readgssi Documentation

Ian Nesbitt

May 04, 2021

TUTORIAL

Welcome to readgssi's documentation. This program was written to read and process ground-penetrating radar files from instruments made by Geophysical Survey Systems Incorporated (GSSI), although I have no affiliation with nor endorsement for the afforementioned organization.

readgssi is a tool intended for use as an open-source reader and preprocessing module for subsurface data collected with GSSI ground-penetrating radar (GPR) devices. It has the capability to read DZT and DZG files with the same pre-extension name and plot the data contained in those files.

CHAPTER

ONE

INSTALLING

1.1 Requirements

I strongly recommend installing the following dependencies via Anaconda (https://www.anaconda.com/distribution/).

- `obspy` (https://docs.obspy.org)
- `numpy` (https://docs.scipy.org/doc/numpy)
- `scipy` (https://docs.scipy.org/doc/scipy/reference)
- `matplotlib` (https://matplotlib.org)
- `pandas` (https://pandas.pydata.org/pandas-docs/stable)
- `h5py` (http://docs.h5py.org/en/stable)

Those that are not available via the Anaconda installer are available on the Python Package Index (PyPI):

- `pynmea2` (https://github.com/Knio/pynmea2)
- `geopy` (https://geopy.readthedocs.io/en/stable/)
- `pytz` (https://pythonhosted.org/pytz/)

Back to top ↑

1.2 Installation guide

Note: This does not cover the installation of Anaconda, as it may differ depending on your system, and there are many excellent resources out there that can explain far better than me for your system of choice. Start with the Anaconda installation guide.

Note: The console commands outlined here use Linux bash script. Mac users should be able to use all the same commands as I do, but Windows users will need to install and understand the Windows Subsystem for Linux (WSL) in order to execute these commands successfully. If you'd like information about installing and using WSL, see this guide for more details: https://docs.microsoft.com/en-us/windows/wsl/install-win10

1.2.1 Installing from PyPI

PyPI is the Python Package Index.

Open a Terminal interface (UNIX) or the Anaconda Prompt (Windows) and make sure Anaconda works:

```
conda --version
```

You should get output that displays the conda version (4.6.13 in this case). If not, *please see note 1 above*.

Once you have conda running, installing requirements is pretty easy. All dependencies are available through conda or pip.

```
conda config --add channels conda-forge
conda create -n readgssi python==3.7 pandas h5py pytz obspy
conda activate readgssi
pip install readgssi
```

That should allow you to run the commands in the next section (*General usage*).

Note: This code is doing a couple important things so if you're unfamiliar with python and/or terminal commands, let's go over what they are. `conda config --add channels conda-forge` tells conda to look in the conda user code repository called "Conda Forge". ObsPy and a lot of other user-created code lives in the Forge. Next, `conda create -n readgssi` creates a virtual environment (more on that in a second).

We tell conda what software to put in that virtual environment using the rest of the line (`python==3.7 pandas h5py pytz obspy`). We want python 3.7 specifically (hence `python==3.7`), and then the latest release of pandas, h5py, pytz, and obspy. This will install several other dependencies, notably numpy which is the library we really care about because it allows us to do math on arrays.

Then, we activate our virtual environment using `conda activate readgssi` which allows us to operate in a "virtual environment" which is basically a python space where you can install dependencies without messing with the functionality of python on the rest of your machine. Now that we're in the virtual environment, we can install things using `pip`, the python package manager. `pip install readgssi` will install the readgssi version available on the Python Package Index (PyPI) into your readgssi environment, but nowhere else. This is useful but can be confusing: if you try to run readgssi from outside the virtual environment you just made, you will not be able to find it! The reason it's useful is that it doesn't modify the version of python or packages that your computer may use for system tasks (no one likes obscure errors, so we try to avoid them. . . and one of the best ways of doing that is by using virtual environments). To get back into the readgssi environment you created, simply do `conda activate readgssi`.

Back to top ↑

1.2.2 Installing from source

If you choose to install a specific commit rather than the latest working release of this software, I recommend doing so via the following commands:

```
conda config --add channels conda-forge
conda create -n readgssi python==3.7 pandas h5py pytz obspy
conda activate readgssi
pip install git+https://github.com/iannesbitt/readgssi
```

If you plan on modifying the code and installing/reinstalling once you've made changes, you can do something similar to the following, assuming you have conda dependencies installed:

```
cd ~
git clone https://github.com/iannesbitt/readgssi

# make code changes if you wish, then:

pip install ~/readgssi
```

Back to top ↑

1.2.3 Installing onto armv7l architecture

This has not been tested (though will be in the future), but installing on the Raspberry Pi and other ARM processors should be possible in theory. Start with this:

```
# from https://github.com/obspy/obspy/wiki/Installation-on-Linux-via-Apt-Repository
deb http://deb.obspy.org stretch main
wget --quiet -O - https://raw.github.com/obspy/obspy/master/misc/debian/public.key |␣
↪sudo apt-key add -
sudo apt-get update
sudo apt-get install python-obspy python3-obspy
sudo apt-get install ttf-bistream-vera
rm -rf ~/.matplotlib ~/.cache/matplotlib
sudo apt-get install python-pandas python-h5py
pip install -U pytz pynmea2 geopy readgssi
```

Todo: Install and test readgssi on armv7l architecture

Back to top ↑

1.3 Testing

There is no testing module as such yet, but a simple test will ensure that most things are working properly:

```
readgssi -V  # this will display the version
readgssi -h  # this will display the help text
```

If it's working, head over to *General usage*.

Todo: Create a testing module and routines.

Back to top ↑

CHAPTER

TWO

GENERAL USAGE

`readgssi` can be run straight from a `bash` console, using a python interpreter like `ipython` or the python console, or scripted in a development environment like Jupyter Notebook, Spyder, or Pycharm. Usage of `bash` is covered in *bash usage*, while usage in python is covered below in *Python usage*.

Note: In the first part of this tutorial, I will separate bash and python operations, but towards the end I will bring them together, as they will produce nearly identical outputs. However, I felt it pertinent to separate the two at the start, since some may know either bash or python but not both. Newcomers to either one will note that both have different benefits over the other, which is why I develop most of the functionality of this program to be accessible from both.

bash is useful for coordinating calls to multiple files in a directory in for loops, which has distinct uses for processing large amounts of files in a much shorter amount of time than RADAN. bash's range of interoperability is much narrower, but it is very good at processing a number of things in a row using similar patterns of parameters.

Python is useful for its ability to hold objects in memory, and to pass objects to and from various functions. Python's range is broader in terms of array manipulation and object passing.

2.1 Python usage

Most python functionality is covered under the modules in the left panel, and in the following sections. I will cover the very basics here.

The `readgssi.readgssi.readgssi()` covers a large portion of the use cases you are likely to want out of readgssi. A properly formulated command is long but should return what you want. In the future, radar arrays will be python classes, which will make things easier.

Simply printing a file's header information to output is easy:

```
>>> from readgssi import readgssi
>>> readgssi.readgssi(infile='DZT__001.DZT', frmt=None, verbose=True)
2019-07-22 16:56:20 - reading...
2019-07-22 16:56:20 - input file:         DZT__001.DZT
2019-07-22 16:56:20 - WARNING: no time zero specified for channel 0, defaulting to 2
2019-07-22 16:56:20 - success. header values:
2019-07-22 16:56:20 - system:             SIR 4000 (system code 8)
2019-07-22 16:56:20 - antennas:           ['3207', None, None, None]
2019-07-22 16:56:20 - time zeros:         [2, None, None, None]
2019-07-22 16:56:20 - ant 0 center freq:  100 MHz
2019-07-22 16:56:20 - date created:       2017-07-25 18:21:24+00:00
2019-07-22 16:56:20 - date modified:      2018-08-06 17:02:24+00:00
2019-07-22 16:56:20 - gps-enabled file:   yes
```

(continues on next page)

(continued from previous page)

```
2019-07-22 16:56:20 - number of channels: 1
2019-07-22 16:56:20 - samples per trace:  2048
2019-07-22 16:56:20 - bits per sample:    32 signed
2019-07-22 16:56:20 - traces per second:  24.0
2019-07-22 16:56:20 - traces per meter:   300.0
2019-07-22 16:56:20 - epsr:               80.0
2019-07-22 16:56:20 - speed of light:     3.35E+07 m/sec (11.18% of vacuum)
2019-07-22 16:56:20 - sampling depth:     33.5 m
2019-07-22 16:56:20 - "rhf_top":          3.4 m
2019-07-22 16:56:20 - offset to data:     131072 bytes
2019-07-22 16:56:20 - traces:             28343
2019-07-22 16:56:20 - seconds:            1180.95833333
2019-07-22 16:56:20 - array dimensions:   2048 x 28343
2019-07-22 16:56:20 - beginning processing for channel 0 (antenna 3207)
>>>
```

Note here that there is a warning regarding the time-zero. That can be set using `zero=[int]`, but won't really come into play until the next section.

See *Reading radar data* for next steps.

Back to top ↑

2.2 bash usage

`readgssi` comes with a UNIX command line interface, for easy bash scripting. This is very useful when processing folders full of many files. If you'd like a full description of all options, enter:

```
readgssi -h
```

You should see readgssi output its help text, which will display options like those below, but in a more condensed form.

Note: Each option flag here below passed to `readgssi.readgssi.readgssi()` after the command has been processed by `readgssi.readgssi.main()`.

Usage:

```
readgssi -i input.DZT [OPTIONS]
```

2.2.1 Required flags

-i file, --infile=file Input DZT file.

2.2.2 Optional flags

-o file, --outfile=file Output file. If not set, the output file will be named similar to the input. See *readgssi.functions.naming()* for naming convention details.

-f str, --format=str Output file format (eg. "csv", "numpy", "gprpy"). See *readgssi.translate*.

-p int, --plot=int Tells `readgssi.plot.radargram()` to create a radargram plot <int> inches high (defaults to 7).

-D int, --dpi=int Set the plot DPI in `readgssi.plot.radargram()` (defaults to 150).

-T, --titleoff Tells `readgssi.plot.radargram()` to turn the plot title off.

-x m, --xscale=m X units for plotting. Will attempt to convert the x-axis to distance, time, or trace units based on header values. See `readgssi.plot.radargram()` for scale behavior. Combine with the `-N` option to enable distance normalization, or `-d int` to change the samples per meter.

-z m, --zscale=m Z units for plotting. Will attempt to convert the x-axis to depth, time, or sample units based on header values. See `readgssi.plot.radargram()` for scale behavior. Combine with the `-E int` option to change the dielectric.

-n, --noshow Suppress matplotlib popup window and simply save a figure (useful for multi-file processing).

-c str, --colormap=str Specify the colormap to use in radargram creation function `readgssi.plot.radargram()`. For a list of values that can be used here, see https://matplotlib.org/users/colormaps.html#grayscale-conversion

-g int, --gain=int Gain constant (higher=greater contrast, default: 1).

-r int, --bgr=int Horizontal background removal (useful to remove ringing). Specifying 0 as the argument here sets the window to full-width, whereas a positive integer sets the window size to that many traces after stacking.

-R, --reverse Reverse (flip array horizontally) using *readgssi.arrayops.flip()*.

-w, --dewow Trinomial dewow algorithm (experimental, use with caution). For details see *readgssi.filtering.dewow()*.

-t int-int, --bandpass=int-int Triangular FIR bandpass filter applied vertically (positive integer range in megahertz; ex. 70-130). For details see *readgssi.filtering.triangular()*.

-b, --colorbar Adds a `matplotlib.colorbar.Colorbar` to the radar figure.

-a, --antfreq=int Set the antenna frequency. Overrides header value in favor of the one set here by the user.

-s, --stack=int Set the trace stacking value or "auto" to autostack, which results in a ~2.5:1 x:y axis ratio.

-N, --normalize Distance normalize. *readgssi.gps.readdzg()* reads the .DZG NMEA data file if it exists, otherwise tries to read CSV with lat, lon, and time fields. Then, the radar array and GPS time series are passed to *readgssi.arrayops.distance_normalize()* where the array is expanded and contracted proportional to the distance traveled between each GPS distance mark. This is done in chunks to save memory.

-d float, --spm=float Specify the samples per meter (SPM). Overrides header value. Be careful using this option on distance-naive files, and files in which "time" was used as the main trigger for trace shots!

-m, --histogram Produces a histogram of data values for each channel using `readgssi.plot.histogram()`.

-E float, --epsr=float User-defined epsilon_r (sometimes referred to as "dielectric"). If set, ignores value in DZT header in favor of the value set here by the user.

-Z int, -Z list, --zero=int, --zero=list Timezero: skip this many samples before the direct wave arrives at the receiver. Samples are removed from the top of the trace. Use a four-integer list format for multi-channel time-zeroing. Example: `-Z 40,145,233,21`.

Command line functionality is explained further in the following sections.

Back to top ↑

CHAPTER

THREE

READING RADAR DATA

3.1 Reading with Python

3.1.1 Simplest usage (reading the header)

As mentioned in the previous section, you can use `readgssi.readgssi.readgssi()` to output some of the header values: * name of GSSI control unit * antenna model * antenna frequency * samples per trace * bits per sample * traces per second * L1 dielectric as entered during survey * sampling depth * speed of light at given dielectric * number of traces * number of seconds * . . . and more. In all likelihood, more than you need or want to know. However if you feel there is something important I'm leaving out, I'd be happy to include it. Open a github feature request issue and let me know what you would like to see.

Printing a file's header information to output is easy. Use `frmt=None` and `verbose=True`.

```
>>> from readgssi import readgssi
>>> readgssi.readgssi(infile='DZT__001.DZT', frmt=None, verbose=True)
2019-07-22 16:56:20 - reading...
2019-07-22 16:56:20 - input file:         DZT__001.DZT
2019-07-22 16:56:20 - WARNING: no time zero specified for channel 0, defaulting to 2
2019-07-22 16:56:20 - success. header values:
2019-07-22 16:56:20 - system:             SIR 4000 (system code 8)
2019-07-22 16:56:20 - antennas:           ['3207', None, None, None]
2019-07-22 16:56:20 - time zeros:         [2, None, None, None]
2019-07-22 16:56:20 - ant 0 center freq:  100 MHz
2019-07-22 16:56:20 - date created:       2017-07-25 18:21:24+00:00
2019-07-22 16:56:20 - date modified:      2018-08-06 17:02:24+00:00
2019-07-22 16:56:20 - gps-enabled file:   yes
2019-07-22 16:56:20 - number of channels: 1
2019-07-22 16:56:20 - samples per trace:  2048
2019-07-22 16:56:20 - bits per sample:    32 signed
2019-07-22 16:56:20 - traces per second:  24.0
2019-07-22 16:56:20 - traces per meter:   300.0
2019-07-22 16:56:20 - epsr:               80.0
2019-07-22 16:56:20 - speed of light:     3.35E+07 m/sec (11.18% of vacuum)
2019-07-22 16:56:20 - sampling depth:     33.5 m
2019-07-22 16:56:20 - "rhf_top":          3.4 m
2019-07-22 16:56:20 - offset to data:     131072 bytes
2019-07-22 16:56:20 - traces:             28343
```

(continues on next page)

(continued from previous page)

```
2019-07-22 16:56:20 - seconds:            1180.95833333
2019-07-22 16:56:20 - array dimensions:   2048 x 28343
2019-07-22 16:56:20 - beginning processing for channel 0 (antenna 3207)
>>>
```

Note here that there is a warning regarding the time-zero. That can be set using `zero=[int]`, as below.

3.1.2 Reading to Python objects

Now, we'll be reading the file into python objects using `readgssi.readgssi.readgssi()`.

If you would like to return the header, radar array, and gps info (if it exists), and set time-zero to 233 samples, the command is simpler. Here, we drop `verbose=True`, and `frmt=None`, which suppresses console output and causes python objects to be returned:

```
>>> hdr, arrs, gps = readgssi.readgssi(infile='DZT__001.DZT', zero=[233])
>>> type(hdr)
<class 'dict'>
>>> type(arr[0])
<class 'numpy.ndarray'>
>>> type(gps)
<class 'pandas.core.frame.DataFrame'>
```

If no GPS file exists, you will get a soft error printed to the console, like this, and the `gps` variable will be `False`:

```
>>> hdr, arrs, gps = readgssi.readgssi(infile='DZT__002.DZT', zero=[233])
2019-07-22 17:28:43 - WARNING: no DZG file found for GPS import
>>> print(gps)
False
```

No valid GPS file means that you will not be able to distance normalize the array using `normalize=True`. If you do happen to have a valid GPS file to normalize with, skip to *Processing radar arrays* to learn how to do it.

Back to top ↑

3.2 Reading with bash

Same as above, you can print a host of information about the DZT specified with a simple command.

From a unix/linux/mac command line or Windows Anaconda Prompt, type:

```
$ readgssi -i DZT__001.DZT
2019-07-22 16:56:20 - reading...
2019-07-22 16:56:20 - input file:         DZT__001.DZT
2019-07-22 16:56:20 - WARNING: no time zero specified for channel 0, defaulting to 2
2019-07-22 16:56:20 - success. header values:
2019-07-22 16:56:20 - system:             SIR 4000 (system code 8)
2019-07-22 16:56:20 - antennas:           ['3207', None, None, None]
2019-07-22 16:56:20 - time zeros:         [2, None, None, None]
2019-07-22 16:56:20 - ant 0 center freq:  100 MHz
2019-07-22 16:56:20 - date created:       2017-07-25 18:21:24+00:00
2019-07-22 16:56:20 - date modified:      2018-08-06 17:02:24+00:00
2019-07-22 16:56:20 - gps-enabled file:   yes
2019-07-22 16:56:20 - number of channels: 1
```

(continues on next page)

(continued from previous page)

```
2019-07-22 16:56:20 - samples per trace:  2048
2019-07-22 16:56:20 - bits per sample:    32 signed
2019-07-22 16:56:20 - traces per second:  24.0
2019-07-22 16:56:20 - traces per meter:   300.0
2019-07-22 16:56:20 - epsr:               80.0
2019-07-22 16:56:20 - speed of light:     3.35E+07 m/sec (11.18% of vacuum)
2019-07-22 16:56:20 - sampling depth:     33.5 m
2019-07-22 16:56:20 - "rhf_top":          3.4 m
2019-07-22 16:56:20 - offset to data:     131072 bytes
2019-07-22 16:56:20 - traces:             28343
2019-07-22 16:56:20 - seconds:            1180.95833333
2019-07-22 16:56:20 - array dimensions:   2048 x 28343
2019-07-22 16:56:20 - beginning processing for channel 0 (antenna 3207)
```

Back to top ↑

CHAPTER

FOUR

PLOTTING RADARGRAMS

Plotting is often essential to data manipulation quality control. Here are some basic examples of plotting and plot rendering methods.

I give examples below, but you will quickly realize that a lot of radar data requires at least a little bit of a touchup before it looks presentable. That's covered in the next section, *Processing radar arrays*. Note that some of the examples below will jump ahead to use methods covered in that section.

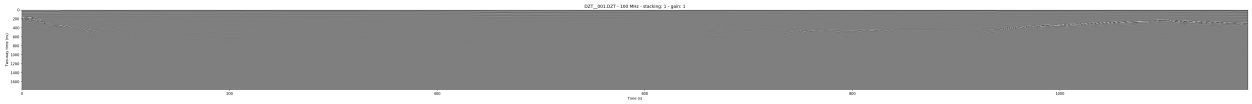
Note: I apologize to metric system users. matplotlib uses inches and dots per inch (DPI) and for consistency's sake I chose to adhere to imperial units for plot size :(

4.1 Basic plotting

4.1.1 Plotting with Python

Plotting in Python just means setting `plot=7` or another integer, which represents the vertical size in inches. In this simple example, we use the `zero=[233]` flag to get rid of the part of the radargram from before the direct wave meets the receiver.

```
from readgssi import readgssi
readgssi.readgssi(infile='DZT__001.DZT', outfile='0a.png', frmt=None,
                  zero=[233], plot=5)
```

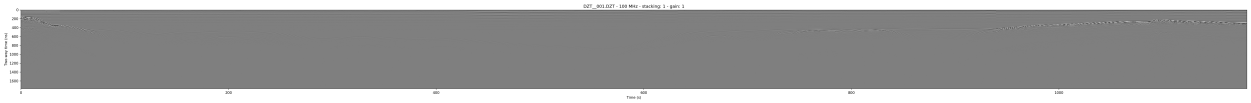


Whoops! That's very long and not very helpful on a standard computer monitor. Let's pretend we've read *Processing radar arrays* and know how to stack arrays horizontally (see *Stacking*), and let's also add some gain to this image as well. (Jump to *Setting gain*)

4.1.2 Plotting with bash

Plotting on the command line is easy. The most basic plotting routine is accessible just by setting the -p flag and specifying a plot height in inches (`-p 5`). Here, we also use a zero of 233 samples (`-Z 233`).

```
readgssi -i DZT__001.DZT -o 0a.png -Z 233 -p 5
```



Whoops! As you notice in the Python example above, this file is very long, which makes viewing tough on a screen (but may be what you want for figure creation).

Back to top ↑

4.2 Setting gain

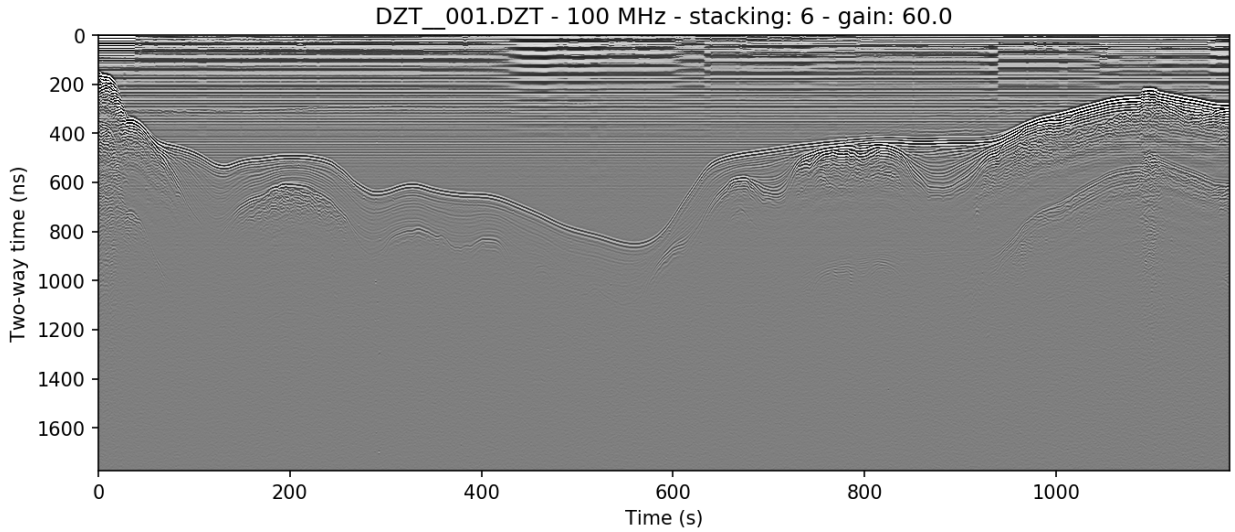
Gain is added using the `gain=int` setting. Let's set that to 60, since this is a lake profile and radar waves attenuate quickly in water. Here, Python and bash examples are given together.

Note: The gain parameter can also be set to a float value between 0 and 1 in order to reduce gain.

Note: This command sets the stacking parameter to "auto", which is explained in *Stacking*.

```
readgssi.readgssi(infile='DZT__001.DZT', outfile='0b.png', frmt=None,
                  zero=[233], plot=5, stack='auto', gain=60)
```

```
readgssi -i DZT__001.DZT -o 0b.png -Z 233 -p 5 -s auto -g 60
```



Wow, looking much better! Now let's see if we can display depth units on the Z-axis.

Back to top ↑

4.3 Changing axis units

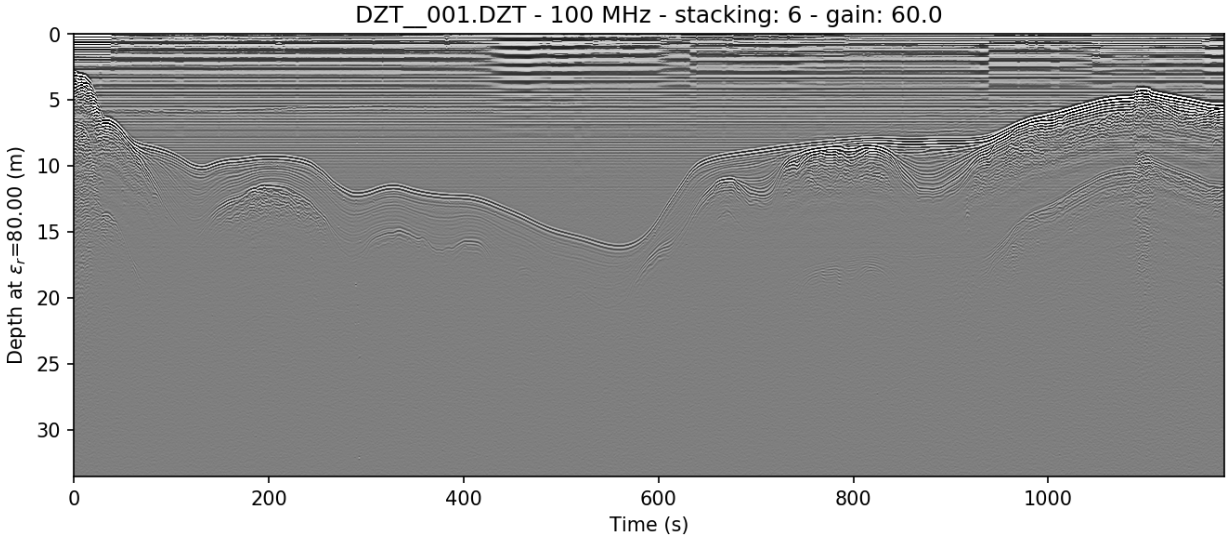
4.3.1 Z axis

The Z axis can be displayed in three different unit types: samples, time, and distance. By default, it will display in nanoseconds (ns). The possible values for time display are "temporal", "time", "nanoseconds", and "ns". Setting the `z` parameter to "samples" sets the axis to display the number of samples (cells) on that axis.

To set the Z-axis to display material depth, we use two separate flags: `epsr=80` or `-E 80` — which modifies the wave velocity by setting the dielectric to roughly that of water at 20 degrees C — and `z='m'` or `-z m`, which sets the z-axis to use those units to calculate profile depths. *"m"* stands for *meters*, but you can also specify "meters", "centimeters"/"cm", or "millimeters"/"mm" explicitly.

```
readgssi.readgssi(infile='DZT__001.DZT', outfile='0c.png', frmt=None,
                  zero=[233], plot=5, stack='auto', gain=60,
                  epsr=80, z='m')
```

```
readgssi -i DZT__001.DZT -o 0c.png -Z 233 -p 5 -s auto -g 60 -z m -E 80
```



If you would like to learn how to remove the horizontal noise in the water column of this image, head to *Getting rid of horizontal noise*.

4.3.2 X axis

Warning: Changing the X-axis units is simple as well, but beware that distance units will not be accurate unless the file is either distance normalized, or was recorded with a survey wheel or DMI and has a proper samples per meter value set. See *Distance normalization* for more information.

The X axis can be displayed in time, traces, and distance. By default, it will display in seconds (s). To set this to "traces", which is calculated from the number of samples on the axis prior to stacking, set the `x='traces'` or `-x traces` flag. See the warning above about setting the axis to distance.

Back to top ↑

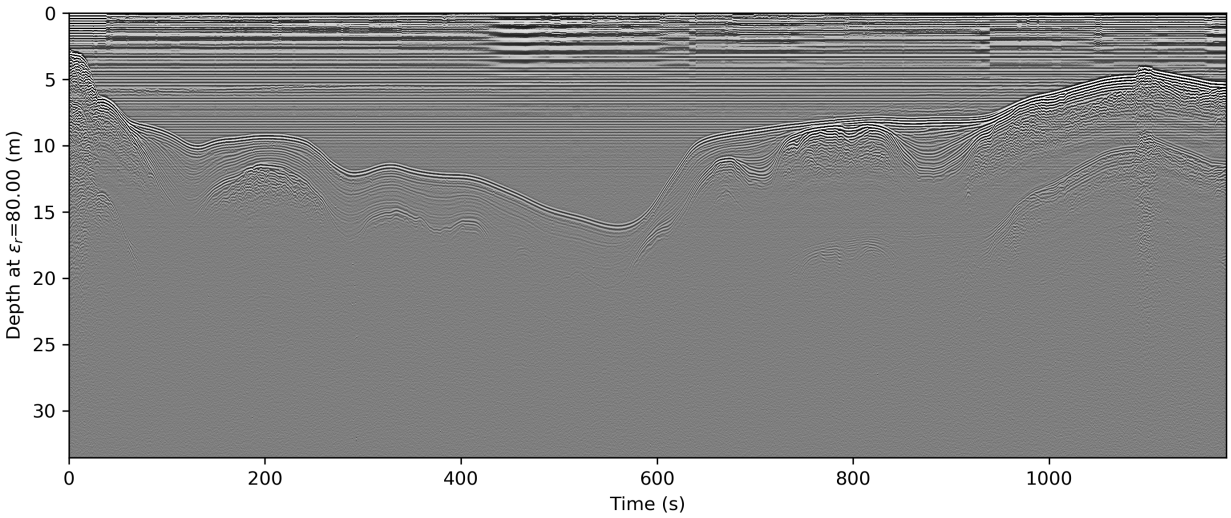
4.4 Making poster-quality figures

Let's say you are really enamored with the way that last figure looks, and you now want to create a figure-quality image for a poster. You'll likely want to drop the title (`title=False` in Python or `-T` in bash), and increase the DPI to something that will work well on a plotter (`dpi=300` in Python or `-D 300` in bash). Pretty simple. Let's see it in action.

Note: I use 300 DPI here to keep file size down, but if you are truly aiming for very high print quality, you may want to increase to 600 DPI to match the capability of most high-end plotters.

```
readgssi.readgssi(infile='DZT__001.DZT', outfile='0d.png', frmt=None,
                  zero=[233], plot=5, stack='auto', gain=60,
                  epsr=80, z='m', title=False, dpi=300)
```

```
readgssi -i DZT__001.DZT -o 0d.png -Z 233 -p 5 -s auto -g 60 -z m -E 80 -T -D 300
```



Back to top ↑

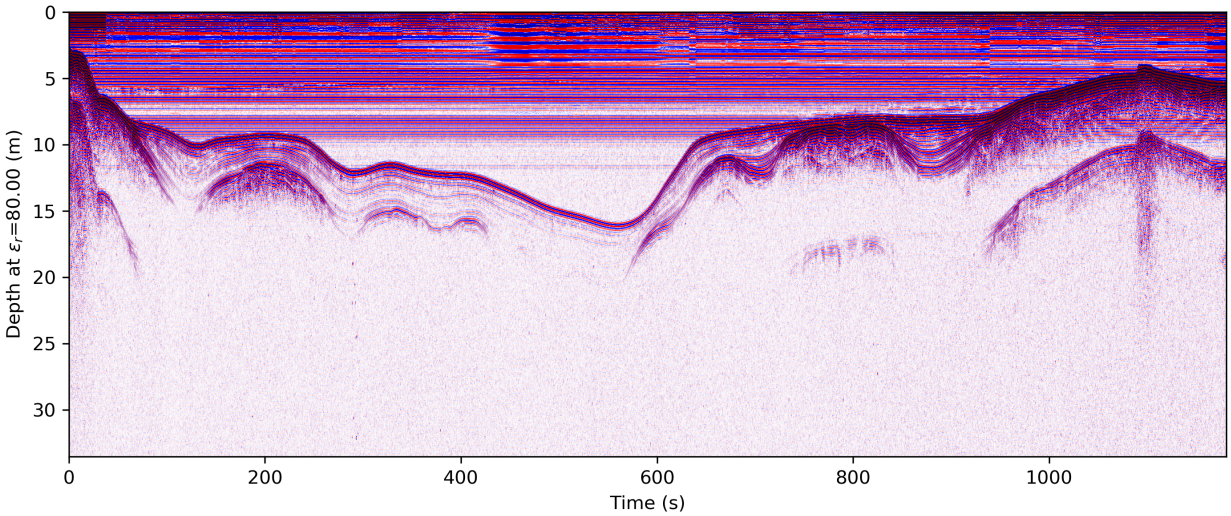
4.5 Changing the colormap

By default, the colormap is Matplotlib's "gray", which is intended to emulate RADAN's default.

Changing colormaps is as simple as specifying a valid `matplotlib.colors.Colormap` to use. For a list of valid colormaps, see the Matplotlib documentation. A popular alternative is "seismic", a diverging blue-white-red colormap used often in sub-bottom seismic surveying.

```
readgssi.readgssi(infile='DZT__001.DZT', outfile='0e.png', frmt=None,
                  zero=[233], plot=5, stack='auto', gain=60,
                  epsr=80, z='m', title=False, dpi=300,
                  colormap='seismic')
```

```
readgssi -i DZT__001.DZT -o 0e.png -Z 233 -p 5 -s auto -g 60 -z m -E 80 -T -D 300 -c␣
↪seismic
```



Changed in version 0.0.16: The default colormap was changed to "gray", because of a previously unnoticed polarity switch in the previous default "Greys".

Back to top ↑

4.6 Suppressing the Matplotlib window

By default, the matplotlib GUI window will display upon successful execution and saving of the radargram, so that you can modify titles and other plot features. To suppress this behavior, set the `noshow=True` or `-n` option.

Because the program will wait for the closure of the Matplotlib window before continuing, this flag is useful for processing folders full of files in bash without user attention.

Note: If plotting is on, readgssi will always save an image, regardless of whether or not the Matplotlib GUI is set to show up. I have found that this behavior makes it easier to save files under the same name but with title and axis label modifications.

This is especially useful when the `outfile` parameter is not set, and the program uses the *`readgssi.functions.naming()`* function to set complex but informative filenames. When saving from the Matplotlib window, click the save button, navigate to the file just saved by the program, then single-click the file name. The save dialog will auto-populate the filename and you can overwrite without the hassle of copying and pasting.

Back to top ↑

CHAPTER

FIVE

PROCESSING RADAR ARRAYS

Note: This section covers some rudimentary (and some more complex) preprocessing methods. Note that these are only a few of the most common methods. If you would like to see another method added here, please open a github issue and briefly explain the method, preferably including the math involved.

5.1 Stacking

Stacking is the process of adding a number of successive neighboring trace columns together, both in order to reduce noise (by cancelling out random variation in neighboring cells) and to condense the X-axis of the radar array. This is very useful in lines with a high number of traces, as it both helps accentuate returns and make the long axis viewable on a reasonable amount of screen space.

The stacking algorithm is available in readgssi by using the `stack=` argument in Python or the `-s` flag in bash. This program contains two methods of stacking: automatic and manual.

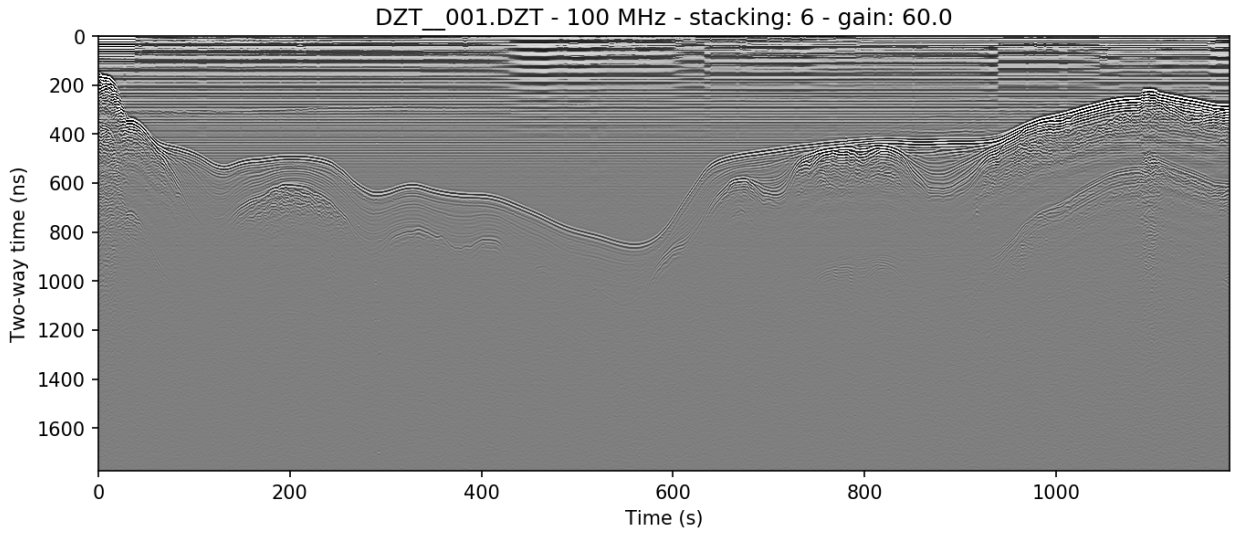
5.1.1 Autostacking

The automatic stacking method checks the ratio of width to height and if it's above 2.5:1, sets the stacking parameter roughly equal to 2.5:1. This can reduce the hassle of trying a number of different stacking values like in RADAN. In Python, this is accessible via the `stack='auto'` argument, while in bash, the flag is `-f auto`.

The file used in *Basic plotting* in the previous section shows the full length of the survey line. Below is the result of autostacking that line and turning the gain up (explained in *Setting gain*).

```
readgssi.readgssi(infile='DZT__001.DZT', outfile='1a.png', frmt=None,
                  zero=[233], plot=5, gain=60, stack='auto')
```

```
readgssi -i DZT__001.DZT -o 1a.png -Z 233 -p 5 -g 60 -s auto
```

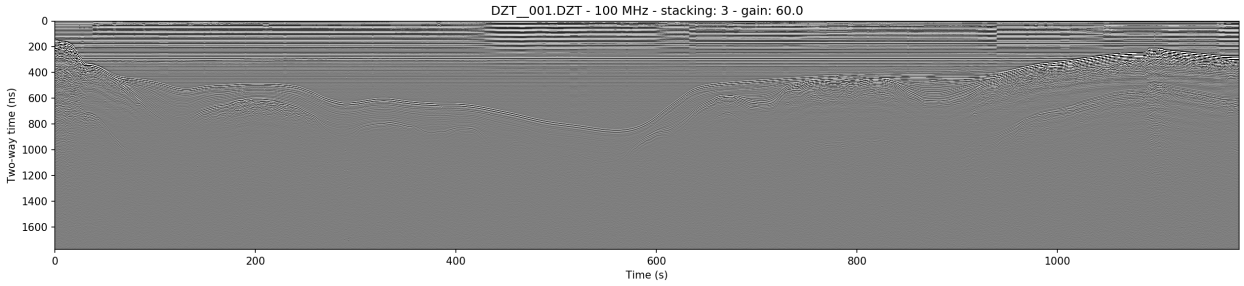


5.1.2 Stacking manually

Sometimes it is preferable to stack a plot a specific number of times determined by the user. Occasionally, you may want to create plots that are longer (have less stacking) or shorter (have more stacking) than the auto method. The example above is stacked 6 times, here we will stack half that amount (i.e. the X-axis will be longer). In python: `stack=3`; in bash: `-s 3`.

```
readgssi.readgssi(infile='DZT__001.DZT', outfile='1b.png', frmt=None,
                  zero=[233], plot=5, gain=60, stack=3)
```

```
readgssi -i DZT__001.DZT -o 1b.png -Z 233 -p 5 -g 60 -s 3
```



Back to top ↑

5.2 Getting rid of horizontal noise

5.2.1 Horizontal average filters (BGR)

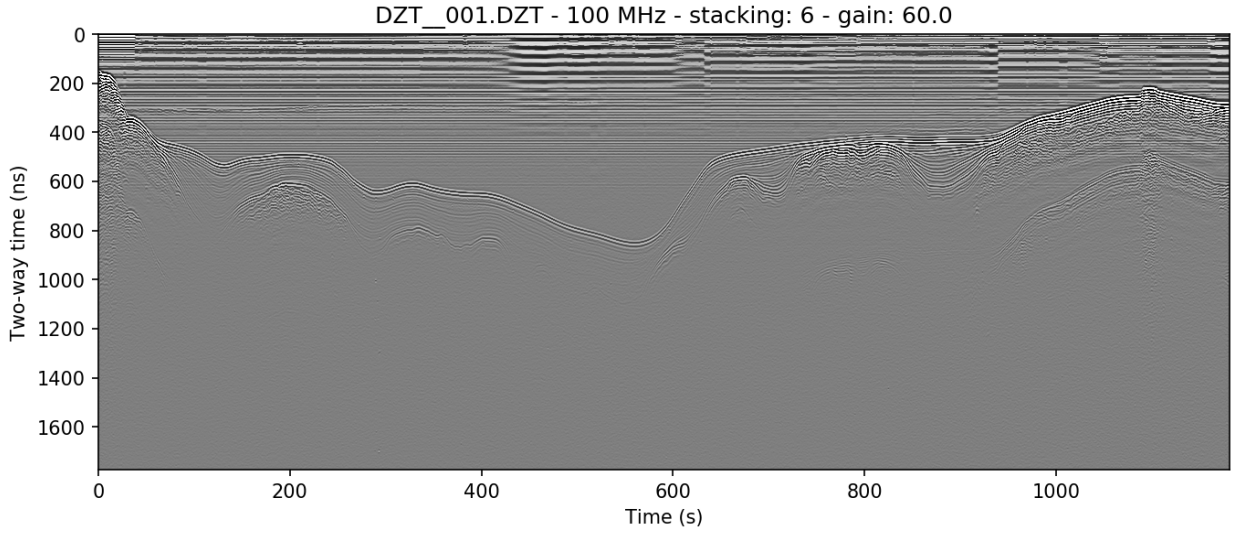
Horizontal average filters, also known as background removal or BGR, are commonly used to remove both low-frequency skew and higher frequency horizontal reverberation banding that can occur in some survey environments. In this program there are two types of BGR: full-width average and moving window average. The former resembles RADAN's simplest BGR algorithm, while the latter emulates its BOXCAR style filter.

Full-width

The full-width BGR filter in readgssi simply takes the average of each row in the array and subtracts that value from the row values themselves, essentially moving their mean value to zero. This can work well in some environments but can cause additional horizontal banding if strongly reflective layers are horizontal for many consecutive traces.

```
readgssi.readgssi(infile='DZT__001.DZT', outfile='2a.png', frmt=None,
                  zero=[233], plot=5, stack='auto', gain=60,
                  bgr=0)
```

```
readgssi -i DZT__001.DZT -o 2a.png -Z 233 -p 5 -s auto -g 60 -r 0
```

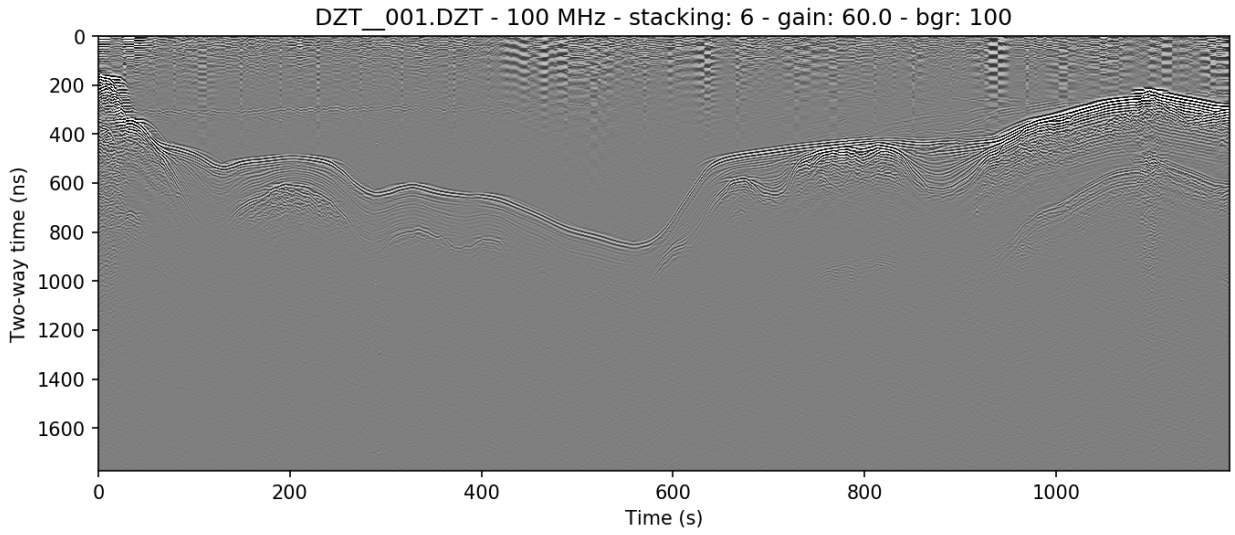


Boxcar/moving window

The boxcar-style method is preferred by many because although it has a tendancy to wipe out data that's too strongly horizontal, it also removes more noise from areas of weak returns and can help make the profile look cleaner. The side effect of this is that it causes artificial wisps on either side of non-horizontal objects, about the size of half the window, and that it can wipe out horizontal layers that are longer than the window length. If you find that it turns horizontal layers into indistinguishable mush, increase the window size and try again.

```
readgssi.readgssi(infile='DZT__001.DZT', outfile='2b.png', frmt=None,
                  zero=[233], plot=5, stack='auto', gain=60,
                  bgr=100)
```

```
readgssi -i DZT__001.DZT -o 2b.png -Z 233 -p 5 -s auto -g 60 -r 100
```



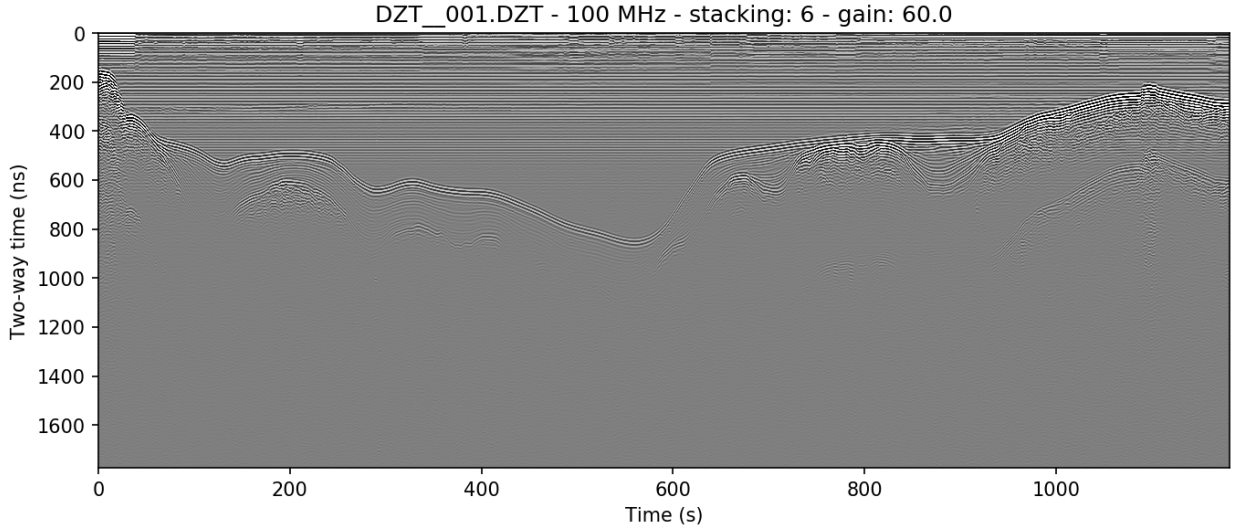
5.2.2 Frequency filter (vertical triangular bandpass)

The vertical filter is more sophisticated and requires proper identification of the antenna's center frequency. Because antennas emit bands of frequencies centered around the manufacturer's specified center frequency, data will often lie within those frequencies. However, noise at other frequency bands is sometimes picked up, whether due to the dielectric of the first layer, or external sources. Often it will be necessary to let pass only the frequencies around the center frequency.

For a 100 MHz antenna, this band can be as wide as 70-130 MHz at low dielectric values. Open water profiles are often much cleaner after being filtered approximately 80% as high as those in higher dielectric media, approximately 60-100 MHz.

```
readgssi.readgssi(infile='DZT__001.DZT', outfile='2c.png', frmt=None,
                  zero=[233], plot=5, stack='auto', gain=60,
                  freqmin=60, freqmax=100)
```

```
readgssi -i DZT__001.DZT -o 2c.png -Z 233 -p 5 -s auto -g 60 -t 60-100
```

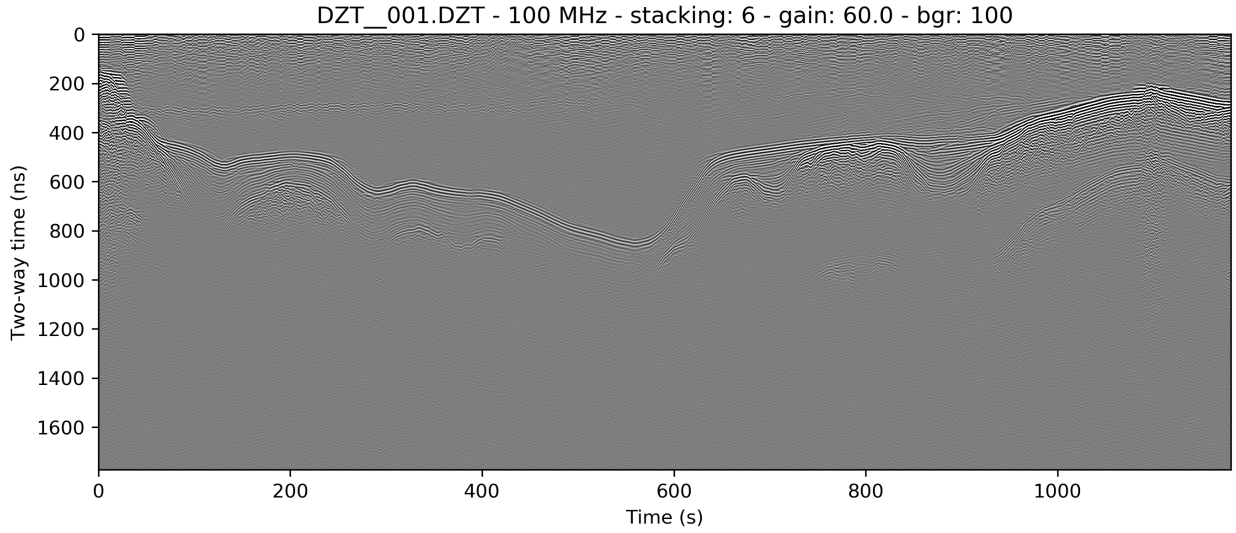


5.2.3 Combining filters

It's typically worthwhile to play with combining filters, as often they can have a compounding effect on cleaning up the profile. See for example what the application of both the horizontal moving window and the vertical triangular filter can do to make the water column of this lake profile look clean enough to see thermoclines:

```
readgssi.readgssi(infile='DZT__001.DZT', outfile='2c.png', frmt=None,
                  zero=[233], plot=5, stack='auto', gain=60, dpi=300,
                  bgr=100, freqmin=60, freqmax=100)
```

```
readgssi -i DZT__001.DZT -o 2c.png -Z 233 -p 5 -s auto -g 60 -D 300 -r 100 -t 60-100
```



Back to top ↑

5.3 Distance normalization

If your files are recorded as time-triggered such as in the case of this lake profile, they need to be distance-normalized before they can be rendered with distance on the X-axis. This can only be done if there is proper GPS information in DZG format.

The relevant function is *readgssi.arrayops.distance_normalize*, accessible with `normalize=True` or `-N`, which calculates the distance traveled between GPS marks and resamples the array to a normalized state, then calculates the new samples per meter value and applies that to the header. The resulting corrected array can be displayed in distance units with `x='m'` or `-x m`.

Warning: Do not use `x='m'` or `-x m` without either a DMI or distance normalization, as the file header samples per meter value could be very wrong (and in some cases will surely be wrong due to how RADAN handles distance, which has known flaws).

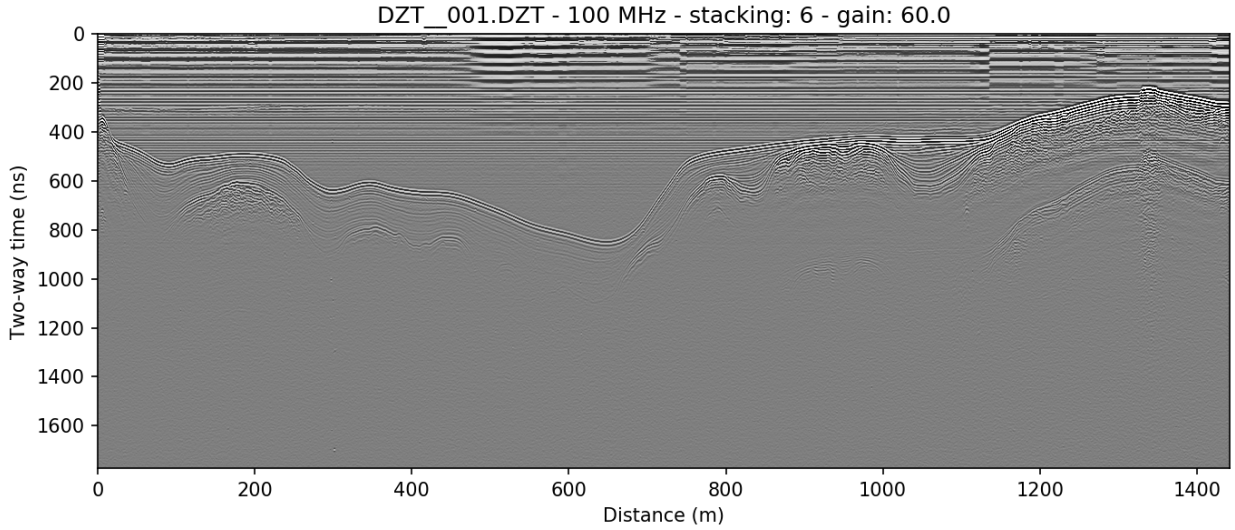
Note: Recording GPS information with a GSSI system that does not have GPS input is not recommended. However, GPS marks can be aligned with user marks in GSSI files if the user can record GPS and radar mark information at the

same time every set number of meters traveled. GPX (GPS exchange format) files with identical marks to GSSI files can be cross-correlated to DZG by using the gpx2dzg software package.

This example distance normalizes and displays the X-axis in meters. Note the change in the beginning of the line, in which the slope appears longer than it really is due to slower survey speed at the start of the line.

```
readgssi.readgssi(infile='DZT__001.DZT', outfile='2c.png', frmt=None,
                  zero=[233], plot=5, stack='auto', gain=60,
                  normalize=True, x='m')
```

```
readgssi -i DZT__001.DZT -o 3a.png -Z 233 -p 5 -s auto -g 60 -N -x m
```



5.3.1 X axis distance units

The X-axis can be modified to display various distance units. These include: kilometers/km, meters/m, and centimeters/cm. To use these, set `x='km' / -x km`, `x='m' / -x m`, or `x='cm' / -x cm`.

See warning above for caveats about using distance units.

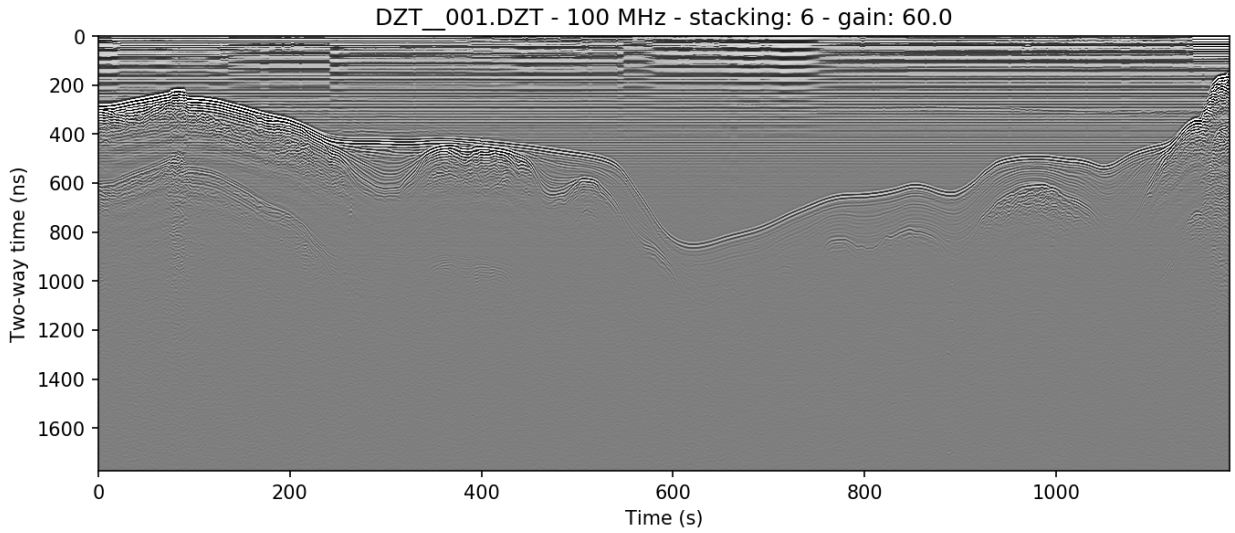
Back to top ↑

5.4 Reversing

Sometimes it is necessary to reverse the travel direction of a survey line in order to show a comparison with a line that travels in the opposite direction. readgssi will read arrays backwards if `reverse=True` or `-R` are set, using the *`readgssi.arrayops.flip()`* function.

```
readgssi.readgssi(infile='DZT__001.DZT', outfile='4a.png', frmt=None,
                  zero=[233], plot=5, stack='auto', gain=60,
                  reverse=True)
```

```
readgssi -i DZT__001.DZT -o 4a.png -Z 233 -p 5 -s auto -g 60 -R
```


DZT__001.DZT - 100 MHz - stacking: 6 - gain: 60.0
Two-way time (ns)
0
200
400
600
800
1000
1200
1400
1600
Time (s)
0
200
400
600
800
1000

Back to top ↑

CHAPTER

SIX

TRANSLATING TO DIFFERENT FORMATS

6.1 CSV

6.1.1 Basic CSV

Translation to csv is easy.

Python:

```
readgssi.readgssi(infile='DZT__001.DZT', outfile='DZT__001.csv', frmt='csv')
```

bash:

```
readgssi -i DZT__001.DZT -o DZT__001.csv -f csv
```

6.1.2 CSV of processed data

It's common to process data before outputting. Here, we distance-normalize and filter before writing to CSV.

Python:

```
readgssi.readgssi(infile='DZT__001.DZT', outfile='DZT__001.csv', frmt='csv',
                  normalize=True, freqmin=60, freqmax=100, bgr=0)
```

bash:

```
readgssi -i DZT__001.DZT -o DZT__001.csv -f csv -N -t 60-100 -r 0
```

6.2 numpy binary

The following python and bash commands do the same (process then output), but output to numpy binary format instead.

Python:

```
readgssi.readgssi(infile='DZT__001.DZT', outfile='DZT__001.csv', frmt='numpy',
                  normalize=True, freqmin=60, freqmax=100, bgr=0)
```

bash:

```
readgssi -i DZT__001.DZT -o DZT__001.csv -f numpy -N -t 60-100 -r 0
```

6.3 GPRPy-compatible format

And finally, these commands output the same data to a format compatible with GPRPy, which involves numpy binary (.npy) and a JSON serialization of header values.

Python:

```
readgssi.readgssi(infile='DZT__001.DZT', outfile='DZT__001.csv', frmt='gprpy',
                  normalize=True, freqmin=60, freqmax=100, bgr=0)
```

bash:

```
readgssi -i DZT__001.DZT -o DZT__001.csv -f gprpy -N -t 60-100 -r 0
```

Back to top ↑

CHAPTER

SEVEN

ADVANCED USAGE WITH BASH

UNIX users have a distinct advantage of being able to easily process entire folders full of DZTs with a simple command. Users who wish to do this should read up on how to construct for loops in Bash or simply follow and modify these examples below.

7.1 Processing all files in a folder

This command makes use of the `ls` function in Bash, which lists all files that match a specific pattern. In this case, we want the pattern to be "any DZT file," which ends up being simply `ls *.DZT` (the `*` symbol is a wildcard, meaning it matches any set of characters, so in this case it would match both `FILE_____005.DZT` and `test.DZT` but not `Test01.dzt` because the .DZT is case sensitive.).

```
for f in `ls *.DZT`; do readgssi -p 8 -n -r 0 -g 40 -Z 233 -z ns -N -x m -s auto -i
↪$f; done
```

The structure of this command is easy to understand if you know a little bit about `for` loops. This command loops over every file with the extension `.DZT` (`ls *.DZT` where `*` indicates a wildcard) and assigns the filename to the `f` variable on each loop. Then, after the semicolon, bash runs readgssi for every pass of the loop. In this case, the parameters are:

```
-p 8     # plot with size 8
-n       # suppress the matplotlib window; useful if you do not want the operation␣
↪interrupted
-r 0     # full-width background removal
-g 40    # gain of 40
-Z 233   # time zero at 233 samples
-z ns    # display the depth axis in nanoseconds
-N       # distance-normalize the profile
-x m     # display the x-axis in meters
-s auto  # apply automatic stacking
-i $f    # recall the `f` variable containing this loop's filename and feed it to the␣
↪input flag of readgssi
```

Finally, end the loop by closing the command with a linebreak `;`, and the `done` marker.

7.2 Processing specific subsets of files

You can make the command even more specific by further modifying the set of files returned by the `ls` command. For example:

```
for f in `ls FILE__{010..025}.DZT`; do readgssi -p 8 -n -r 0 -g 40 -Z 233 -z ns -N -x␣
↪m -s auto -i $f; done
```

This command will process only the 16 files in the numeric sequence between and including 010 and 025 in the set (`FILE__010.DZT`, `FILE__011.DZT`, ..., `FILE__025.DZT`). `bash` handles the zero padding for you as well. Pretty cool.

CHAPTER

EIGHT

TROUBLESHOOTING

Questions, feature requests, and bugs: please open a github issue. Kindly provide the error output, describe what you are attempting to do, and attach the DZT/DZX/DZG file(s) causing you trouble.

If you have a question that involves sensitive or proprietary data, send me an email confidentially at ian dot nesbitt at g mail dot com.

Thanks for reporting errors and helping to keep scientific software free and open-source!

8.1 Filtering errors

The filtering algorithm *readgssi.filtering.triangular()* will fail if you use scipy 1.2.x. Please upgrade scipy to 1.3.0 to avoid errors while filtering.

8.2 Antenna code errors

Of all the errors you are likely to encounter, these are the most numerous, easiest to fix, and hardest to predict. GSSI is liberal when it comes to naming antennas, so antenna codes, which are the primary identifying feature of the center frequency of the antenna, are numerous. This wouldn't be such an issue if GSSI had a list of them somewhere.

Alas, they don't, so I've had to try to compile my own, and it's incomplete. If you come across a KeyError in *readgssi.dzt.readdzt()* related to a variable called `ANT`, chances are your antenna needs to be added to the list. Copy and paste the error message into a new github issue and attach the DZT file to the message. I'll try to respond within 24 hours.

If you want to modify the code yourself, have a look at the `ANT` dictionary in `readgssi.config`. Use the key from the error message to create a new entry in the `ANT` dictionary that has both your key and the frequency of the antenna you're using. Once you've added the line with your antenna code and the frequency, reinstall and test your modified version of `readgssi` by *Installing from source*.

If your modified code is in a folder in your home directory, you should be able to reinstall using the command `pip install ~/readgssi`.

If you've tested it and it's working, please create a pull request with the changes, or open an issue and describe the changes you made.

The dictionary of antenna codes and center frequencies as of May 04, 2021 (version 0.0.19) is below.

```
ANT = {
    # 'code': integer center frequency
    '100MHz': 100,
    '200MHz': 200,
```

(continues on next page)

(continued from previous page)

```
    '270MHz': 270,
    '350MHz': 350,
    '400MHz': 400,
    '500MHz': 500,
    '800MHz': 800,
    '900MHz': 900,
    '1600MHz': 1600,
    '2000MHz': 2000,
    '2300MHz': 2300,
    '2600MHz': 2600,
    '3200': 'adjustable',
    '3200MLF': 'adjustable',
    'gprMa': 'adjustable',      # gprMax support
    'GSSI': 'adjustable',       # support for issue #11
    'CUSTOM': 'adjustable',
    '3207': 100,
    '3207AP': 100,
    '5106': 200,
    '5106A': 200,
    '50300': 300,
    '350': 350,
    '350HS': 350,
    'D400HS': 350,
    '50270': 270,
    '50270S': 270,
    'D50300': 300,
    '5103': 400,
    '5103A': 400,
    '50400': 400,
    '50400S': 400,
    '800': 800,
    'D50800': 800,
    '3101': 900,
    '3101A': 900,
    '51600': 1600,
    '51600S': 1600,
    'SS MINI': 1600,
    '62000': 2000,
    '62000-003': 2000,
    '62300': 2300,
    '62300XT': 2300,
    '52600': 2600,
    '52600S': 2600,
}
```

Back to top ↑

CHAPTER

NINE

CONTRIBUTING TO THIS PROJECT

9.1 Code contributions

Contributions to this project are always welcome. If you have questions or comments about how this software works, I want to hear from you. Even if coding isn't your thing, I want to make it easier for you to get involved.

There is no formal structure for contributions at the moment, but I will respond promptly to any pull request or issue on github (https://github.com/iannesbitt/readgssi). If and when you encounter bugs, kindly provide the error output, describe what you are attempting to do, and attach the DZT/DZX/DZG file(s) causing you trouble.

If you have a question that involves sensitive or proprietary data, send me an email confidentially at ian dot nesbitt at g mail dot com.

Thanks for helping to keep scientific software free and open-source!

9.2 Contributing in other ways

`readgssi` is open-source software written as a side project during the completion of my Master's degree at the University of Maine. It is written and maintained because it makes me feel warm and fuzzy inside, and because it helps me do my own work. It is not a money-making venture. It is intended for use by the wider scientific community as a way to understand field data faster, better, and with less hassle.

Keeping software free and open source is an important way to help scientists make science transparent and accessible to the wider community, and also to allow them to collaborate on better scientific outcomes and discoveries. Therefore I strongly urge code contributions over financial ones.

However, if this work has helped you or your lab personally and you would like to say thanks by buying me a coffee, please email me and/or seek me out at the next AGU/EGU, or drop me a couple bucks at https://paypal.me/IanNesbitt/. Thank you for your support, it is very much appreciated. Graduate student salaries aren't what they used to be!

Back to top ↑

CHAPTER

TEN

READGSSI.READGSSI (MAIN MODULE)

- genindex
- modindex
- search

CHAPTER

ELEVEN

READGSSI.DZT (READS DZTS)

readgssi.dzt.**header_info**(*header*, *data*)

Function to print relevant header data.

Parameters

- **header** (*dict*) – The header dictionary
- **data** (*numpy.ndarray*) – The data array

readgssi.dzt.**readdzt**(*infile, gps=Empty DataFrame Columns: [] Index: [], spm=None, start_scan=0, num_scans=-1, epsr=None, antfreq=[None, None, None, None], verbose=False, zero=[None, None, None, None]*)

Function to unpack and return things the program needs from the file header, and the data itself.

Parameters

- **infile** (*str*) – The DZT file location
- **gps** (*bool*) – Whether a GPS file exists. Defaults to False, but changed to pandas.DataFrame if a DZG file with the same name as infile exists.
- **spm** (*float*) – User value of samples per meter, if specified. Defaults to None.
- **epsr** (*float*) – User value of relative permittivity, if specified. Defaults to None.
- **zero** (*list[int,int,int,int]*) – List of time-zero values per channel. Defaults to a list of None values, which resolves to rh_zero.
- **verbose** (*bool*) – Verbose, defaults to False

Return type header (dict), radar array (numpy.ndarray), gps (False or pandas.DataFrame)

readgssi.dzt.**readtime**(*bytez*)

Function to read dates from rfDateByte binary objects in DZT headers.

DZT rfDateByte objects are 32 bits of binary (01001010111110011010011100101111), structured as little endian u5u6u5u5u4u7 where all numbers are base 2 unsigned int (uX) composed of X number of bits. Four bytes is an unnecessarily high level of compression for a single date object in a filetype that often contains tens or hundreds of megabytes of array information anyway.

So this function reads (seconds/2, min, hr, day, month, year-1980) then does seconds*2 and year+1980 and returns a datetime object.

For more information on rfDateByte, see page 55 of GSSI's SIR 3000 manual.

Parameters **bytes** (*bytes*) – The rfDateByte to be decoded

Return type datetime.datetime

- genindex
- modindex
- search

CHAPTER

TWELVE

READGSSI.ARRAYOPS (ARRAY MANIPULATION)

readgssi.arrayops.**distance_normalize**(*header*, *ar*, *gps*, *verbose=False*)

Distance normalization algorithm. Uses a GPS array to calculate expansion and contraction needed to convert from time-triggered to distance-normalized sampling interval. Then, the samples per meter is recalculated and inserted into the header for proper plotting.

Usage described in the *Distance normalization* section of the tutorial.

Parameters

- **header** (*dict*) – Input data array
- **ar** (*numpy.ndarray*) – Input data array
- **gps** (*pandas.DataFrame*) – GPS data from *readgssi.gps.readdzg()*. This is used to calculate the expansion and compression needed to normalize traces to distance.
- **verbose** (*bool*) – Verbose, defaults to False.

Return type header (dict), radar array (numpy.ndarray), gps (False or pandas.DataFrame)

readgssi.arrayops.**flip**(*ar*, *verbose=False*)

Read the array backwards. Used to reverse line direction. Usage covered in the *Reversing* tutorial section.

Parameters

- **ar** (*numpy.ndarray*) – Input data array
- **verbose** (*bool*) – Verbose, defaults to False

Return type radar array (numpy.ndarray)

readgssi.arrayops.**reducex**(*ar*, *header*, *by=1*, *chnum=1*, *number=1*, *verbose=False*)

Reduce the number of traces in the array by a number. Not the same as *stack()* since it doesn't sum adjacent traces, however *stack()* uses it to resize the array prior to stacking.

Used by *stack()* and *distance_normalize()* but not accessible from the command line or readgssi.readgssi().

Parameters

- **ar** (*numpy.ndarray*) – Input data array
- **by** (*int*) – Factor to reduce by. Default is 1.
- **chnum** (*int*) – Chunk number to display in console. Default is 1.
- **number** (*int*) – Number of chunks to display in console. Default is 1.
- **verbose** (*bool*) – Verbose, defaults to False.

Return type radar array (numpy.ndarray)

readgssi.arrayops.**stack**(*ar*, *header*, *stack='auto'*, *verbose=False*)

Stacking algorithm. Stacking is the process of summing adjacent traces in order to reduce noise — the thought being that random noise around zero will cancel out and data will either add or subtract, making it easier to discern.

It is also useful for displaying long lines on a computer screen. Usage is covered in the *Stacking* section of the tutorial.

`stack='auto'` results in an approximately 2.5:1 x:y axis ratio. `stack=3` sums three adjacent traces into a single trace across the width of the array.

Parameters

- **ar** (*numpy.ndarray*) – Input data array
- **by** (*int*) – Factor to stack by. Default is "auto".

Return type radar array (numpy.ndarray)

- genindex
- modindex
- search

CHAPTER

THIRTEEN

READGSSI.FILTERING

readgssi.filtering.**bgr**(*ar*, *header*, *win=0*, *verbose=False*)

Horizontal background removal (BGR). Subtracts off row averages for full-width or window-length slices. For usage see *Getting rid of horizontal noise*.

Parameters

- **ar** (*numpy.ndarray*) – The radar array
- **header** (*dict*) – The file header dictionary
- **win** (*int*) – The window length to process. 0 resolves to full-width, whereas positive integers dictate the window size in post-stack traces.

Return type numpy.ndarray

readgssi.filtering.**bp**(*ar*, *header*, *freqmin*, *freqmax*, *zerophase=True*, *verbose=False*)

Vertical butterworth bandpass. This filter is not as effective as *triangular()* and thus is not available through the command line interface or through readgssi.readgssi.readgssi().

Filter design and implementation are dictated by obspy.signal.filter.bandpass().

Parameters

- **ar** (*np.ndarray*) – The radar array
- **header** (*dict*) – The file header dictionary
- **freqmin** (*int*) – The lower corner of the bandpass
- **freqmax** (*int*) – The upper corner of the bandpass
- **zerophase** (*bool*) – Whether to run the filter forwards and backwards in order to counteract the phase shift
- **verbose** (*bool*) – Verbose, defaults to False

Return type numpy.ndarray

readgssi.filtering.**dewow**(*ar*, *verbose=False*)

Polynomial dewow filter. Written by fxsimon.

> **Warning:** This filter is still experimental.

Parameters

- **ar** (*numpy.ndarray*) – The radar array
- **verbose** (*bool*) – Verbose, default is False

Return type numpy.ndarray

readgssi.filtering.**triangular**(*ar*, *header*, *freqmin*, *freqmax*, *zerophase=True*, *verbose=False*)

Vertical triangular FIR bandpass. This filter is designed to closely emulate that of RADAN.

Filter design is implemented by scipy.signal.firwin() with numtaps=25 and implemented with scipy.signal.lfilter().

Note: This function is not compatible with scipy versions prior to 1.3.0.

Parameters

- **ar** (*np.ndarray*) – The radar array
- **header** (*dict*) – The file header dictionary
- **freqmin** (*int*) – The lower corner of the bandpass
- **freqmax** (*int*) – The upper corner of the bandpass
- **zerophase** (*bool*) – Whether to run the filter forwards and backwards in order to counteract the phase shift
- **verbose** (*bool*) – Verbose, defaults to False

Return type numpy.ndarray

- genindex
- modindex
- search

CHAPTER

FOURTEEN

READGSSI.FUNCTIONS (REUSABLES)

A number of helper functions used by readgssi.

readgssi.functions.**dzterror**(*e=''*)
Prints an error message then calls *gpx2dzg.functions.genericerror()* and passes *filetype='DZT'*.

e [str] The error message to print.

readgssi.functions.**dzxerror**(*e=''*)
Prints an error message then calls *gpx2dzg.functions.genericerror()* and passes *filetype='DZX'*.

e [str] The error message to print.

readgssi.functions.**genericerror**(*filetype='file'*)
Prints a standard message for a generic error using the *gpx2dzg.functions.printmsg()* function. This is called from functions in *gpx2dzg.io*.

filetype [str] The type of file this message is about. Used to format error string.

readgssi.functions.**naming**(*outfile=None*, *infile_basename=None*, *chans=[1]*, *chan=0*, *normalize=False*, *zero=None*, *stack=1*, *reverse=False*, *bgr=False*, *win=None*, *gain=None*, *dewow=None*, *freqmin=None*, *freqmax=None*, *plotting=None*, *zoom=None*, *absval=False*)
The Dr. Seth W. Campbell Honorary Naming Scheme

Descriptive naming, used to indicate the processing steps done on each file, if a specific output filename is not given. The theory behind this naming scheme is simple: it can be tough to remember how you made that plot!

Named for my graduate advisor, whom I love and cherish, who introduced me to this way of naming outputs.

```
naming scheme for exports:
  CHARACTERS    |      MEANING
    Ch0         |  Profile from channel 0 (can range from 0 - 3)
    Dn          |  Distance normalization
    Tz233       |  Time zero at 233 samples
    S8          |  Stacked 8 times
    Rv          |  Profile read in reverse (flipped horizontally)
    Bgr75       |  Background removal filter with window size of 75
    Dw          |  Dewow filter
    Bp70-130    |  triangular FIR filter applied from 70 to 130 MHz
    G30         |  30x contrast gain
    Abs         |  Color scale represents absolute value of vertical gradient
    Z10.20.7.5  |  zoom from 10-20 axis units on the x-axis and 5-7 on the z-axis
```

Parameters

- **outfile** (*str*) – The base output filename. If None, a new `outfile` will be generated from the input file basename. If it already exists, subsequent arguments will be appended. Defaults to None.
- **infile_basename** (*str*) – The input file basename (without file extension). Defaults to None.
- **chans** (*list[int,int,int,int]*) – A list of channels, which is converted to the number of channels using `len()`. Defaults to [1].
- **chan** (*int*) – The current channel number. Defaults to 0.
- **normalize** (*bool*) – Whether or not the array is distance-normalized. Defaults to False.
- **zero** (*int*) – The zero point for this particular channel. Defaults to None.
- **stack** (*int*) – The number of times the array was stacked. Defaults to 1.
- **reverse** (*bool*) – Whether or not the file was reversed. Defaults to False.
- **bgr** (*bool*) – Whether or not BGR was applied. Defaults to False.
- **win** (*int*) – The BGR window size if applicable. 0 is full-width BGR, greater than 0 is window size. Defaults to None.
- **gain** (*float*) – The gain value applied to the plot. Defaults to None.
- **dewow** (*bool*) – Whether or not dewow was applied. Defaults to None.
- **freqmin** (*int*) – The lower corner of the bandpass filter if applicable. Defaults to None.
- **freqmax** (*int*) – The upper corner of the bandpass filter if applicable. Defaults to None.
- **plotting** (*int*) – Stand-in for whether or not a plot was generated. The integer represents the plot height. Defaults to None.
- **zoom** (*list[int,int,int,int]*) – The zoom extents applied to the image. Defaults to None.
- **absval** (*bool*) – Whether or not the plot is displayed with absolute value of gradient. Defaults to False.

readgssi.functions.**printmsg**(*msg*)

Prints with date/timestamp.

Parameters **msg** (*str*) – Message to print

readgssi.functions.**zoom**(*zoom*, *extent*, *x*, *z*, *verbose=False*)

Logic to figure out how to set zoom extents. If specified limits are out of bounds, they are set back to boundary extents. If limits of a specified axis are equal, they are expanded to the full extent of that axis.

Parameters

- **zoom** (*list[int,int,int,int]*) – Zoom extents to set programmatically for matplotlib plots. Must pass a list of four integers: `[left, right, up, down]`. Since the z-axis begins at the top, the "up" value is actually the one that displays lower on the page. All four values are axis units, so if you are working in nanoseconds, 10 will set a limit 10 nanoseconds down. If your x-axis is in seconds, 6 will set a limit 6 seconds from the start of the survey. It may be helpful to display the matplotlib interactive window at full extents first, to determine appropriate extents to set for this parameter. If extents are set outside the boundaries of the image, they will be set back to the boundaries. If two extents on the same axis are the same, the program will default to plotting full extents for that axis.

- **extent** (*list[int,int,int,int]*) – Full extent boundaries of the image, in the style `[left, right, up, down]`.
- **x** (*str*) – X axis units
- **z** (*str*) – Z axis units
- **verbose** (*bool*) – Verbose, defaults to False

- genindex
- modindex
- search

CHAPTER

FIFTEEN

READGSSI.GPS (INGEST GPS INFO)

Reads GPS information from DZG files.

readgssi.gps.**msgparse**(*msg*)

Deprecated since version 0.0.12.

This function returns the NMEA message variables shared by both RMC and GGA.

Parameters **msg** (*pynmea2.nmea.NMEASentence*) – A pynmea2 sentence object.

Return type datetime.datetime, float, float

readgssi.gps.**pause_correct**(*header*, *dzg_file*, *threshold=0.25*, *verbose=False*)

This is a streamlined way of removing pauses from DZG files and re-assigning trace values. GSSI controllers have a bug in which GPS sentences are collected with increasing trace numbers even though radar trace collection is stopped. This results in a misalignment between GPS and radar traces of the same number. This function attempts to realign the trace numbering in the GPS file by identifying stops via a calculated velocity field.

Disclaimer: this function will identify and remove ALL pauses longer than 3 epochs and renumber the traces accordingly. Obviously this can have unintended consequences if the radar controller remains collecting data during these periods. Please be extremely cautious and only use this functionality on files you know have radar pauses that are accompanied by movement pauses. A backup of the original DZG file is made each time this function is run on a file, which means that if you make a mistake, you can simply copy the DZG backup (.DZG.bak) and overwrite the output (.DZG).

Any time you are working with original files, it is always good to have a "working" and "raw" copy of your data. Experimental functionality in readgssi cannot be held responsible for its actions in modifying data. You are responsible for keeping a raw backup of your data just in case.

A detailed explanation of each step taken by this function is available in the code comments.

Parameters

- **header** (*dict*) – File header produced by *readgssi.dzt.readdzt()*
- **dzg_file** (*str*) – DZG GPS file (the original .DZG, not the backup)
- **threshold** (*float*) – Numerical velocities threshold, under which will be considered a "pause"
- **verbose** (*bool*) – Verbose, defaults to False

Return type corrected, de-paused GPS data (pandas.DataFrame)

readgssi.gps.**readdzg**(*fi*, *frmt*, *header*, *verbose=False*)

A parser to extract gps data from DZG file format. DZG contains raw NMEA sentences, which should include at least RMC and GGA.

NMEA RMC sentence string format: `$xxRMC,UTC hhmmss,status,lat DDmm.sss,lon DDDmm.sss,SOG,COG,date ddmmyy,checksum *xx`

NMEA GGA sentence string format: `$xxGGA,UTC hhmmss.s,lat DDmm.sss,lon DDDmm.sss, fix qual,numsats,hdop,mamsl,wgs84 geoid ht,fix age,dgps sta.,checksum *xx`

Shared message variables between GGA and RMC: timestamp, latitude, and longitude

RMC contains a datestamp which makes it preferable, but this parser will read either.

Parameters

- **fi** (*str*) – File containing gps information
- **frmt** (*str*) – GPS information format ('dzg' = DZG file containing gps sentence strings (see below); 'csv' = comma separated file with: lat,lon,elev,time)
- **header** (*dict*) – File header produced by *readgssi.dzt.readdzt()*
- **verbose** (*bool*) – Verbose, defaults to False

Return type

GPS data (pandas.DataFrame)

The dataframe contains the following fields: * datetimeutc (`datetime.datetime`) * trace (`int` trace number) * longitude (`float`) * latitude (`float`) * altitude (`float`) * velocity (`float`) * sec_elapsed (`float`) * meters (`float` meters traveled)

- genindex
- modindex
- search

CHAPTER

SIXTEEN

READGSSI.PLOT

- genindex
- modindex
- search

CHAPTER

SEVENTEEN

READGSSI.TRANSLATE (OUTPUTS)

readgssi.translate.**csv**(*ar*, *outfile_abspath*, *header=None*, *verbose=False*)

Output to csv. Data is read into a pandas.DataFrame, then written using pandas.DataFrame.to_csv().

Parameters

- **ar** (*numpy.ndarray*) – Radar array
- **outfile_abspath** (*str*) – Output file path
- **header** (*dict*) – File header dictionary to write, if desired. Defaults to None.
- **verbose** (*bool*) – Verbose, defaults to False

readgssi.translate.**dzt**(*ar*, *outfile_abspath*, *header*, *verbose=False*)

Warning: DZT output is only currently compatible with single-channel files.

This function will output a RADAN-compatible DZT file after processing. This is useful to circumvent RADAN's distance-normalization bug when the desired outcome is array migration.

Users can set DZT output via the command line by setting the `-f dzt` flag, or in Python by doing the following:

```
from readgssi.dzt import readdzt
from readgssi import translate
from readgssi.arrayops import stack, distance_normalize

# first, read a data file
header, data, gps = readdzt('FILE__001.DZT')

# do some stuff
# (distance normalization must be done before stacking)
for a in data:
    header, data[a], gps = distance_normalize(header=header, ar=data[a], gps=gps)
    header, data[a], stack = stack(header=header, ar=data[a], stack=10)

# output as modified DZT
translate.dzt(ar=data, outfile_abspath='FILE__001-DnS10.DZT', header=header)
```

This will output `FILE__001-DnS10.DZT` as a distance-normalized DZT.

Parameters

- **ar** (*numpy.ndarray*) – Radar array

- **infile_basename** (*str*) – Input file basename
- **outfile_abspath** (*str*) – Output file path
- **header** (*dict*) – File header dictionary to write, if desired. Defaults to None.
- **verbose** (*bool*) – Verbose, defaults to False

readgssi.translate.**gprpy**(*ar*, *header*, *outfile_abspath*, *verbose=False*)

Save in a format GPRPy can open (numpy binary .npy and a .json formatted header file).

Note: GPRPy support for this feature is forthcoming (https://github.com/NSGeophysics/GPRPy/issues/3#issuecomment-460462612).

Parameters

- **ar** (*numpy.ndarray*) – Radar array
- **outfile_abspath** (*str*) – Output file path
- **header** (*dict*) – File header dictionary to write, if desired. Defaults to None.
- **verbose** (*bool*) – Verbose, defaults to False

readgssi.translate.**h5**(*ar*, *infile_basename*, *outfile_abspath*, *header*, *verbose=False*)

Warning: HDF5 output is not yet available.

In the future, this function will output to HDF5 format.

Parameters

- **ar** (*numpy.ndarray*) – Radar array
- **infile_basename** (*str*) – Input file basename
- **outfile_abspath** (*str*) – Output file path
- **header** (*dict*) – File header dictionary to write, if desired. Defaults to None.
- **verbose** (*bool*) – Verbose, defaults to False

readgssi.translate.**json_header**(*header*, *outfile_abspath*, *verbose=False*)

Save header values as a .json so another script can take what it needs. This is used to export to GPRPy.

Parameters

- **header** (*dict*) – The file header dictionary
- **outfile_abspath** (*str*) – Output file path
- **verbose** (*bool*) – Verbose, defaults to False

readgssi.translate.**numpy**(*ar*, *outfile_abspath*, *header=None*, *verbose=False*)

Output to binary numpy binary file (.npy) with the option of writing the header to .json as well.

Parameters

- **ar** (*numpy.ndarray*) – Radar array
- **outfile_abspath** (*str*) – Output file path

- **header** (*dict*) – File header dictionary to write, if desired. Defaults to None.
- **verbose** (*bool*) – Verbose, defaults to False

readgssi.translate.**segy**(*ar*, *outfile_abspath*, *header*, *verbose=False*)

> **Warning:** SEGY output is not yet available.

In the future, this function will output to SEGY format.

Parameters

- **ar** (*numpy.ndarray*) – Radar array
- **outfile_abspath** (*str*) – Output file path
- **header** (*dict*) – File header dictionary to write, if desired. Defaults to None.
- **verbose** (*bool*) – Verbose, defaults to False

readgssi.translate.**writetime**(*d*)

Function to write dates to `rfDateByte` binary objects in DZT headers. An inverse of the *readgssi.dzt.readtime()* function.

DZT `rfDateByte` objects are 32 bits of binary (01001010111110011010011100101111), structured as little endian u5u6u5u5u4u7 where all numbers are base 2 unsigned int (uX) composed of X number of bits. Four bytes is an unnecessarily high level of compression for a single date object in a filetype that often contains tens or hundreds of megabytes of array information anyway.

So this function reads a datetime object and outputs (seconds/2, min, hr, day, month, year-1980).

For more information on `rfDateByte`, see page 55 of GSSI's SIR 3000 manual.

Parameters **d** (*datetime*) – the `datetime.datetime` to be encoded

Return type bytes

- genindex
- modindex
- search

CHAPTER

EIGHTEEN

READGSSI.CONSTANTS (ESSENTIALS)

This module contains a number of variables that readgssi needs to perform physics calculations and interpret radar information from DZT files.

18.1 Physical constants

`c = 299792458` - celerity of electromagnetic waves in a vacuum

`Eps_0 = 8.8541878 * 10**(-12)` - epsilon naught, the vacuum permittivity

`Mu_0 = 1.257 * 10**(-6)` - mu naught, the vacuum permeability

18.2 GSSI constants

`MINHEADSIZE = 1024` - minimum DZT file header size in bytes

`PAREASIZE = 128` - DZT file fixed info area size in bytes

18.3 Dictionaries

`UNIT` - dictionary of GSSI field units and associated unit codes

`ANT` - dictionary of GSSI antennas and associated antenna codes (read more about these in *Antenna code errors*)

- genindex
- modindex
- search

CHAPTER

NINETEEN

READGSSI.CONFIG (ESSENTIALS)

This module contains some things readgssi needs to operate, both command line and python-related. It contains the distribution name, author, and help text.

- genindex
- modindex
- search

CHAPTER

TWENTY

INDICES AND TABLES

- genindex
- modindex
- search

Back to top ↑

PYTHON MODULE INDEX

r

INDEX