OBS,pos,[-30,30],200)

In this line, *pos* is your starting position estimate and *[-30,30]* defines the clock offset time range to search, i.e., -30 to +30 seconds with respect to the true time. The outputs are:
tc is an estimate of the time offset between the tag clock and GPS time, in seconds.
rerr is an estimate of the location error (in metres) that will result in the first GPS location if you use this clock offset. If *rerr* is less than a few hundred metres, give *tc* a try in step 3 below. If *rerr* is high, then either your starting position estimate is not good or you need to allow a larger/different time offset search.

A +/- 30 second search should be plenty for typical clock offsets that come about from incorrect setting of the time on the computer that you use to configure the tag. However, the computer must be set to the local time AND to the correct time zone and needs to correctly account for any change of hour due to summer/winter time. To check this is correctly configured, go into the computer's date and time tool (usually at the bottom right of the window for Windows PCs) and select 'Change date and time settings'. If the time zone is correct for your area, the summer/winter time changes will also be handled properly. Remember that it is the last computer you use to talk to the tag before a deployment that sets the time on the tag.

If you suspect that the computer was set to the wrong time-zone, you will need to do a wider search. For example, if you think that the computer was set one hour ahead or behind the correct time, you may need to check both around +3600 seconds and -3600, i.e.:

[tc,rerr] = gps_timesearch(OBS,[lat,long],3600+[-30,30],200)

[tc,rerr] = gps_timesearch(OBS,[lat,long],-3600+[-30,30],200)

If you are still not getting small enough *rerr*, one of the following steps might be needed.

If you really don't know within +/- 2 degrees what the start position is, do the following:

check_sv_elevation(OBS,latr,longr,tcr) ;

Here *latr, longr* and *tcr* are vectors defining the ranges of values to search over. Each vector contains two values, e.g., the minimum latitude and the maximum latitude in the case of *latr.* For example you could use: *latr=[47,53], longr=[-64,-74],tcr=[-60,60]*. The larger the area and time span over which you search, the longer it will take. *check_sv_elevation* looks at which satellites should have been visible over the first 30 mins to 1 hour of GPS captures and compares these to the satellites that were actually detected. It will give you an idea of the most likely starting point with +/-1 degree resolution. This can be improved further as follows.

If you know the starting point within +/- 3 degrees (e.g., as a result of running *check_sv_elevations*, run the following:

find_start_point(OBS,latr,longr,tcr) ;

As before *latr*, *longr* and *tcr* are vectors defining the ranges of values to search over. This function is slower and more precise than *check_sv_elevation* and so needs to be used with a smaller search range. If you get a likely starting point for the GPS track from *find_start_point*, you can then try to get the time offset using *gps_timesearch* as above.

3. Compute positions
Armed with an estimate of the start position (*pos*) and clock offset (*tc*), you can run the GPS processor on the pseudo-ranges to compute the track. This can take several hours if there were a lot

of captures. The function plots the decoded positions as it goes but these can include the occasional outlier due, e.g., to insufficient satellites in a capture. These inaccurate positions are automatically detected and removed from the final track.

[POS,N,gps] = gps_posns(OBS,pos,tc);

This function returns three things:
POS is a structure containing the latitude, longitude and time of each reliable decoded GPS position. The latitude and longitude are in decimal degrees and the time is in the Matlab/Octave datenum format (a single number that can be used to encode date and time).
N and GPS are matrices/structures that contain all of the decoded positions including the unreliable ones, and some information on accuracy. You will likely not need these but you should save them just in case (see step 4 below).

To plot the track do:

plot(POS.lon,POS.lat,'g.-')

To add a background map showing coastline, places and topography in the area, do:

plot_google_maps('MapType','hybrid')

You may need to force the map to cover a larger area if the animal is far off-shore and you want to see the nearest land-mass on the map. To do this, enter the longitude and latitude limits you want the map to have using:

axis([lon_min,lon_max,lat_min,lat_max])

and then re-run the plot_google_maps line to refresh the map.

4. Save the result.
First save all of the outputs of *gps_posns* in a *.mat* file in case you need to check them later:

save([depid,'trk.mat'],'POS','N','gps')

This will make a file in your current working directory called the same as your deployment id but with a '*trk.mat*' ending.

Then generate a *.nc* file for the GPS track data. For this you need the *info* structure that you made in Section 3. If you need to read this in from a .nc file, use *load_nc*.

First we need to convert the absolute time stamps in POS into time in seconds since the start of the tag deployment. This makes it easier to connect the GPS positions with the other data collected by the tag:

gpst = gps2tag_time(POS.T, info) ;   % seconds since 'tag-on' for each gps position

Now generate a data structure with metadata the same way as for pressure and acceleration:

POS = sens_struct([POS.lat,POS.lon], gpst, depid, 'pos') ;

Note that instead of a sensor sampling rate, we passed the time of each track point to *sens_struct* because the track is irregularly sampled.

You could use *add_nc* to save POS in the *.nc* file for this deployment or you could make a new *.nc* file containing just the track using:

save_nc([depid,'trk'], info, POS)

This will make a file called the same as the deployment id but with a '*trk.nc*' ending in your current working directory.

8. Quick guide

Here is a summary of the Matlab/Octave instructions to read in and work with sensor data. These lines assume that you have done all of the initializing steps discussed above, i.e., you have got the latest tools from the website, put them on your path and you have edited the *species.csv* and *researchers.csv* files.

1. Make a deployment id and define the raw data directory, e.g.:

```
depid = 'mn18_175d';
recdir = 'e:/mn18/mn18_175d';
```

2. Read in the sensor data:

```
X = d3readswv_commonfs(recdir,depid,5);        % for 5 Hz data
```

3. Make and edit the info structure (it's a good idea to put the following lines in a script because they probably don't change over an experiment, fieldseason or cruise):

```
info = make_info(depid,'D4',depid(1:2),'me');     % 'me' should be your initials
info.dephist_deploy_locality = 'Cape Cod, Massachusetts';
info.dephist_deploy_method = 'suction cup';
info.dephist_deploy_location_lat = 41.7;
info.dephist_deploy_location_lon = -69.9;
info. ....                                        % whatever else you want to add
```

4. Form sensor structures for each of the sensor types that you want to work with (change the channel numbers if your data has a different order of sensor data in *X.x*):

```
T = sens_struct(X.x{7},X.fs(7),depid,'temp');
P = sens_struct(X.x{8},X.fs(8),depid,'pres') ;
A = sens_struct([X.x{1:3}],X.fs(1),depid,'acc') ;
M = sens_struct([X.x{4:6}],X.fs(4),depid,'mag') ;
```

5. Get the calibration information. This is one of the few tools that is specific to D4 tags:

```
CAL = d4findcal(recdir,depid);
```

6. Apply the calibration data to the sensor channels and check if they need fine-tuning:

```
Tc = do_cal(T,CAL.TEMP);
Pc = do_cal(P,CAL.PRESS,'T',Tc) ;
[AA,ac] = auto_cal_acc(A,CAL.ACC) ;
CAL.ACC = ac ;
Ac = do_cal(A,CAL.ACC) ;              % Ac is now in the tag frame
```

To fix offsets and temperature sensitivity in the pressure sensor, try one of these:

```
[Pf,pc] = fix_pressure(Pc,Tc);
Pf = fix_offset_pressure(Pc,600,1200);        % adjust the numbers to make it work well
```

Save the improved calibration data for the deployment

```
save([depid,'cal.mat'],'CAL')
```

7. Save the data and metadata to a *.nc* file

```
ncname = [depid,'sens5'] ;            % pick a suitable name
save_nc(ncname,info,Tc,Pc,Ac) ;       % replace Pc with Pf if you had to fix the pressure
```

9. Converting calibrated data from DTAG-2 to .nc files

If you want to convert *'prh'* files that you have already made for DTAG-2 deployments into the new *.nc* format, it is really easy to do with the following steps. It may be quickest to combine these steps into a script for each field season then you just need to change the depid and re-run it for each deployment from that season.

```
depid = 'pw04_297f'                   % pick the deployment you want to process
```

From here on, everything is automatic (once you have edited a few things!):

```
[c,tagon,s,ttype,f,tagid]=tagcue(0,depid) ;    % get some information on the deployment
loadprh(depid) ;                  % pick the PRH file you want, if there are multiple versions
```

Make the metadata (replace xx with the initials of the data owner, making sure that they are listed in researchers.csv):

```
info = make_info(depid,'D2',depid(1:2),'xx') ;
```

Edit the fields below so that they reflect the deployment method and location etc:

```
info.dephist_deploy_method = 'suction cup';
info.dephist_deploy_locality = 'Tenerife, Canary Is';
```

Add some more fields to info saying where the data came from:

```
info.dtype_source = sprintf('%sprh%d',depid,fs) ;     % indicate the file that the data came from
info.dtype_nfiles = 1 ;
info.dtype_format = 'mat' ;
info.device_serial = num2str(tagid) ;
info.dephist_device_datetime_start = datestr(tagon,info.dephist_device_regset) ;
info.dephist_deploy_datetime_start = datestr(tagon,info.dephist_device_regset) ;
```

Convert the data matrices into sensor structures:

```
P = sens_struct(p,fs,depid,'pres') ;
A = sens_struct(9.81*Aw,fs,depid,'acc') ;      % note 9.81* to convert from g's to m/s2
M = sens_struct(Mw,fs,depid,'mag') ;
```

Indicate in the metadata that the accelerometer and magnetometer data are in the animal frame:

```
A.frame = 'animal' ;
M.frame = 'animal' ;
```

Save the data and metadata to a .nc file:

```
ncname = sprintf('%ssens%d',depid,fs) ;      % make a name for the .nc file
save_nc(ncname,info,P,A,M) ;                 % save the data
```

10. Updating the attributes in a tag

TBD

11. Troubleshooting

1. Some of the tools that read in the raw data files (e.g., *d3readswv*, *d3audit*) generate helper files to make subsequent calls faster. These files are stored in your current Matlab/Octave directory and start with an underscore character, e.g., *'_mn18_175ddir.txt'*. You don't normally need to do anything with these files - they are just for internal use of the tools. However, if you change the raw data either by adding or removing files, these helper files will no longer be current. This might happen because you get more data for the same deployment or because you find that some of the data files were recorded when the tag was not on the animal. To force the software tools to re-read the raw data and re-generate the helper files, you do this:

```
clear_cues(depid)
```

2. If *d3readswv* or any of the tools that require a *recdir* are not working, check that you have the right directory and that the hard drive is plugged in if you are using an external drive. Windows arbitrarily changes the drive letter it assigns to drives (e.g., from 'e:' to 'f:') so check in the Explorer window or in 'My Computer' to find out what letter it is calling your drive.

3. Make sure you have the latest Matlab/Octave tools and that you have added their paths (including sub-folders) to the Matlab/Octave search path.