ODAS MATLAB Library
Technical Manual
Version 4.02

2016-05-25

William Douglas and Rolf Lueck

Rockland Scientific International Inc.
520 Dupplin Rd
Victoria, BC, CANADA, V8Z 1C1
www.rocklandscientific.com

Contents

List of Figures

List of Tables

Listings

1 History

- 2005-02-01 (RGL) Original release, V1.
- 2009-04-08 (RGL) v2, major improvement to User Manual.
- 2009-07-23 (RGL) Added pitch and roll sensors.
- 2011-09-15 (RGL/AWS) v3, major changes related to ODAS header v6
- 2012-12-04 (RGL/WD) v3.1, additional functions and changes to setupfilestr
- 2013-02-26 (WD) v3.1.1, minor bug fixes in library and documentation
- 2014-00-00 (WM/RGL) v3.2, numerous undocumented improvements.
- 2015-12-17 (WD/RGL), Version 4.0. comprehensive conversion into physical units that applies to all RSI instruments, using `odas_p2mat`. `quick_look` that functions with all vertical profilers and MicroRiders mounted on Slocum gliders and Sea-gliders.
- 2016-05-25 (WD/RGL), Version 4.02. Bugfixes and various other small improvements.

2 Introduction

The ODAS Library of Matlab functions provides the tools to process data collected with all instruments made by Rockland Scientific International (RSI). All functions are based on Matlab and the user must have a valid license for Matlab in order to use the functions described here. Some functions, such as `plot_spec`, which is used to view the spectra within the structure returned by `quick_look`, work only with Matlab versions 2014b and later versions.

The flow of information, from raw data to products that are useful for scientific work, is depicted in Figure 2.0.1. An instrument produces raw data files according to user-supplied information in a configuration-file (which, typically, has the name `setup.cfg`) using data acquisition software that is supplied with your instrument. The result is an ODAS data file with a base-name that is determined by an entry in the configuration-file. Three digits are automatically appended to the base-name and the name is completed with the extension ".`p`". The functions in the Matlab Library convert this raw binary file into a Matlab mat-file that can be used to convert the data into physical units and to conduct other processing of your data. The outputs from the functions in the ODAS Matlab Library, in the form of figures and arrays of information, can be used for scientific analysis and other purposes.

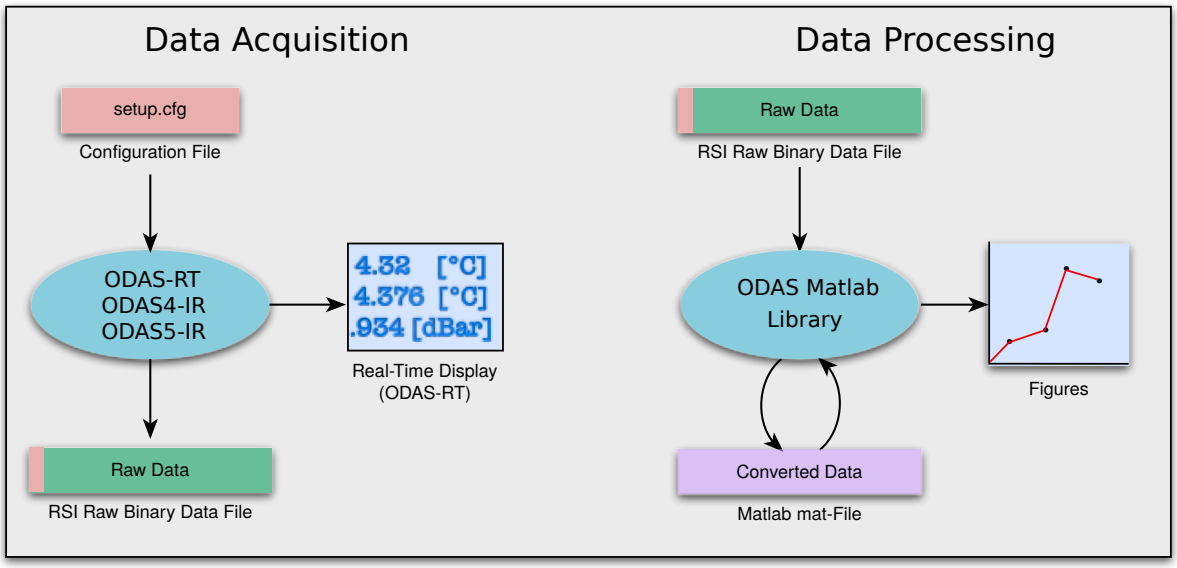

Figure 2.0.1: Data Acquisition and Post Processing Overview

This document is a "technical guide" to the ODAS Matlab Library. It is not a tutorial document that teaches a user how to process their data.

2.1 Typical usage of the ODAS Matlab Library

The ODAS Matlab Library was originally conceived to give users a set of functions for writing their own data processing software. Advanced users may wish to continue to use the library for building blocks for their own software. Most users may want to use the function `quick_look` (and its companion function `odas_p2mat`) to process their data.

The function `quick_look` works with all vertical profilers and for data collected with a MicroRider mounted on a Slocum glider and a Sea-glider. It does not (yet) work for data collected with horizontal profilers, such as an AUV and the Nemo float. This function:

- converts your data into physical units and places it into a mat-file, using the function `odas_p2mat`,
- plots numerous figures for data visualization, and
- creates a Matlab structure containing the rate of dissipation, ϵ, shear and scalar spectra, and many other products.

The function `odas_p2mat` will convert data into physical units that was collected with *all* platforms and vehicles.

A comprehensive description of `quick_look` and `odas_p2mat` is beyond the scope of this document and is provided in RSI Technical Note 039. Here, we provide only a brief description and a few examples in sections 5.5 and 5.6. The successful acquisition of data and its conversion to physical units, and the estimation of the rate of dissipation, ϵ, are predicated on a proper configuration file for your instrument. This file is usually named `setup.cfg` and is described in sections 3 and 4. A data file can be converted into physical units using one of two methods.

- By calling `odas_p2mat` directly. This will create a mat-file with the same name as the raw data file.
- Or, by calling `quick_look` (with proper input parameters) because it will call `odas_p2mat`, before making its plots and estimating the rate of dissipation from the shear probe data.

The behaviour of `odas_p2mat` and `quick_look` is controlled by a 'structure'[1] in their input arguments. This structure contains 'fields' and the values of these fields determine how these function process data. If no structure is passed to these functions, then they use a default structure containing fields with default values. The names of the fields used by these two functions are unique and so, a single structure can be used with both functions. Calling either function without any input arguments returns their default structure. Calling the function `odas_p2mat` without input arguments returns the default structure for converting data into physical units. Calling `quick_look` without any input arguments, returns a structure with the default values for the fields used by *both* `odas_p2mat` (for conversion inot physical units) and by `quick_look` (for data visualization). You can capture the default structure and customize the values of its fields to suit your processing requirements. You can save it in the local directory for later use.

[1] See the Matlab documentation for an explanation of a structure.

The function `quick_look` calls `odas_p2mat` only if a mat-file does not exist, or if the existing mat-file was converted into physical units using fields that differ from the ones in the structure in the input arguments. You will be queried for permission to erase the mat-file.

These two, and all other, functions in the Matlab ODAS Library are detailed in section 6.

2.2 Data file structure

Data files produced by RSI instruments have the extension ".p" and are raw binary files. Prior to 2010, RSI instruments produced version-1 data files. Since 2010, RSI instruments produce version-6 files. The Matlab library is tailored to, and tested with, version-6 files. It will *not* work with version-1 files. However, version-1 files can be upgraded to version-6 files, using the function `patch_setupstr` (see section 6).

The internal structure of a data file is illustrated in Figure 2.2.1. More detail is provided in Appendix A. The inclusion of the configuration file into the data file allows you to automate nearly all aspects of data processing. The conversion of raw data files into mat-files, deconvolution, and the conversion into physical units can now be done without any reference to auxiliary information, if the configuration file is populated with the correct information. Section 3 provides a detailed explanation of the configuration file.

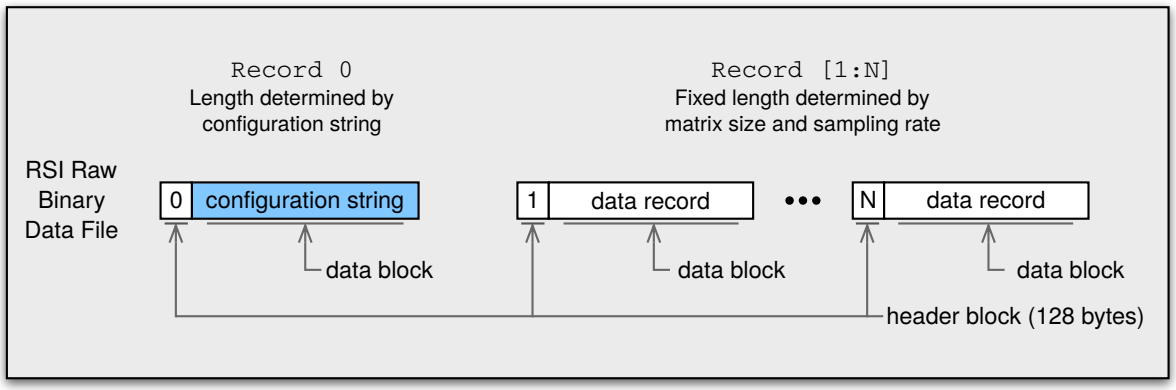

Figure 2.2.1: Data-file format. The configuration string is a copy of the configuration file used during data acquisition.

3 Configuration Files

A configuration file serves three purposes. First, it contains all settings required to configure the RSI data acquisition software for data collection. Upon start-up, the data acquisition software will read the configuration file and acquire data according to the contents of this file.

Second, the configuration file is used to store information required to convert the recorded raw data into physical units. This allows an RSI instrument to, for example, report the current depth based on the pressure transducer. In addition, after the data file is collected, the task of data processing is greatly simplified because the information required to convert the raw data into physical units is embedded in the data file. When a raw data file is converted into a mat-file, the configuration file that was used for the data acquisition is saved into the string variable `setupfilestr`.

Third, the configuration file can be used to hold user provided information which is available when the data file is processed. While not required by the instrument nor required for conversion into physical units, user provided data can greatly assist in organizing the collected data files and even in automating the process of data analysis. For example, the location of a profile can be added to a configuration file. When processing the resulting data files, reference to the location will be available and can be automatically extracted by your Matlab scripts.

The contents of a configuration string are accessed with the `setupstr` function. This function scans a configuration string for structured information and returns values that match a provided query. You can modify the information in the configuration string using the `extract_setupstr` and `patch_setupstr` functions, to correct some errors present at the time of data acquisition. Examples are provided in section 3.3.

3.1 Anatomy of a Configuration File

Configuration files are standard ASCII[2] text files consisting of a basic structure composed of `parameters`, `sections`, and `comments`. Section 3.4 shows an example of a configuration file.

Parameters

Information within a configuration file is stored as `parameters`. Parameters consists of two parts delimited by an equal sign - a `name` and a `value`.

```
name=value
```

[2]Configuration files can only contain 7-bit characters - values from 0x00 to 0x7F.

Each line can have a maximum of one parameter. An equal sign separates the name of a parameter from its value. Extra white-space surrounding the name or value is ignored. Thus, "`name=value`" is equivalent to " `name = value` ".

Sections

Groups of parameters are identified by `sections`. A section declaration must be on its own line and consists of the `name` string enclosed by square brackets (`[ ]`).

```
[section]
```

The declaration of a section automatically ends the previous section. Sections are referenced by their *section identifier*, a value constructed in one of two ways. First, sections are scanned for a *parameter* with the name "`name`". If present, this parameter value is used as the section identifier. If not found, the *section* `name` is used as the section identifier.

To ensure sections can be uniquely referenced, each occurrence of a parameter named `name` must be unique. If a section does not contain a `name` parameter, then the name of that section must be unique, because it will be used as the section identifier.

Parameters declared before the first section declaration are automatically assigned to the root section, `[root]`. Parameters in the root section are referenced using the section name `root`, even if the root section is not explicitly declared.

Comments

All characters after a semicolon (`;`) are considered *comments* and are ignored. Comments are still valuable and can assist people in understanding the meaning and value of a parameter.

```
; A generic example showing comments.
[section]     ; Description of section.
date = value  ; This is the date of creation.
```

3.2 Adding Values

Information can be added to a configuration file but the resulting file must adhere to the format described in section 3.1. In addition, RSI instruments expect certain sections and parameters to exist so care should be taken when modifying a configuration file. A configuration file that has been modified should be tested by using it for data acquisition *before* a real deployment.

If you want to add new sections to your configuration file, make sure that it has a unique name and never use a parameter with the name "`id`", because that is used in conjunction with

the [matrix] section to identify a channel and to demultiplex your data (unless, of course, that channel exists in your instrument and it is identified in the [matrix] section).

Edit existing [channel] sections to change the coefficients for conversion into physical units, particularly after installing a new sensor. You must not added a section that uses the parameter name and assign it a value that is used by RSI (see Table 3.7.2 for a partial list of names used by RSI). In general, avoid using the parameter name by simply calling it something else, such as ship_name=, for example.

The following example depicts a portion of a configuration file that has been modified by a user to support additional parameters.

Listing 1: Partial configuration file example

```
[matrix]
num_rows = 2
row01    = 255 0 1 2 3
row02    =   0 0 1 2 3

[cruise] ; User provided section with unique name
boat     = Titanic
operator = The Unlucky Captain
date     = April 14, 1912

[channel]
id         = 8
name       = sh1
type       = shear
diff_gain  = 1.045
sens       = 0.0638
adc_fs     = 4.096
adc_bits   = 16
SN         = M358  ; User provided probe SN
```

In this example, the user has created a section with the name "cruise". The name cruise is acceptable because it does not conflict with existing section names or name parameters. Inside this section, the parameters "boat", "operator", and "date" have been added.

A parameter has also been added to an existing section. The section with name channel has been appended with parameter "SN = M358". This is valid because the parameter name "SN" will not conflict with any RSI defined channel parameters. See table 3.8.1 for a list of RSI defined channel parameters that should be avoided.

Multiple sections named channel can exist within a configuration file. These sections must contain the unique parameter "name=channel_name" so that the section can be referenced. The name parameter acts as the section identifier.

You find the value of the parameter named "SN" in the above example, by making the following query:

```
>> probeSN = setupstr( setupfilestr, 'sh1', 'sn' )
```

```
probeSN =
    'M358'
```

Notice the value "sh1" is used to identify the section. Parameters with name "name" are unique because they are used as the section identifier.

The following query differs because the section name [cruise] is used as the section identifier:

```
>> boat = setupstr( setupfilestr, 'cruise', 'boat' )
boat =
    'Titanic'
```

A full explanation of the setupstr function can be found in section 6.

3.3 Extracting Values

The setupstr function is used to query a configuration string for parameter values. Different results are returned depending on how it is called.

Syntax

```
>> object = setupstr( config_string )
>> sections = setupstr( config_string, 'section' )
>> value = setupstr( config_string, 'section', ...
                                        'parameter_name' )
>> [S P V] = setupstr( config_string, 'section', ...
                                          'parameter_name', ...
                                          'parameter_value' )
```

Argument	Description
config_string	the configuration string.
section	the desired section descriptor.
parameter_name	the desired parameter name.
parameter_value	the desired parameter value.
object	structure index containing the contents of the configuration file. Use in place of config_string to (optionally) speed up subsequent queries.
S	cell array of matching section descriptors.
P	cell array of matching parameter names.
V	cell array of matching parameter values.

Note:

1. The section and parameter inputs are interpreted as modified regular expressions. Users do not have to understand regular expressions because queries using ASCII characters and numbers result in direct matches.
2. When section and parameter inputs are given as empty strings, (''), the `setupstr` function matches all values.
3. All returned values are formatted as cell arrays.

Finding Sections

When called with two string arguments, the `setupstr` function will return a cell array of section identifiers that match the requested section.

```
>> sections = setupstr( config_string, 'section' )
```

The requested section, `'section'`, is matched against each section identifier found in the configuration string. The returned value, "`sections`", is a cell array of matching section identifiers.

This is commonly used to test if a section exists. The function is called with `section` set to a requested section identifier and if an empty cell array is returned, there are no matches. The function can also be used to find sections. Calling the function with an empty string "`''`" for `'section'` returns a cell array containing all section identifiers.

Finding Parameter Values

When called with three string arguments, the `setupstr` function will return a cell array of parameter values that match the requested section and parameter name.

```
>> value = setupstr( config_string, 'section', ...
                                    'parameter_name' )
```

The function returns a cell array containing all parameter values where the parameter name matches `'parameter_name'` and the section identifier matches `'section'`.

Advanced Search

Calling the `setupstr` function with four arguments returns a tuple of three cell arrays. Each matching parameter has their section identifier in "`S`", parameter name in "`P`", and parameter value in "`V`".

```
>> [S P V] = setupstr( config_string, 'section', ...
                                      'parameter_name', ...
                                      'parameter_value' )
```

When queried using regular expressions, the function is useful for performing searches of a configuration string. The following command will search all sections for a parameter named "`id`" set to a valid integer within the configuration string shown in listing 1.

```
>> [S P V] = setupstr( config_string, '', 'id', '[0-9]+' )
S =
    'sh1'
P =
    'id'
V =
    '8'
```

The returned tuple contains values for each matching parameter. In this example, the regular expression "`[0-9]+`" matches all positive integers.

Faster Searches

It is possible to save the configuration string as an indexed structure to be used in future function calls. This indexed structure is used in place of the configuration string while all other arguments stay the same. The time required to perform the first function call does not change but subsequent function calls will be much faster.

```
>> obj = setupstr( config_string );
>> boat = setupstr( obj, 'cruise', 'boat' )
boat =
    'Titanic'
>> user = setupstr( obj, 'cruise', 'operator' );
user =
    'The Unlucky Captain'
```

3.3.1 `setupstr` Examples

Extracting a parameter value

Configuration File Segment:

```
[Cruise_Info]
cruise_name=Gulf of Mexico
```

Matlab Code:

```
>> cruiseName = setupstr( setupfilestr, 'cruise_info', ...
                                        'cruise_name' )
cruiseName =
    'Gulf of Mexico'
>> disp( char(cruiseName) )
```

```
Gulf of Mexico
```

Find the parameter value with the name `cruise_name` located within the section `Cruise_Info`. Display the resulting value on the console. The "`char()`" function is used to convert the cell into a string. The arguments are case in-sensitive.

Find the serial number of a shear probe

Configuration File Segment:

```
[channel]
id        = 8
name      = sh1
type      = shear
diff_gain = 1.045
sens      = 0.0638
adc_fs    = 4.096
adc_bits  = 16
SN        = M358  ; User provided probe SN
```

Matlab Code:

```
>> obj = setupstr( setupfilestr );
>> serialNumber = setupstr( obj, 'sh1', 'sn' )
serialNumber =
    'M358'
```

Read the serial number of the shear probe used for the `sh1` channel. This is an optional parameter. The function only work if the parameter is in the configuration file.

The query was made using two function calls. The first call generates an index to speed subsequent searches and the second call queries this index.

Plotting profile locations on a map

This example assumes multiple profiles have been acquired and the resulting data files have been converted into `.mat` files. The `.mat` files are located in the current directory and each has a variable called `setupfilestr` containing the configuration string.

Configuration File Segment:

```
[profile]
date = 2012/01/01
GPSx = -123.376242
GPSy = 48.44792
```

Matlab Code:

```
>> locationX = [];
>> locationY = [];
>> locationDate = {};
>> offset = 2;             % Point names are offset from points
>>                         % so they do not overlap.
>> files = dir( '*.mat' );
>> for file = files',
>>   load( file.name, 'setupfilestr' );
>>   x = setupstr( setupfilestr, 'profile', 'gpsx' );
>>   y = setupstr( setupfilestr, 'profile', 'gpsy' );
>>   t = setupstr( setupfilestr, 'profile', 'date' );
>>   locationX(end+1) = str2double( char(x) );
>>   locationY(end+1) = str2double( char(y) );
>>   locationDate{end+1} = char( t );
>> end
>> scatter( locationX, locationY )
>> text( locationX+offset, locationY+offset, locationDate )
```

This code results in a scatter plot of profile locations labelled with the profile date. So long as the operator sets the GPS coordinates within the configuration file before each profile, a map of profile locations can be generated directly from the profile data.

3.4 Example Configuration File

Listing 2: Example Configuration File

```
; [root]  This is a comment and is ignored by RSI instruments
; and software. Including comments can be helpful for users -
; this one indicates the start of the root section.
rate    = 512
disk    = c:/data
prefix  = dat_
recsize = 1
no-fast = 7
no-slow = 2

[matrix]
num_rows = 4
row01 = 255  0  1  2  3  5  7  8  9
row02 =   4  6  1  2  3  5  7  8  9
row03 =  10 11  1  2  3  5  7  8  9
row04 =  32  0  1  2  3  5  7  8  9

[instrument_info]
vehicle = vmp

[cruise_info]
date   = 2011-01-15
cid    = BT201101
cname  = Bermuda Triangle
vessel = HMS Pinafore

[channel]
id       = 10
name     = P
type     = poly
coef0    = 3.67
coef1    = 0.062076
coef2    = 4.724e-8

[channel]
id       = 32
name     = V_Bat
type     = voltage
G        = 0.1
adc_fs   = 4.096
adc_bits = 16

; Additional channels definitions required but not shown.
```

3.5 [root] Section

The root section is usually not declared explicitly. Instead, it is implied. It consists of all parameters before the first section declaration. In listing 2, the root section starts with parameter "rate" and continues to section "[matrix]". For this example, the start of the root section has been identified with a comment.

The root section contains parameters used to configure RSI instruments and the data acquisition software. The required parameters are listed in table 3.5.1;

Table 3.5.1: Required parameters for section [root].

Parameter	Description
rate	sampling rate of the channel matrix in rows per second – usually 512. This is also the sampling rate for each fast channel (samples/second) because each fast channel is in every row. See section 3.7.
disk	full path to the directory where the data files are to be written. Use forward slashes ("/") to delimit directories regardless of operating system. For internally recording instruments, this is usually /data. For real-time instruments, this parameter can be blank (empty) in order to default to the local directory.
prefix	base name for the generated data files. A three digit number is automatically appended to the base name every time data acquisition is started. Internally recording instruments are limited to a total of 8 characters. The extension ".p" is automatically appended to the file name.
recsize	requested size of a data record in seconds. The value is almost always 1. The actual size may differ slightly because records must contain the samples from a whole number multiple of the address matrix.
no-fast	number of fast columns in the address matrix (see details below). It can be any value ≥ 0.
no-slow	number of slow columns in the address matrix. It can be any value ≥ 0.
man_com_rate	(ODAS-RT) communication speed parameter for real-time instruments. The rate, in Mbit/s, is set to the inverse of the provided integer value. A value of 1 equals a rate of 1 Mbit/s and a value of 5 equals a rate of 0.2 Mbit/s. The jumpers must be set on the RSTRANS board to match this value. This parameter is not need for internally recording instruments.

Optional parameters for the [root] section are listed in table 3.5.2.

Table 3.5.2: Optional parameters for section `[root]`.

Parameter	Description
`max_pressure`	maximum pressure (dbar) before the weight release is activated. It is used on instruments that utilize a weight release mechanism.
`max_time`	maximum duration of data acquisition, in seconds, before the weight release is activated. This provides a redundant condition for releasing a weight.
`brate`	(ODAS-RT) speed (baud-rate) of the serial port connected to an auxiliary device. Only required when ODAS-RT must support such devices.
`reclen`	(ODAS-RT) record length (bytes) of an auxiliary serial device. Only required when ODAS-RT must support such devices.
`command`	(ODAS-RT) optional command to send to the serial device.
`stop_after_release`	(ODAS-IR) integer indicating the number of seconds that data acquisition will continue after the release mechanism is triggered. The default value of 0 implies that the acquisition will not stop.

3.6 [instrument_info] Section

The instrument information section has one required parameter (vehicle), but it should be used to provide additional information, such as instrument model, serial number, etc. The conversion of data into physical units, the method of estimating the speed of profiling, and data visualization are vehicle-dependent.

Table 3.6.1: Parameters for section [instrument_info].

Parameter	Description
vehicle	Mandatory. The vehicle of your instrument. Recognized values are vmp, rvmp, slocum_glider, sea_glider, auv, stand, xmp, and nemo. The rvmp is an uprising VMP.
model	Optional. The model of your instrument. For example, vmp-250, or mr_1000. This is currently used only for trouble shooting.
sn	Optional. The serial number of your instrument. Used for trouble shooting.

3.7 [matrix] Section

The [matrix] section defines the channel address matrix – the order in which data channels will be sampled. Data channels correspond to sensors so this matrix can be used to identify which sensors are sampled and, when combined with the "rate" parameter in [root] section, the frequency at which they are sampled.

A minimum of two parameters are required in the [matrix] section. One provides the number of rows and the other(s) specifies the channels in a row (see Table 3.7.1).

Parameter	Description
num_rows	height of the matrix, as measured by the number of rows.
row01	the first row consisting of a list of channel numbers separated by white-space. Each channel number is an integer between 0 and 255 and is called a channel id. White-space consists of either spaces or tab characters.
row{XX}	additional rows require a parameter with the name "row" plus the row number where rows are counted from the top starting at 1. When the row number consists of a single digit, the number is prepended with a 0. Each additional row is of the same type as parameter "row01".

Table 3.7.1: Parameters for section [matrix]

The data acquisition software samples one row at a time, returning to the first row after the completion of the last row. The sampling frequency is determined by the "rate" parameter – set to 512 in listing 2. In this example, 512 rows are sampled every second. Because there are 8 rows in this example, the entire matrix will be sampled 64 times per second.

The sampler traverses each row, from left to right, sampling the channels in sequence. When traversing the first row in listing 2, the channels 255, 0, 1, 2, 3, 5, 7, 8, and 9 will be sampled in the provided order. The duration between samples, τ, is uniform and given by

$$\tau = 1/(RC) \tag{1}$$

where R is the rate and C is the number of columns in the address matrix. The inverse of τ is the aggregate sampling rate and its exact[3] value (to $\pm 1 \times 10^{-6}$) is provided in the record header (see Table A.1.1 of the appendix).

Fast channels are defined as those channels that appear in the same column within each row of a channel matrix. From listing 2, channels 1, 2, 3, 5, 7, 8, and 9 are considered fast. Because they exist in each row, fast channels will be sampled at the same rate as rows – defined by the "rate" parameter and equal to 512 in this example. The root parameter "no-fast" specifies the number of fast channels within a single row – 7 in this example.

[3] The timing is derived from the integer division of a 24×10^6 Hz ± 1 PPM clock and cannot always give the exact value requested by the rate parameter.

Channels that are not fast are called slow – they are not in each row of the channel matrix. They are sampled at a lower rate, R_s, given by

$$R_s = \frac{R}{N_r} N \tag{2}$$

where N_r is the number of rows in the address matrix and N is the number of occurrences of a channel in the address matrix. For listing 2, channel 10 occurs once within the matrix resulting in a sampling rate of $R_s = 64\,\mathrm{s}^{-1}$. Channel 4 occurs twice and will have a sampling rate of $128\,\mathrm{s}^{-1}$. The root parameter "`no-slow`" specifies the number of slow channels within a single row (or the number of columns of slow channels). For this example, the value is 2.

By convention, slow channels must always be to the left of fast channels within the channel matrix.

The process of conversion to physical units includes generating the time vectors `t_fast` and `t_slow` corresponding to the samples in fast and slow channels. If a slow channel has more than one entry in the address matrix, then the users must generate their own time vector for such a channel.

Some of the common channel `id` values, and the signals to which they are associated, are listed in table 3.7.2. The configuration file shipped with your instrument will identify every working channel.

ID	Signal	Typical Name
0	ADC ground signal.	Gnd
1	Accelerometer x-axis.	Ax
2	Accelerometer y-axis.	Ay
3	Accelerometer z-axis.	Az
4	Thermistor probe #1 without pre-emphasis.	T1
5	Thermistor probe #1 with pre-emphasis.	T1_dT1
6	Thermistor probe #2 without pre-emphasis.	T2
7	Thermistor probe #2 with pre-emphasis.	T2_dT2
8	Shear probe #1.	Sh1
9	Shear probe #2.	Sh2
10	Pressure transducer without pre-emphasis.	P
11	Pressure transducer with pre-emphasis.	P_dP
12	Voltage across the pressure transducer. Sometimes used for micro-conductivity sensor #1 with pre-emphasis.	PV, C1_dC1
16,17	Sea-Bird SBE3F thermometer.	SBT
18,19	Sea-Bird SBE4C conductivity sensor.	SBC
32	Voltage of the battery or power source.	V_Bat
40	X-axis from an ADIS dual-axis digital inclinometer.	Incl_X
41	Y-axis from an ADIS dual-axis digital inclinometer.	Incl_Y
42	Temperature from an ADIS dual-axis digital inclinometer.	Incl_T
48, 49	JAC conductivity sensor.	JAC_C
50	JAC thermometer.	JAC_T
52	JAC fluorometer.	Chlorophyll
53	JAC backscatter sensor.	Turbidity
64	Micro-conductivity sensor #1 without pre-emphasis.	C1
65	Micro-conductivity sensor #1 with pre-emphasis.	C1_dC1
66	Micro-conductivity sensor #2 without pre-emphasis.	C2
67	Micro-conductivity sensor #2 with pre-emphasis.	C2_dC2
80	Vertical component of magnetic field from magnetometer.	Mz
81	Horizontal component of magnetic field from magnetometer.	My
82	Horizontal component of magnetic field from magnetometer.	Mx
255	This special character should be present at the start of every channel matrix. It is used for error detection and correction.	Sp_Char

Table 3.7.2: Partial list of channel ids and their typical corresponding sensors and signals.

3.8 [channel] Section

Channel sections are used to describe a channel. The parameters in a section identify the address of a signal, the name it will have in the mat-file, its type and the values used to convert the signal into physical units.

It is sufficient (and recommended) to identify the start of a channel section using `[channel]`, but for backwards compatibility, the section name `channel` is not required.

The parameter "`type`" is important because additional parameters are required depending on it's value. These additional parameters are used to convert raw data into physical units. See section 4 for a complete list of additional parameters.

The required and optional parameters for a channel section are shown in table 3.8.1.

Parameter	Description
`id`	Channel number, or address, as specified in the address matrix (section 3.7). This value must be unique.
`name`	String reference to this channel. This is the name that the channel will have in the mat-file. The channel will be converted into physical units regardless of the value of `name`, but most data processing software makes assumptions about the names of signals. This value must be unique. The RSI recommended names are in table 3.7.2.
`type`	Identifies how a raw signal is converted into physical units. The recognized values are listed in table 4.0.2. Most types require additional parameters for the conversion into physical units (see section 4).
`diff_gain`	The gain of the analog differentiator used to apply pre-emphasis to this signal. This value is found within the instrument calibration report. In most cases, this value is already entered into the `setup.cfg`-file that was shipped with your instrument. These channels typically have the name `X_dX` where `X` is the channel name without pre-emphasis. This parameter is only used for channels with pre-emphasis.
`units`	(optional – ODAS-RT only) A string for units of this signal. Used only for real-time display with real-time telemetering instruments. This string should be short for good readability of the display.
`display`	(optional – ODAS-RT) Boolean value that determines if the record-average value of a signal is displayed within the ODAS-RT gui. Default value is `true` for all signals without pre-emphasis. Use `false` to suppress the display of a signal

Table 3.8.1: Parameters for section `[channel]`

4 Channel Types and Conversion into Physical Units

In the following subsections, the variable N refers to the raw data value (counts) reported by an instrument. It is a 16-bit signed integer for all channels, except as noted in Table 4.0.2.

Valid channel types are listed in table 4.0.2.

Section	Type	Description
4.1	`accel`	Linear accelerometer.
4.2	`amt_o2`	AMT dissolved oxygen sensor.
4.3	`gnd`	ADC (analog-to-digital converter) ground.
4.4	`inclt`	Temperature from an ADIS dual-axis digital inclinometer.
4.5	`inclxy`	Angular data from an ADIS dual-axis digital inclinometer.
4.6	`jac_c`	JAC conductivity sensor using JAC electronics. 32-bit unsigned integer.
4.7	`jac_t`	JAC thermometer using JAC electronics. 16-bit unsigned integer.
4.8	`magn`	3-axis magnetometer.
4.9	`o2_43f`	Sea-Bird SBE43F dissolved oxygen sensor. 32-bit unsigned integer.
4.10	`piezo`	Piezo-accelerometer (vibration) sensor.
4.11	`poly`	Generic polynomial up to order 9.
4.12	`rsi_jac_c`	JAC conductivity sensor using RSI electronics. 32-bit float.
4.13	`rsi_jac_t`	JAC thermometer using RSI electronics. 32-bit float.
4.14	`sbc`	Sea-Bird SBE4C conductivity sensor. 32-bit unsigned integer.
4.15	`sbt`	Sea-Bird SBE3F thermometer. 32-bit unsigned integer.
4.16	`shear`	Shear probe.
4.18	`t_ms`	FP07 thermistor, using a MicroSquid or MicroPod instrument.
4.17	`therm`	FP07 thermistor, using the ASTP (acceleration, shear, temperature and pressure) board.
4.19	`ucond`	Micro-conductivity sensor.
4.20	`vector`	A Nortek model Vector ADV (acoustic doppler velocimeter) voltage signal.
4.21	`voltage`	Generic voltage signal.

Table 4.0.2: Channel types used for converting raw data into physical units.

4.1 type = `accel`

The "`accel`" type describes a channel that samples a linear accelerometer. Two coefficients, a_0 and a_1, are used to generate the acceleration in physical units of $\mathrm{m\,s^{-2}}$ from the observed value of N (counts) using the formula;

$$A = 9.81 \left(\frac{N - a_0}{a_1} \right) \left[\mathrm{m\,s^{-2}} \right] \ . \tag{3}$$

The required parameters are:

Parameter	Variable	Description
coef0	a_0	offset
coef1	a_1	inverse slope

Listing 3: Example channel section for type accel.

```
[channel]
id    = 1
name  = Ax
type  = accel
coef0 = 0.5
coef1 = 0.689
units = [m/s^2]
```

Please, see section 4.10 for notes regarding piezo-accelerometers (also called vibration sensors).

4.2 type = amt_o2

The "amt_O2" type identifies a channel that presents the voltage signal from an AMT dissolved oxygen sensor. The conversion into physical units is too complicated to describe here. It is fully documented in the Micro-Squid Dissolved Oxygen Instrument User Manual, that was shipped with your instrument.

The required parameters are as follows.

Parameter	Description
adc_bits	Number of bits in the analog-to-digital converter, typically 16.
adc_fs	Full-scale voltage range of the analog-to-digital converter, typically 5.120.
adc_zero	Voltage of analog-to-digital converter corresponding to a raw datum of $N = 0$.
b0 -- b3	Temperature coefficients of sensor. Supplied by AMT with each sensor.
U0	Voltage of AMT sensor for zero dissolved oxygen concentration, typically 0.125. Determined by the user.
U100	Voltage at 100 % dissolved oxygen concentration, typically 2 to 4. Determined by the user.
T_cal	Temperature of above sensor calibration, in °C.
PL	Atmospheric pressure at time of calibration, in mbar.
cal_date	Optional date and time of calibration.
SN	Highly recommended optional serial number of the AMT sensor. This parameter provides a check on b0 -- b3.

Listing 4: Example channel section for type amt_o2.

```
[channel]
; The microSquid-DO
id       = 81
name     = AMT_O2
type     = amt_o2

; These parameters are for using raw counts data from an
; analog-to-digital converter.
adc_bits = 16
adc_fs   = 5.120
adc_zero = 2.56

; From AMT sensor report
b0       = +1.217597e0
b1       = -1.292987e-2
b2       = +1.321276e-4
b3       = -1.491988e-6
SN       = 12170707

; The next parameters come from the customer's own
; calibration of the AMT DO sensor
U0       = 0.125      ; Voltage at zero-oxygen concentration
U100     = 2.950      ; Voltage in well bubbled water
T_cal    = 21.7       ; temperature of water, celsius
PL       = 1000       ; air pressure in mbar
cal_date = 2014-06-23
```

4.3 type = gnd

The "gnd" type identifies a channel that is used to monitor the ground signal of an AD converter. These channels are used to monitor the noise level of the converter and to check for unusual offsets. This type can also be used for any signal that you do not want to convert into physical units.

$$N = N \tag{4}$$

Parameter	Description
	No coefficients required.

Listing 5: Example channel section for type gnd.

```
[channel]
id    = 0
name  = Gnd
type  = gnd
units = [counts]
```

4.4 type = inclt

This type is used to convert the temperature data from an ADIS dual-axis digital inclinometer (see also section 4.5).

The ADIS16209 inclinometer produces 16-bit words containing both data and status flags. Conversion into physical units consists of checking and stripping the flags, followed by a linear polynomial evaluation. The function adis

$$N_a = \text{adis}(N) \tag{5}$$

clears the flags, and the conversion is

$$T = a_0 + a_1 N_a \quad [\ ^\circ\text{C}\ ] \ . \tag{6}$$

Parameter	Variable	Description
coef0	a_0	From above, usually set to 624.
coef1	a_1	From above, usually set to -0.47.

Listing 6: Example channel section for type `inclt`.

```
[channel]
id    = 42
name  = Incl_T
type  = inclt
coef0 = 624
coef1 = -0.47
units = [C]
```

4.5 type = inclxy

This type is used to convert the angular data from an ADIS dual-axis digital inclinometer (see also section 4.4)

The ADIS16209 inclinometer produces 16-bit words containing both data and status flags. Conversion into physical units consists of checking and stripping the flags, followed by a linear polynomial evaluation. The function `adis`

$$N_a = \text{adis}(N) \tag{7}$$

clears the flags, and the conversion is

$$\phi = a_0 + a_1 N_a \quad [\,^\circ\,] \ . \tag{8}$$

Parameter	Variable	Description
`coef0`	a_0	From above, usually set to 0.
`coef1`	a_1	From above, usually set to 0.025.

Listing 7: Example section for type `inclxy`.

```
[channel]
id    = 40
name  = Incl_X
type  = inclxy
coef0 = 0
coef1 = 0.025
units = [deg]

[channel]
id    = 41
name  = Incl_Y
type  = inclxy
coef0 = 0
coef1 = 0.025
units = [deg]
```

4.6 type = jac_c

This type is used to convert the conductivity channels from a JAC_CT sensor (JAC, JFE Advantech Company) that is connected to JAC electronics. Two 16-bit channels, one for current and the other for voltage, are joined into a 32 bit numeric value by the `read_odas` function. The convert function extracts these channels to calculate conductance, Y, which is the ratio of current-to-voltage. The conductance is converted into conductivity using the polynomial coefficients in the calibration report provided by JAC. Namely,

$$Y = \frac{N_I}{N_V} \tag{9}$$

$$\sigma = A + BY + CY^2 \quad \left[\mathrm{mS\,cm^{-1}}\right] \ . \tag{10}$$

Listing 8: Example channel section for type `jac_c`.

```
[channel]
id    = 64, 65
name  = JAC_C
type  = jac_c
A     = +8.155651e-03
B     = +3.856615e+01
C     = -1.460329e-02
units = [mS/cm]
```

4.7 type = jac_t

This type is used to convert the temperature channel from a JAC_CT sensor (JAC, JFE Advantech Company) that is connected to JAC electronics. The coefficients are obtained directly from the calibration report provided by JAC. The data, N, are 16-bit unsigned integers.

$$T = A + BN + CN^2 + DN^3 + EN^4 + FN^5 \quad [\ ^\circ\mathrm{C}\ ] \ . \tag{11}$$

Listing 9: Example channel section for type `jac_t`.

```
[channel]
id     = 66
name   = JAC_T
type   = jac_t
A      = -5.523044e+00
B      = +1.068060e-03
C      = -1.249469e-08
D      = +2.876148e-13
E      = -3.488626e-18
F      = +2.531447e-23
units  = [C]
```

4.8 type = magn

This type is used to convert data from a 3-axis magnetometer (such as PNI MicroMag3). The magnetic field, in units of µT along each axis is:

$$M = \frac{N - a_0}{a_1} \quad [\mu\mathrm{T}] \ . \tag{12}$$

The heading can be calculated with;

$$\frac{180}{\pi}\mathrm{angle}(M_x + jM_y) \quad [\ ^\circ\ ] \ . \tag{13}$$

Parameter	Variable	Description
Parameters found in the instrument calibration report:		
`coef0`	a_0	From above
`coef1`	a_1	From above

Listing 10: Example section of type magn.

```
[channel]
id    = 33
name  = Mx
type  = magn
coef0 = -15.5
coef1 = -64.03
units = [uT]

[channel]
id    = 34
name  = My
type  = magn
coef0 =-17.5
coef1 =-66.05
units =[uT]
```

4.9 type = o2_43f

This type is used to convert data from a Sea-Bird SBE43F Dissolved Oxygen sensor. The data, N, are unsigned 32-bit words formed from two 16-bit channels. The first step is a conversion of the raw data to the frequency of the sensor output, using

$$f = \frac{n_p \cdot f_{ref}}{N} \quad [\text{Hz}] \ , \tag{14}$$

followed by the conversion of frequency into percent dissolved oxygen saturation, using

$$O_2 = 100 S_{oc} \left(F_{offset} + f \right) \quad [\%] \ . \tag{15}$$

Parameter	Variable	Description
coef0	F_{offset}	Frequency offset, sensor dependent.
coef1	Soc	Sensor slope, sensor dependent.
Instrument dependent parameters:		
coef2	f_{ref}	Reference frequency, usually 24×10^6 Hz.
coef3	n_p	Number of periods of f used to derive N, usually 128.

Listing 11: Example channel section for type o2_43f.

```
[channel]
id    = 34, 35
name  = O2_43F
type  = o2_43f
coef0 = 1.9179e-1
coef1 = 2.4380e-4
coef2 = 24e6
coef3 = 128
units = [%]
```

4.10 type = piezo

The "piezo" type describes a channel that samples a piezo-accelerometer, which is also called a vibration sensor. This type has one optional coefficients, a_0. The signal is not converted into physical units because these sensors are not calibrated. They only need to have an output that is linear with respect to acceleration. The conversion of the raw data, N [counts], uses the formula;

$$A = N - a_0 \quad [\text{ counts }] \ . \tag{16}$$

The optional parameter is:

Parameter	Variable	Description
a_0	a_0	value subtracted from the raw data, usually to remove an offset.

Listing 12: Example channel section for type piezo.

```
[channel]
id    = 81
name  = Ax
type  = piezo
a_0   = -6453
units = [counts]
```

Usually, a_0 is zero and can be omitted, such as when the data are collected with an ASTP board. For data collected with a GPIO board, the offset of this board can be removed using the value given above. However, the signal processing is unchanged other than that the figure of the profile of acceleration will be close to zero centred.

The type piezo is new to the ODAS Library. It is necessary to have this new type in order to plot vibration spectra "on scale" with shear spectra. The piezo accelerometers are not calibrated and are left in raw units, and this makes the data values quite large compared to a linear accelerometer that is converted into physical units. However, RSI has

previously instructed users to declare the piezo accelerometers as type `accel` and declare its two coefficients to be `0` and `1`. This means that existing data files have a technically erroneous configuration string. To avoid amending existing data files, the function `setupstr`, which parses the information out of the configuration string, looks for channels of type `accel` that have coefficients of `0` and `1` and recognizes them as type `piezo`. There is no possibility that a linear accelerometer will ever have these coefficients, because such a sensor would have a resolution of g and be useless.

4.11 type = poly

This type applies a polynomial to the data.

$$N_a = \sum_{k=0}^{m} a_k N^k \quad \text{[units sensor dependent]} \ . \tag{17}$$

Parameter	Variable	Description
`coef0`	a_0	Least significant coefficient.
`coef1`	a_1	Coefficients must be contiguous, starting at 0.
`coef2`	a_2	
⋮	⋮	
`coef`m	a_m	Most significant coefficient with maximum value of $m = 9$.

The coefficients must be contiguous up to the order beyond which they are all zero. So, if a coefficient, say a_j, is zero but some coefficient a_m, with $m > j$, is not zero, then you must include coefficient $a_j = 0$ in the list of coefficients.

Listing 13: Example channel section for type `poly`.

```
[channel]
id    = 10
name  = P
type  = poly
coef0 = 3.6700e0
coef1 = 6.2076e-2
coef2 = 4.7224e-8
units = [dbar]

[channel]
id    = 88
name  = UnknownSensor
type  = poly
coef0 = 0
coef1 = 1
```

In listing 13, the first example contains typical values for a pressure channel. The second example utilizes coefficients that result in a polynomial expression that does not modify the

data values. The same result is obtained by declaring the channel to be
`type = gnd` .

4.12 type = rsi_jac_c

This type is used to convert the conductivity channels from a JAC_CT sensor (JAC, JFE Advantech Company) connected to the RSI electronics. This sensor uses two 16-bit channels to hold one 32-bit, single-precision, floating point value that represents the conductance, Y of this sensor. The conductance is converted into conductivity using a linear (first-order) polynomial. The first coefficient is a small offset while the second is the inverse of the cell constant. There is no correction for the thermal coefficient of expansion of the cell nor for its pressure coefficient of contraction, because both are currently unknown.

$$\sigma = 10\,(a + bY) \quad \left[\ \mathrm{mS\,cm^{-1}}\ \right] \ . \tag{18}$$

Listing 14: Example channel section for type `rsi_jac_c`.

```
[channel]
id    = 64, 65
name  = JAC_C
type  = rsi_jac_c
a     = +0.0031
b     = +37.885
units = [mS/cm]
```

4.13 type = rsi_jac_t

This type is used to convert the temperature channel from a JAC_CT sensor (JAC, JFE Advantech Company) connected to the RSI electronics. This sensor uses two 16-bit channels to hold one 32-bit, single-precision, floating point value that represents the non-dimensional resistance (resistance ratio, R_T/R_0) of the thermistor. R_T and R_0 are the resistance of the thermistor and and a reference resistor, respectively. The temperature is calculated with the Steinhart-Stein equation, namely

$$T = \left\{ \frac{1}{T_0} + \sum_{k=1}^{3} \frac{1}{\beta_k} \cdot \ln^k \left(\frac{R_T}{R_0} \right) \right\}^{-1} - 273.15 \quad [\ ^\circ\mathrm{C}\ ] \ . \tag{19}$$

Listing 15: Example channel section for type `jac_t`.

```
[channel]
id    = 66, 67
name  = JAC_T
type  = rsi_jac_t
T_0     = 285.631
beta_1  = 3988.662
beta_2  = 3.7655e5
beta_3  = 2.5478e7
units = [C]
```

4.14 type = sbc

This type is used to convert data from a Sea-Bird SBE4C conductivity sensor. The data, N, are unsigned 32-bit words formed from two 16-bit channels. The conversion consists of calculating the sensor frequency

$$f = \frac{n_p \, f_{ref}}{1000 \, N} \quad [\text{ kHz }] \ , \tag{20}$$

followed by the calculation of conductivity

$$\sigma = g + hf^2 + if^3 + jf^4 \left[\text{ mS cm}^{-1} \right] \ . \tag{21}$$

Parameter	Variable	Description
Parameters found in the Sea-Bird Calibration report:		
coef0	g	
coef1	0	This coefficient is always zero and must be specified.
coef2	h	
coef3	i	
coef4	j	
Instrument dependent parameters:		
coef5	f_{ref}	Reference frequency, usually 24×10^6 Hz.
coef6	n_p	Number of periods of f used to derive N, usually 128.

Listing 16: Example channel section for type `sbc`.

```
[channel]
id    = 18, 19
name  = SBC
type  = sbc
coef0 = -9.90912533e0
coef1 =  0
coef2 =  1.41415635e0
coef3 =  3.07973519e-4
coef4 =  5.32880619e-5
coef5 =  24e6
coef6 =  128
units = [mS/cm]
```

4.15 type = sbt

This type is used to convert data from a Sea-Bird SBE3F thermometer. The data, N, are unsigned 32-bit words formed from two 16-bit channels. The conversion consists of calculating the sensor frequency

$$f = \frac{n_p \, f_{ref}}{N} \quad [\text{Hz}] \ , \tag{22}$$

followed by the calculation of temperature

$$T = \left\{ g + h \ln\left(\frac{f_0}{f}\right) + i \ln^2\left(\frac{f_0}{f}\right) + j \ln^3\left(\frac{f_0}{f}\right) \right\}^{-1} - 273.15 \quad [\ ^\circ\text{C}\ ] \ . \tag{23}$$

Parameter	Variable	Description
Parameters found in the Sea-Bird Calibration report:		
`coef0`	g	
`coef1`	h	
`coef2`	i	
`coef3`	j	
`coef4`	f_0	
Instrument dependent parametersl:		
`coef5`	f_{ref}	Reference frequency, usually 24×10^6 Hz.
`coef6`	n_p	Number of periods of f used to derive N, usually 128

Listing 17: Example channel section for type `sbt`.

```
[channel]
id    = 16, 17
name  = SBT
type  = sbt
coef0 = 4.35614236e-3
coef1 = 6.38892665e-4
coef2 = 2.11815756e-5
coef3 = 1.79451268e-6
coef4 = 1000.0
coef5 = 24e6
coef6 = 128
units = [C]
```

4.16 type = shear

This type is used to convert data from a shear probe sensor. Conversion does not account for the speed of profiling. It is left to the user to determine the speed. On vertical profilers, a suitable vector of speed can be developed from the high-resolution pressure record, which should be smoothed with a low-pass filter using a cut-off of approximately 1 or 2 Hz. On horizontal profilers, speed information may not be in your data file. The profiling speed has to be obtained from other sources, such as a hotel-file produced by the controller on a vehicle. To complete the conversion to physical units, you must divide the result, $\hat{s}$, of this type of conversion by the square of the speed, [m s^{-1}], of profiling.

$$\hat{s} = \frac{N \, \frac{adc\_fs}{2^{adc\_bits}} + adc_{zero} - sig_{zero}}{2\sqrt{2} \cdot diff\_gain \cdot sens} \quad . \tag{24}$$

Parameter	Description
Parameters found in the Instrument Calibration report:	
`adc_fs`	Full-scale voltage of the analog-to-digital converter, typically 4.096 or 5.120.
`adc_bits`	Number of bits in the analog-to-digital converter, typically 16.
`adc_zero`	Optional. The voltage at which the analog-to-digital converter output is zero. Default is zero.
`sig_zero`	Optional. The voltage at which the sensor output is zero. Default is zero.
`diff_gain`	Differentiator gain in seconds.
Sensor dependent parameters:	
`sens`	shear probe sensitivity [V per m^2/s^2] .

Listing 18: Example channel section for type `shear`.

```
[channel]
id        = 8
name      = sh1
type      = shear
sens      = 0.087
adc_fs    = 4.096
adc_bits  = 16
diff_gain = 1.0
```

NOTE: Remember to divide $\hat{s}$ by speed-squared before interpreting your data as shear in units of s^{-1}.

4.17 type = `therm`

This type is used to convert data from an FP07 thermistor into temperature, when the thermistor is connected to an ASTP board. The conversion is a three-step process. First, the raw data, N, are converted into a non-dimensionalized thermistor voltage, Z, using

$$Z = \frac{N - a}{b} \cdot \frac{adc\_fs}{2^{adc\_bits}} \cdot \frac{2}{G\ E_B} \ , \tag{25}$$

which is used to calculate the non-dimensional thermistor resistance

$$\frac{R_T}{R_0} = \frac{1 - Z}{1 + Z} \ , \tag{26}$$

where R_T and R_0 are the resistance of the thermistor and a reference resistor, respectively. This ratio is converted into temperature using the Steinhart-Stein equation, namely

$$T = \left\{ \frac{1}{T_0} + \sum_{k=1}^{m} \frac{1}{\beta_k} \cdot \ln^k \left(\frac{R_T}{R_0} \right) \right\}^{-1} - 273.15 \quad [\ ^\circ\mathrm{C}\ ] \ . \tag{27}$$

Parameter	Description
Parameters found in the Instrument Calibration report:	
a	offset, in units of counts
b	slope
G	Gain applied to signal before sampling.
E_B	Bridge excitation voltage.
adc_fs	Full-scale voltage of the analog-to-digital converter, typically 4.096 or 5.120.
adc_bits	Number of bits in the analog-to-digital converter, typically 16.
diff_gain	Differentiator gain in seconds. Only required for pre-emphasized channels.
Parameters found in the Thermistor Calibration report:	
beta_1	The first material constant
beta_2	The second, optional, material constant
T_0	Temperature, in K, at which the resistance of the thermistor, R_T equals the reference resistor, R_0

Listing 19: Example channel section for type `therm`.

```
[channel]
id         = 4
name       = T1
type       = therm
a          = -6.4
b          = 0.99854
G          = 6
E_B        = 0.68247
beta_1     = 3143.55
beta_2     = 249529
T_0        = 289.301
adc_fs     = 4.096
adc_bits   = 16
units      = [C]

[channel]
id         = 5
name       = T1_dT1
type       = therm
diff_gain  = 0.99
```

The above example is for a thermistor which produces two signals that appear on two different channels. The temperature signal, which is approximately linear with respect to temperature, is on channel "`id=4`" with name "`T1`". The thermistor also produces a pre-emphasized signal on channel "`id=5`" with name "`T1_dT1`". Both channels are of type "`therm`". The gain of the differentiator "`diff_gain`" is the only calibration parameter required for "`T1_dT1`". It can be used to deconvolve "`T1_dT1`" to produce a high resolution temperature value. Calibration coefficients shown for channel "`id=4`" can also be entered for channel "`id=5`". However, the repeat entry of the calibration coefficients is not necessary if the user chooses to use the coefficients associated with "`id=4`" when converting the "`T1_dT1`" signal into physical

units.

There is currently no justification for using an order higher than 2 for the calculation of temperature (27) because the FP07 thermistor is not protected from pressure and will give a depth-dependent error that could be as large as a few milli degrees. That is, $m \leq 2$ in (27). The first- and second-order equations are accurate to $\pm 0.04\,^{\circ}\mathrm{C}$ and $\pm 0.004\,^{\circ}\mathrm{C}$, respectively, over the oceanic temperature range, at atmospheric pressure.

4.18 type = t_ms

This type is used to convert data from an FP07 thermistor into temperature, when the thermistor is connected to a MicroSquid or a MicroPod board. The conversion is a four-step process. First, the raw data, N, are converted into an absolute voltage, Z_a, using

$$Z_a = N \frac{adc\_fs}{2^{adc\_bits}} + adc_{zero} \ , \tag{28}$$

which is converted into a non-dimensional voltage, Z, using

$$Z = \frac{Z_a - a}{b} \cdot \frac{2}{G\ E_B} \ . \tag{29}$$

This non-dimensional voltage is then used to calculate the non-dimensional thermistor resistance, using

$$\frac{R_T}{R_0} = \frac{1 - Z}{1 + Z} \ , \tag{30}$$

where R_T and R_0 are the resistance of the thermistor and a reference resistor, respectively. The resistance ratio is converted into temperature using the Steinhart-Stein equation, namely

$$T = \left\{ \frac{1}{T_0} + \sum_{k=1}^{m} \frac{1}{\beta_k} \cdot \ln^k \left(\frac{R_T}{R_0} \right) \right\}^{-1} - 273.15 \quad [\ ^{\circ}\mathrm{C}\ ] \ . \tag{31}$$

Parameter	Description
Parameters found in the Instrument Calibration report:	
a	offset, in units of volts
b	slope
G	Gain applied to signal before sampling.
E_B	Bridge excitation voltage.
adc_fs	Full-scale voltage of the analog-to-digital converter, typically 4.096 or 5.120.
adc_bits	Number of bits in the analog-to-digital converter, typically 16.
adc_zero	The voltage at which the analog-to-digital converter output is zero. Defaults to zero.
diff_gain	Only required for pre-emphasis channels - differentiator gain in seconds.
Parameters found in the Thermistor Calibration report:	
beta_1	The first material constant
beta_2	Optional. The second material constant
T_0	Temperature, in K, at which $R_T = R_0$

Listing 20: Example channel section for type t_ms.

```
[channel]
id         = 84
type       = t_ms
name       = T1_dT1
diff_gain  = 9.6
adc_zero   = 2.560
adc_fs     = 5.120
adc_bits   = 16
a          = 2.04701
b          = 1.0002
G          = 6
E_B        = 0.68237
;
SN         = T716
beta_1     = 3009.07
beta_2     = 265218
T_0        = 265.218
cal_date   = 2015-05-28
```

The MicroSquid and MicroPod sensors produce only a single, pre-emphasized analog voltage signal. So, the parameter diff_gain has to be included. In this example, the data were collected with the General Purpose Input Output (GPIO) board. Its analog-to-digital converter has the specifications shown in the example.

There is currently no justification for using an order higher than 2 for the calculation of temperature because the FP07 thermistor is not protected from pressure and will give a depth dependent error that could be as large as a few milli degrees. That is, $m \leq 2$ in equation 31. The first- and second-order equation is accurate to $\pm 0.04\,^{\circ}\mathrm{C}$ and $\pm 0.004\,^{\circ}\mathrm{C}$, respectively, over the oceanic temperature range at atmospheric pressure.

4.19 type = ucond

This type is used to convert data from a micro-conductivity sensor into physical units. The conversion is a three-step process. First, the data are converted into an absolute voltage, Z_a, using

$$Z_a = \frac{adc_{fs}}{2^{adc_{bits}}} N + adc_{zero} \ , \tag{32}$$

which is converted into a conductance, Y, using

$$Y = \frac{Z_a - a}{b} \ \ [\ \mathrm{S}\ ] \ . \tag{33}$$

The conductance is then converted into conductivity using the cell-constant, K.

$$\sigma = 10\frac{Y}{K} \ \ [\ \mathrm{mS\,cm^{-1}}\ ] \ , \tag{34}$$

The cell constant has units of m.

Parameter	Description
Parameters found in the Instrument Calibration report:	
a	offset
b	slope
adc_fs	Full-scale voltage of the analog-to-digital converter, typically 4.096 or 5.120.
adc_bits	Number of bits in the analog-to-digital converter, typically 16.
adc_zero	Optional. The voltage at which the analog-to-digital converter output is zero. Defaults to zero.
Parameters found in the Sensor Calibration report:	
K	sensor cell constant [m]

Listing 21: Example channel section for type ucond.

```
[channel]
id        = 64
name      = C1
type      = ucond
a         = 1.216
b         = 196.7
K         = 1.03e-3
adc_fs    = 4.096
adc_bits  = 16
adc_zero  = 0
units     = [mS/cm]

[channel]
id        = 65
name      = C1_dC1
type      = ucond
diff_gain = 1.0
```

4.20 type = vector

This type is used to convert the analog voltage output from a Nortek Vector ADV (Acoustic Doppler Velocimeter). The output from the ADV is centred on 2.500 V at a velocity of zero. The sensitivity (or scale factor) of the velocity with respect to voltage, $\mathrm{m\,s^{-1}\,V^{-1}}$, is determined through the configuration of the ADV by the user. The data is typically acquired with the General Purpose Input Output board which has an input range of 0 to 5.120 V and is centred on 2.560 V. The conversion, for velocity component U, is

$$U = \left(adc_{zero} + \frac{adc_{fs}}{2^{adc_{bits}}}\ N - V_{offset} \right)\ S + bias \quad \left[\ \mathrm{m\,s^{-1}}\ \right]\ . \tag{35}$$

where S is the sensitivity of the ADV output.

Parameter	Description
adc_fs	Full-scale voltage of the analog-to-digital converter, typically 5.120.
adc_bits	Number of bits in the analog-to-digital converter, typically 16.
adc_zero	The voltage at which the analog-to-digital converter outputs zero.
offset	The offset of the output voltage of the ADV, typically 2.500.
sens	The sensitivity of the output of the ADV. This is configurable by the ADV user.
bias	Optional. The bias of the ADV output in units of $\mathrm{m\,s^{-1}}$. Defaults to zero.

Listing 22: Example section for type vector.

```
[channel]
id          = 83
name        = U
type        = vector
; instrument dependent parameters
adc_fs      = 5.120
adc_bits    = 16
adc_zero    = 2.560
; sensor dependent parameters
offset      = 2.500 ; voltage corresponding to zero velocity
sens        = 14/5 ; the sensitivity, metres-per-second per volt
bias        =-0.011 ; in m/s, bias due to ADC and DAC.
; units     = [m/s]
```

In this example, the ADV was configured for a range of $\pm 7\,\mathrm{m\,s^{-1}}$, making the sensitivity equal to $14/5\,\mathrm{m\,s^{-1}\,V^{-1}}$. The sensitivity can be expressed using any valid Matlab syntax[4], and it is convenient to leave it as a fraction. The bias is usually very small, optional, and defaults to zero.

Similar sections should be added for the other two velocity components.

[4]This does not work for the real-time display of a telemetering instruments.

4.21 type = voltage

This type is used to convert data that is to be represented as a voltage.

$$V = \left(\frac{adc_{fs}}{2^{adc_{bits}}} \; N - adc_{zero} \right) G^{-1} \quad [\text{ V }] \; . \tag{36}$$

Parameter	Description
G	Gain applied to signal before sampling by the analog-to-digital converter.
adc_fs	Full-scale voltage of the analog-to-digital converter, typically 4.096 or 5.120.
adc_bits	Number of bits in the analog-to-digital converter, typically 16.
adc_zero	Optional. The voltage at which the analog-to-digital converter outputs zero. Defaults to zero.

Listing 23: Example section for type voltage.

```
[channel]
id       = 32
name     = V_Bat
type     = voltage
G        = 0.1
adc_fs   = 4.096
adc_bits = 16
adc_zero = 0
units    = [V]
```

In this example, the battery voltage of an instrument, which is usually monitored on channel id=32, is attenuated by a factor 10 (G=0.1) before it is sampled.

5 Common Tasks

5.1 Extract Configuration String

It is possible to extract the configuration string from any RSI v6 or higher data file. The ODAS Matlab library provides the function `extract_setupstr.m` to perform this task.

Syntax

```
>> extract_setupstr data_file_name
>> result = extract_setupstr('data_file_name');
>> extract_setupstr('data_file_name', 'config_file_name');
```

Examples

The configuration string embedded in a data file named "`DAT_001.P`", is extracted and saved into the text file named"`DAT_001.cfg`" using

```
>> extract_setupstr( 'DAT_001', 'DAT_001.cfg');
```

There are three methods of extracting the configuration string and printing it on the screen:

```
>> extract_setupstr DAT_001
>> extract_setupstr( 'DAT_001' );
>> extract_setupstr( 'DAT_001.P' );
```

The file extension is optional. However, had the file been specified with a lower case extension, `DAT_001.p`, the call would have failed, and returned a warning indicating that the requested file could not be found, because the actual file name is `DAT_001.P`.

5.2 Patch a corrected data file

It is possible to patch a configuration file into an existing RSI data file. This is commonly done to correct erroneous coefficients in a data file. It is also used to upgrade a pre-v6 data file into a v6 data file, so that it can be processed with the ODAS Matlab library.

Data files are patched with the function `patch_setupstr.m`.

Syntax

```
>> patch_setupstr data_file_name config_file_name
>> patch_setupstr('data_file_name', 'config_file_name');
```

The parts of the data file that controlled the acquisition must not be changed. Generally, avoid changes to the [root] and [matrix] sections of the configuration string.

Example - Patching a Data File

If data file named "DAT_001.p" and configuration file named "corrected_config.cfg" both exist in the current directory, then the configuration string in the data file is replaced using:

```
>> patch_setupstr('DAT_001', 'corrected_config');
```

File extensions are optional, but the file name extension must be ".p" or a ".P", for a data file, and it must be ".cfg" or ".CFG" for a configuration file.

Example - Upgrading a Data File

Older data files can be upgraded to v6 data files using the patch_setupstr function. This is not a trivial task because it requires generating a new configuration file based on the setup file (typically named setup.txt) that was used for data acquisition. The following example assumes the v1 data file DAT_002.P was generated using the setup file setup.txt.

```
>> edit('DAT_002.cfg'), edit('setup.txt');
```

When queried to create the file DAT_002.cfg, select "YES". This opens the editor containing the setup file "setup.txt" and an empty configuration file "DAT_002.cfg". The configuration file must now be constructed based on the setup file. Use listing 2 as a template. Read section 4 for a list of parameters required in each channel. Instrument and probe calibration sheets will be required for your calibration coefficients. After you have created a proper configuration file, patch the data file using

```
>> patch_setupstr( 'DAT_002.P', 'DAT_002.cfg' );
```

If you get an error, your configuration file "DAT_002.cfg" is invalid. Edit the configuration file and try again making sure that the instrument configuration matches the one found in "setup.txt". Also, be sure that all required parameters are included within the new configuration file.

5.3 ODAS conversion example

In this section we will show how the ODAS Matlab Library converts raw data into physical units, using the configuration-file example of Listing 2. Syntax detail is provided in section 6.

Given a vector, P, containing raw pressure data, call convert_odas as follows;

```
>> P_CNV = convert_odas( P, 'P', setupfilestr );
```

where the vector to convert is the first argument to the function. The next argument is a string containing the `name` of the channel according to its specification in the configuration string. The name string is usually identical to the name of the vector, but this is not a requirement. The function will look into all sections that have an `id`-parameter (which indicates that they are channel sections) and tries to find the one that has the parameter "`name=P`". It will then extract all calibration coefficients, and the `type`-parameter, and pass these to a routine. This routine uses the coefficients, in a manner dictated by the `type`-parameter, to produce the output, `P_CNV`, that is in physical units.

5.4 ODAS deconvolution and conversion example

Here we show how the channel entries for thermistor 1 (as shown in Listing 19) are used to deconvolve a channel with pre-emphasis and then convert it into physical units. For background, note that the channel `T1_dT1` can be represented mathematically with

$$T_1 + G_D \frac{\partial T_1}{\partial t}, \tag{37}$$

where $G_D = 0.99\,\mathrm{s}$ is the gain of the differentiator, or the portion of the time rate of change of $T1$ that was added to $T1$ to produce the signal `T1_dT1`.

The command,

```
>> T1_fast = deconvolve( 'T1_dT1', [], T1_dT1, fs_fast, ...
                         setupfilestr );
```

will deconvolve the signal "`T1_dT1`", to remove the addition of the time rate-of-change, and return a new, high resolution, version of "`T1`" in the vector `T1_fast`.

- The first argument is the section name in which the parameter "`diff_gain`" (G_D) is found. The required "`diff_gain`" value is extracted from the supplied "`setupfilestr`". Alternatively, you can provide a numeric "`diff_gain`" value in place of "`setupfilestr`".
- The second argument is used for sensors that also have a channel without pre-emphasis. Initial estimates used in the deconvolution algorithm can exploit the signal without emphasis to improve the initial accuracy. Channels that have a differentiator gain much larger than 1 s, such as the pressure channel, benefit substantially from access to the channel without pre-emphasis. An empty matrix is used when the channel without pre-emphasis is not available.
- The third argument is the vector to deconvolve.

- The fourth argument provides the sampling rate of the vector. The Matlab scalars `fs_fast` and `fs_slow` will be in your mat-file.
- The fifth argument contains the configuration string from which to read the `diff_gain`.

This new, high-resolution but raw, temperature signal can be converted into physical units using the command:

```
>> T1_fast = convert_odas( T1_fast, 'T1', setupfilestr );
```

The syntax is the same as for the pressure conversion (section 5.3). However, the function will use the calibration coefficients located in the `[channel]` section that has the entry `name = T1`, rather than the section that contains `T1_dT1`, because the calibration coefficients reside only in the `T1`-section. There are no redundant (or duplicate) coefficients in the `[channel]` section for `T1_dT1` (see Listing 19).

5.5 odas_p2mat – turn-key conversion into physical units

The function `odas_p2mat` will convert all channels into physical units. It searches for the `type` parameter and converts the channels into physical units, regardless of the name of the channel. The default structure that controls the conversion into physical units can be seen by calling the function without arguments:

```
>> convert_info = odas_p2mat
convert_info =
    MF_extra_points: 0
               MF_k: 4
            MF_k_mag: 1.7000
              MF_len: 256
           MF_st_dev: 'st_dev'
        MF_threshold: []
                 aoa: []
     constant_speed: []
      constant_temp: []
         hotel_file: []
       speed_cutout: 0.0500
          speed_tau: []
        time_offset: 0
            vehicle: ''
```

Fields with names starting with `MF_` are irrelevant for vertical profilers and Slocum gliders.

The field `vehicle` is very important. The proper method is to identify the vehicle in the `[instrument_info]` section of your configuration file (see section 3.6). If so, it should be specified as an empty string, `''`, in the `convert_info` structure, as shown above. If you fail to

identify the `vehicle` in the configuration file *and* you do not specify it in the `convert_info` structure, then it defaults to `vmp`. If you do specify the vehicle in the `convert_info` structure, then that value overrides all other specifications.

Most fields in the structure `convert_info` (it can have any legal name) are for estimating the speed of profiling, which is a challenge for AUVs, the Nemo horizontal profiler, and for the Sea-glider. The field `speed_cutout` is used to limit the lowest estimates of speed in order to avoid large values of gradients due to dividing time-derivatives by speed. Speeds smaller than `speed_cutout` are set to `speed_cutout`.

The speed of profiling will be calculated from the rate-of-change of pressure, for a VMP, and with the additional information of the angle-of-attack, the field `aoa`, for a Slocum glider (in which case, it defaults to 3°). You can specify a `constant_speed` if the rate-of-change of pressure gives a misleading speed (such as for profiles through strong up- and down-drafts in a tidal channel) or simply to get a "first glimps" of your data when the speed information is not yet available.

The field `speed_tau` is for smoothing the speed estimates. Its default value is vehicle-dependent (see the file `default_vehicle_attributes.ini`[5] for a listing). You can override the default values by specifying the smoothing time-scale here.

If the speed of profiling cannot be determined from information in your RSI raw data file, then you can specify the name of a `hotel_file` that contains the speed information (see section 6 for more information on hotel files). If the time stamps in the hotel file are offset from those in your RSI raw binary data file, then you can specify this with a non-zero value for the field `time_offset` so that the speed values are synchronized with your data file.

If you do not trust the temperature obtained from the FP-07 thermistor, then specify a value for the field `constant_temp` because it will be used to estimate the kinematic viscosity (which is later used for estimating the rate of dissipation of kinetic energy, ϵ).

Examples

The command

```
odas_p2mat ('Dat_033')
```

converts the data file `Dat_033.p` and creates the mat-file `Dat_033.mat`, and fills it with your data in physical units and adds many ancillary vectors and scalars, using the vehicle that you specified in the configuration file at the time of data acquisition and the default values for the processing parameters.

The command

```
odas_p2mat ('Dat_033', 'constant_speed', 1.5)
```

[5]This file is located in the directory holding all functions of the ODAS Matlab Library.

Does the same but uses a speed of $1.5\,\mathrm{m\,s^{-1}}$ for converting time derivatives into gradients.

The command

```
my_values = odas_p2mat;
my_values.speed_cutout = 0.1;
my_values.vehicle='slocum_glider';
odas_p2mat ('Dat_033', my_values)
```

converts the raw data file `Dat_033.p` into the file `Dat_033.mat`, in physical units, but uses a speed cutout of $0.1\,\mathrm{m\,s^{-1}}$ and tries to treat the data as if it was collected with a Slocum glider. Treating the data as if they were collected with a Slocum glider will give successful conversion, if the vehicle was not specified in the configuration file (which is bad practice) or if it was wrongly specified in the configuration file.

5.6 quick_look – data visualization and ϵ estimation

The function `quick_look` will call the function `odas_p2mat` if a mat-file does not exist, or if the mat-file was created using parameters different from the ones in the call to `quick_look`.

The default processing parameters can be seen by calling the function without arguments:

```
>> ql_info = quick_look
ql_info =
                  HP_cut: 0.4000
                  LP_cut: 30
                    YD_0: 0
               despike_A: [8 0.5000 0.0400]
               despike_C: [10 1 0.0400]
              despike_sh: [8 0.5000 0.0400]
             diss_length: 8
                    f_AA: 98
                 f_limit: Inf
              fft_length: 2
               fit_2_isr: 1.5000e-05
               fit_order: 3
            make_figures: 1
                 op_area: 'open_ocean'
                 overlap: 4
          profile_min_P: 1
          profile_min_W: 0.2000
    profile_min_duration: 20
             profile_num: 1
       MF_extra_points: 0
                  MF_k: 4
```

```
      MF_k_mag: 1.7000
        MF_len: 256
     MF_st_dev: 'st_dev'
  MF_threshold: []
           aoa: []
constant_speed: []
 constant_temp: []
    hotel_file: []
  speed_cutout: 0.0500
     speed_tau: []
   time_offset: 0
       vehicle: ''
```

All fields after `profile_num` are specific to `odas_p2mat` and are only used if a conversion to physical units is required. The default values are well suited to processing data collected with a VMP. The fields that have names starting with "`profile_`" are used to specify what you mean by a "profile". The data must exceed the minimum pressure *and* the minimum vertical speed for *at least* a minimum duration, in a direction appropriate for your vehicle, in order to be considered a profile. The appropriate directions are; down for a vmp; up for a rvmp; and *both* up and down for a glider (both the Slocum and the Sea-glider, see section 3.6 and the file `default_vehicle_attributes.ini`).

Data files can contain more than one profile, in which case you specify the profile that you want by using the field `profile_num`.

The rate of dissipation will be estimated over segments of length `diss_length` seconds using using fft-segments that are `fft_length` seconds long. Dissipation estimates will be overlapped every `overlap` seconds. Shear probe data are high-pass filtered at `HP_cut` Hz. For some profile graphs, the shear probe data (not the dissipation estimates) are low-pass filtered at `LP_cut` Hz. The anti-aliasing filter in your instrument is assumed to be `f_AA` Hz and `f_limit` is the maximum frequency used to estimate the rate of dissipation. There are additional limits.

Examples

The command

```
diss_033_01 = quick_look('Dat_033', 20, 30)
```

processes the file `Dat_033.p`, creates the mat-file `Dat_033.mat` (if necessary), and creates numerous figures that visualize the first (default) profile, using default parameters, and provides frequency and wavenumber spectra for the depth range of 20 to 30 dbar. It returns the structure `diss_033_01` that contains, among a great many other items, the *profile* of the rate of dissipation, ϵ, (according to the default specifications), spectra of the shear and scalar gradients corresponding to each estimate of ϵ, and the average value of each channel of data (and derived signals) for each ϵ.

The command

```
diss_033_02 = quick_look('Dat_033', 20, 30, 'profile_num', 2)
```

does the same thing as the above example except that it uses the second profile in the data file. The returned structure is given a different name so that the two profiles can be compared.

If many of the default fields that control the processing need to be customized, then it may be wiser to capture the default structure using

```
ql_info = quick_look
```

adjust the value of the fields in `ql_info`, and possibly save them in the local directory, and then use the command

```
diss_033_09 = quick_look('Dat_033', 45, 65, ql_info, 'profile_num', 9)
```

to process, for example, the ninth profile according to the specifications in `ql_info`.

More information is provided in RSI Technical Note 039.

5.7 show_P – the pressure record in your file

You frequently do not know how many profiles are in your data file. The function `show_P` may help you to identify the profiles. For example, the command

```
plot(show_P('Dat_033')); grid on; set(gca,'ydir','rev')
```

will extract the record-average pressure from your raw data file `Dat_033.p`, convert it into physical units, and plot it on the screen. The file does not have to be converted into physical units.

Also, the structure returned by `quick_look` has a field named `profiles_in_this_file` that tells you how many profiles were detected in the file. This number, of course depends on how you specified a "profile", and if the results surprise you, then you should scrutinize your definition of a profile. Specifically, look at the fields in the input structure that begin with `profile_` .

6 ODAS Functions

This section lists all functions that are part of the ODAS Matlab library, as well as the hotel-file scripts.

adis

Convert inclinometer data into meaningful raw counts

Syntax

```
[xOut, oldFlag, errorFlag] = adis( xIn )
```

Argument	Description
`xIn`	Vector of words from inclinometer.
`xOut`	Data vector extracted from words as 2s-complimented count values.
`oldFlag`	Index vector to data elements that are not new.
`errorFlag`	Index vector to data elements with error-flags set.

Normally, errorFlag and oldFlag will be empty vectors.

Description

The ADIS16209 inclinometer outputs words that combine data with status flags. The data and status flags must be seperated before the data can be converted into physical units. This function extracts the data and converts it into valid 16bit, 2s-compliment words.

ADIS16209 words have the following format:

```
    bit15 = new data present
    bit14 = error flag
bit[13:0] = X and Y inclination data, 2s-compliment (X and Y channels)
bit[11:0] = temperature data, unsigned              (temperature channel)
```

Examples

```
>> [raw, oldData, errors] = adis( ADIS16209_words )
```

Extract and convert raw count values from ADIS16209 words.

cal_FP07_in_situ

Calibrate thermistor probe using in situ data

Syntax

```
[T_0, beta, Lag] = cal_FP07_in_situ( file_name, T_ref, T, SN, ... )
```

Argument	Description
file_name	Name of mat-file containing data used to calibrate a thermistor.
T_ref	Name of vector within the mat-file that contains the reference temperature in degrees celsius. Usually from a SBE4F thermometer or a JAC_CT.
T	Name of thermistor to calibrate, typically 'T1' or 'T2'.
SN	Serial number of thermistor.
...	Optional elements describing a profile. Can be provided in a structure or as key / value pairs. See below for more details.
T_0	Value of parameter T0, used in the Steinhart-Hart equation. When called with no input parameters, a structure containing default input parameter values is returned for reference.
beta	beta coefficients, in ascending order, of the fit to the SH equation. i.e. beta_1, beta_2, beta_3
Lag	Delay in seconds between the thermistor and the reference thermometer. Typically a negative value because the reference sensor is usually behind the thermistor being calibrated.

Description

Function to calibrate a FP-07 thermistor probe using in-situ data. Reliable temperature data must be within the specified mat-file. They will be the refercence temperature – usually a Sea-Bird SBE3F or a JAC-CT.

This function makes 5 figures. Figure 1 shows you the portion of the data file that will be used for the calibration. It then detrends the thermistor data and scales it so that it is approximately aligned with the reference thermometer (figure 2). It then plots the cross-correlation coefficient between the thermistor and the reference thermometer and estimates the lag between these two signals (Figure 3). Next it plots the the natural logarithm of the resistance ratio against the inverse of the absolute temperature to show the quality of the regression (Figure 4). Finally, it plots a depth profile of the lagged reference temperature, the newly calibrated thermistor temperature, and their difference.

Optional input parameters include;

Argument	Description
`profile_num`	Integer identifying the profile number to be analyzed. Default = 1. Some files may contain multiple profiles.
`direction`	String identifying the direction of profiling. Either 'up' or 'down' - default = 'down'.
`min_duration`	Minimum time, in seconds, for a profile to be deemed a profile. Direction of travel must be monotonic, rate of change of pressure must be above a minimum level, and pressure must exceed a minimum value. Default = 20 s.
`P_min`	Minimum pressure for a profile. Default = 1 dBar
`min_speed`	Minimum speed of a profile, actually dP/dt. Default = 0.4 dBar/s. Use a smaller value, ~0.1 dBar/s, for gliders.
`order`	Fit order to the Steinhart-Hart equation. Value can be 1, 2, or 3. Default = 2.

channel_sampling_rate

Calculate sampling rate of the requested channel

Syntax

```
rate = channel_sampling_rate( channel, configStr, fs_fast )
```

Argument	Description
`channel`	Name of the requested channel.
`configStr`	Configuration string associated with the data.
`fs_fast`	Actual sampling rate of the instrument in rows / second. This value is extracted from the data file header and typically saved within the variable 'fs_fast'.
`rate`	Sampling rate of specified channel in [samples/second].

Description

The sampling rate defined as the number of times a channel is sampled every second. This function calculates the sampling rate based on the number of times the channel occurs within the channel matrix and the actual sampling rate of the instrument.

Examples

```
>> rate = channel_sampling_rate( 'P', setupfilestr, fs_fast )
```

Returns the sampling rate of channel 'P'. The configuration string found in 'setupfilestr' is used to find the number of occurances of 'P' within the channel matrix. This value, along with 'fs_fast', is used to calculate the resulting channel sampling rate.

check_bad_buffers

Find bad buffers within a data file

Syntax

```
[badRecord] = check_bad_buffers( file )
```

Argument	Description
`file`	Name of data file with or without the '.p' extension.
`badRecords`	Vector index to records with bad buffers.

Description

Scan a RSI raw binary data file for bad buffers. Communication problems can sometimes cause the data acquisition software to drop data. These events are detected and recorded within the data acquisition log file. The header in the corresponding data file is also flagged.

This fuction searches the headers within a RSI binary data file for the bad buffer flag. The indexes of these records are returned in BadRecord.

Examples

```
>> badIndexes = check_bad_buffers( 'my_data_file.p' )
```

Search for bad buffers within the file `my_data_file.p`. If `badIndexes` is empty, no bad buffers were found.

clean_shear_spec

Remove acceleration contamination from all shear channels.

Syntax

```
[cleanUU, AA, UU, UA, F] = clean_shear_spec( A, U, nFFT, rate )
```

Argument	Description
`A`	Matrix of acceleration signals usually represented by [Ax Ay Az] where Ax Ay Az are column vectors of acceleration components.
`U`	Matrix of shear probe signals with each column representing a single shear probe.
`nFFT`	Length of the fft used for the calculation of auto- and cross-spectra.
`rate`	Sampling rate of the data in Hz
`cleanUU`	Matrix of shear probe cross-spectra after the coherent acceleration signals have been removed. The diagonal has the auto spectra of the clean shear.
`AA`	Matrix of acceleration cross-spectra. The diagonal has the auto-spectra.
`UU`	Matrix of shear probe cross-spectra without the removal of coherent acceleration signals. The diagonal has the auto spectra of the original shear signal.
`UA`	Matrix of cross-spectra of shear and acceleration signals.
`F`	Column vector of frequency for the auto- and cross-spectra.

Description

Remove components within a shear probe signal that are coherent with signals reported by the accelerometers. It is very effective at removing vibrational contamination from shear probe signals.

It works only in the spectral domain.

Developed by Louis Goodman (University of Massachusetts) and adapted by RSI.

For best results and statistical significance, the length of the fft (nFFT) should be several times shorter than the length of the vectors, [size(U,1)], passed to this function. That is, several ffts have to be used to form an ensemble-averaged auto- and cross-spectrum.

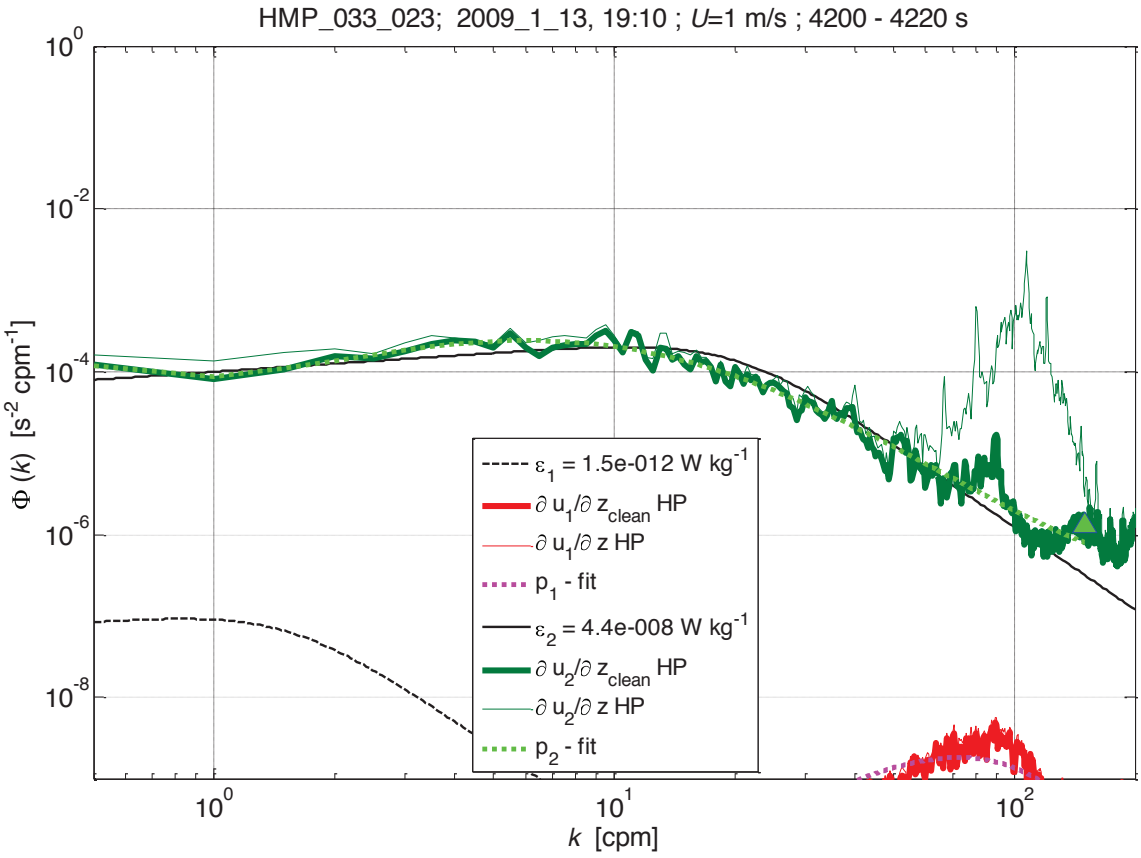

Figure 6.0.1: Original spectrum (thin green line) and after noise removal (thick green line). Data collected with Remus-600 in Buzzards Bay MA, courtesy of Tim Boyd , Scottish Association for Marine Science. Only shear probe 2 was installed, so, the spectrum for probe 1 falls off the deep end of the plot.

conductivity_TPcorr

Apply a correction for the thermal expansion and pressure contraction of a Sea-Bird SBE4C conductivity cell.

Syntax

```
[sbcCorr] = conductivity_TPcorr( sbc, sbt, pres, CPcor, CTcor )
```

Argument	Description
`sbc`	Conductivity from Sea-Bird SBE4C conductivity sensor, in units of mS/cm.
`sbt`	Temperature from Sea-Bird SBE3F thermometer in units of Celsius.
`pres`	Pressure in units of dBar.
`CPcor`	Correction factor for pressure. default: -9.57e-8
`CTcor`	Correction factor for temperature. default: 3.25e-6
`sbcCorr`	Conductivity with correction applied, in units of mS/cm.

Inputs CPcor and CTcor are an optional pair. Both must be either included or excluded.

Description

Apply a correction for the thermal expansion and pressure contraction of the Sea-Bird SBE4C conductivity cell.

The algorithm used:

$$sbcCorr = \frac{sbc}{1 + CTcor \cdot sbt + CPcor \cdot pres}$$

convert_odas

Convert data from raw counts to physical units

Syntax

```
[phys, units] = convert_odas(X, name, empty, config, ver )
```

Argument	Description
`X`	Data vector to convert.
`name`	Name of the channel containing the calibration coefficients.
`empty`	Parameter ignored unless exactly 3 parameters are used in the function call. When three parameters are used, it is assumed that this parameter contains what would have been provided within the 'config' parameter.
`config`	Source of the calibration coefficients. This is typically the variable 'setupfilestr' but can also be the name of a file containing the configuration string. Alternatively, this can be a structure containing the configuration string within the field 'setupfilestr'.
`ver`	Parameter ignored, can optionally be removed. Included for backwards compatibility.
`X_phys`	X converted into physical units.
`units`	String representation of the converted variable units.

Description

Convert data from raw counts into physical units. Data acquisition does not modify the raw data coming from the sensors. All data can be converted using this function.

Information required to perform the conversion from counts to physical units is contained within the configuration string. Users must provide convert_odas with the configuration string and the channel section name name parameter. The convert_odas function will then extract the calibration coefficients and perform the conversion.

Different types of channels use different conversion algorithms. The "type" parameter within the configuration-string determines which algorithm is used. There are functions within convert_odas.m for each supported type. To see exactly how conversion is performed for a specific type, refer to the appropriate embedded function.

Users can add support for new types or change the behaviour of existing types by suppling their own conversion functions. See the example included within this function for more

details.

For detailed information on the conversion from raw data to physical units, please see section 4.

Examples

```
>> T2      = convert_odas( T2,      'T2', '', setupfilestr )
>> T2_fast = convert_odas( T2_fast, 'T2', '', setupfilestr )
```

Convert data using calibration coefficients from section 'T2' found within the configuration file 'setupfilestr'. The 'T2_fast' data is the result of deconvolving the T2_dT2 channel. The correct calibration coefficients for this channel are found within the 'T2' channel section of 'setupfilestr'.

```
>> P_slow = convert_odas( P_slow, 'P', setupfilestr )
```

Convert the deconvolved pressure data, P_slow, using the calibration coefficients found within the 'P' channel section of the configuration string, 'setupfilestr'.

correct_amt

Adjust O2 data for temperature, salinity, and depth

Syntax

```
amt_O2 = correct_amt( O2, T, S, D, setupfilestr, name )
```

Argument	Description
`O2`	Disolved oxygen signal from AMT sensor. Data must be of type 'amt_o2' and converted using convert_odas before using this function.
`T`	Temperature in degree celsius. Can be either empty, a scalar, or a vector. If empty, it defaults to 10 degC.
`S`	Salinity in units of PSU. Can be either empty, a scalar, or a vector. If empty, it defaults to 0 PSU.
`D`	Depth in units of metres. Can be either empty, a scalar, or a vector. If empty, it defaults to 0 m.
`setupfilestr`	Configuration string or file name (with extension) used for data acquisition. If data is recorded with a third-party instrument then you must construct a configuation file according to the specifications in the User Manual for your dissolved oxygen measuring instrument.
`name`	Section name, in the configuration-file or the configuration-string, that contains the temperature coefficients for the sensor. If not provided, the first section with type 'amt_o2' is selected.
`amt_O2`	Oxygen data after being adjusted for temperature, salinity, and depth.

Description

Apply the temperature-, salinity-, and depth-corrections to an AMT O2 signal to complete its conversion into physical units. The input is the AMT O2 signal *after* conversion using the function convert_odas. The channel type must be 'amt_o2'.

The temperature, salinity and depth information can each be supplied in one of three different forms. If you use an empty matrix, [ ], then the default value is applied to every data point of O2. If you specify a scalar (single) value, then this value is applied to every data point. If you supply a vector, then the vector is applied, point-for-point, to the oxygen signal, O2. If you supply a vector is must be the same size as O2.

Corrections are applied in the following manner.

- Initial temperature adjustment by multiplying by ET. The value of ET is provided by AMT.
- Division by the factor (PN - Pw), where PN is a reference pressure (1013) and Pw is the water vapour saturtion pressure (dBar) calculated from temperature.
- Multiplication by the in situ O2 saturation concentration. The O2 saturation concentration is calculated as a function of temperature and salinity.
- Adjust for depth my multiplying by "exp(-0.3775*Depth./T_K)" where Depth is in meteres and T_K is in kelvin.

Example

```
>> O2_tmp = convert_odas( AMT_O2, 'AMT_O2', setupfilestr )
>> AMT_O2 = correct_amt( O2_tmp, T1_fast, [], 15, setupfilestr )
```

Before correct_amt is called, convert_odas is used to perform the initial stage of conversion into physical units. correct_amt is then used to adjust for the specific temperature, salinity, and depth. The above example demonstrates using a data vector for temperature (T1_fast), the default value for salinity ([ ]), and a constant value for depth (15).

csd_matrix_odas

Estimate the cross- and auto-spectrum of one or two real matrices

Syntax

```
[Cxy, F, Cxx, Cyy] = csd_matrix_odas(x, y, nFFT, rate, window, overlap, msg)
```

Argument	Description
`x`	Matrix over which to estimate the cross-spectrum.
`y`	Matrix over which to estimate the cross-spectrum. If empty, only calculate the autospectra of x. Length must match 'x'.
`nFFT`	Length of x, y segments over which to calculate the ffts.
`rate`	Sampling rate of the vectors.
`window`	Window of length nFFT to place over each segment before calling the fft. If empty, window defaults to the cosine window and is normalized to have a mean-square of 1. The window is used as given and it is up to the user to make sure that the window preserves the variance of the signals.
`overlap`	Number of points to overlap between the successive segments used to calculate the fft. A good value is nFFT/2 which corresponds to 50%.
`msg`	Message string indicating how each segment is to be detrended before calling the fft. Options are 'constant', 'linear', 'parabolic', 'cubic', and 'none'. If empty, omitted, or 'none' then no detrending or mean removal is performed on the segments.
`Cxy`	Complex cross-spectrum matrix of x and y if x and y are different. Otherwise, the auto-spectrum matrix of x.
`F`	Frequency vector corresponding to the cross- and/or auto-spectrum. Ranges from 0 to the Nyquist frequency.
`Cxx`	Auto-spectrum matrix of x if x is different from y.
`Cyy`	Auto-spectrum matrix of y if x is different from y and y is not empty.

Description

Estimate the cross- and auto-spectrum of one or two matrices, where every column is a real vector, and the number of rows is the length of the vectors. For best results nFFT should be several times shorter than the lengths of `x` and `y` so that the ensemble average of the segments of length nFFT have some statistical significance. The cross-spectrum has the property that its integral from 0 to the Nyquist frequency equals the covariance of `x` and `y`,

if they are real variables. Use the function `csd_odas` if either `x` or `y`, or both, are complex vectors.

This function tests if `y` is empty, `y=[ ]`. If so, it computes only the auto-spectrum and places it into `Cxy`, and it will not return `Cxx` and `Cyy`, or it returns them empty. Thus, there is no need for a separate function to calculate auto-spectra. In addition, if `y` is not an empty matrix, then the two auto-spectra are optionally available and can be used to calculate the coherency spectrum and the transfer function of `y` relative to `x`. This function `csd_matrix_odas` is similar to the RSI function `csd_odas`, when `x` and `y` are columns vectors.

This function was motivated by the need to make the Goodman coeherent-noise reduction function, `clean_shear_spec`, more efficient. If you use `csd_odas` to build the auto- and cross-spectral matrices, then there are many redundant calls to the fft-function, and this redundancy increases with the number of columns in the input matracies. The function `csd_matrix_odas` does not make redundant fft calls.

csd_odas

Estimate the cross- and auto-spectrum of one or two vectors.

Syntax

```
[Cxy, F, Cxx, Cyy] = csd_odas(x, y, nFFT, rate, window, overlap, msg)
```

Argument	Description
`x`	Vector over which to estimate the cross-spectrum.
`y`	Vector over which to estimate the cross-spectrum. If empty or equal to x, only calculate the autospectra. Length must match 'x'.
`nFFT`	Length of x, y segments over which to calculate the ffts.
`rate`	Sampling rate of the vectors.
`window`	Window of length nFFT to place over each segment before calling the fft. If empty, window defaults to the cosine window and is normalized to have a mean-square of 1. The window is used as given and it is up to the user to make sure that the window preserves the variance of the signals.
`overlap`	Number of points to overlap between the successive segments used to calculate the fft. A good value is nFFT/2 which corresponds to 50%.
`msg`	Message string indicating how each segment is to be detrended before calling the fft. Options are 'constant', 'linear', 'parabolic', 'cubic', and 'none'. If empty, omitted, or 'none' then no detrending or mean removal is performed on the segments.
`Cxy`	Complex cross-spectrum of x and y if x and y are different. Otherwise, the auto-spectrum of x.
`F`	Frequency vector corresponding to the cross- and/or auto-spectrum. Ranges from 0 to the Nyquist frequency.
`Cxx`	Auto-spectrum of x if x is different from y.
`Cyy`	Auto-spectrum of y if x is different from y and y is not empty.

Description

Use this function only if the vectors are complex. Otherwise, use the function `csd_matrix_odas`. This function will become inactive and point to `csd_matrix_odas` once it can handle complex signals.

Estimate the cross- and auto-spectrum of one or two vectors. For best results nFFT should be several times shorter than the lengths of `x` and `y` so the ensemble average of the segments

of length nFFT has some statistical significance. The cross-spectrum has the property that its integral from 0 to the Nyquist frequency equals the covariance of `x` and `y`, if they are real variables. The vectors `x` and `y` can be complex.

This function tests if `x` is identical to `y` and if `y` is empty, `y=[ ]`. If so, it computes only the auto-spectrum and places it into `Cxy`, and it will not return `Cxx` and `Cyy`, or it returns them empty. It behaves likewise if `y` equals the empty matrix. Thus, there is no need for a separate function to calculate auto-spectra. In addition, if `x` and `y` are not the same vector, then the two auto-spectra are optionally available and can be used to calculate the coherency spectrum and the transfer function of `y` relative to `x`. This function `csd_odas` is very similar to the Matlab function `csd` that is no longer supported by MathWorks. We do not like the new approach used by MathWorks for spectral estimation because their new methods do not support the linear detrending of each segment before calling the fft. It only allows the detrending of the entire vectors `x` and `y`, which can leave undesirable low-frequency artifacts in the cross- and auto-spectrum. In addition, you do not need the Signal Processing Toolbox to use this function. The equation for the squared-coherency spectrum is:

$$G_{xy}(f) = \frac{|C_{xy}(f)|^2}{C_{xx}(f)C_{yy}(f)}$$

where G_{xy} is the squared coherency-spectrum, C_{xy} is the complex cross-spectrum, and C_{xx} and C_{yy} are the autospectra. The complex transfer function, H_{xy}, of y relative to x, is:

$$H_{xy}(f) = \frac{C_{xy}(f)}{C_{xx}(f)}$$

deconvolve

Deconvolve signal plus derivative to produce high resolution data

Syntax

```
xHires = deconvolve( name, X, X_dX, fs, diff_gain, ver )
```

Argument	Description
`name`	Channel name that contains the 'diff_gain' parameter. Can be left empty if 'diff_gain' is explicitly provided.
`X`	Low-resolution signal. Data without pre-emphasis is not required but will improve the initial accuracy of the output if available. This is most notable for high gain channels such as pressure.
`X_dX`	Pre-emphasized signals.
`fs`	Sampling rate of the data in Hz.
`diff_gain`	Differentiator gain. Provide either a numeric value, the configuration string, or the configuration object generated from parsing the configuration string.
`ver`	(Depreciated - not required) ODAS header version. Included for backward compatibility.
`xHires`	Deconvolved, high-resolution signal.

Description

Deconvolve a vector of pre-emphasized data (temperature, conductivity, or pressure) to yield high-resolution data. The pre-emphasized signal (x+gain*dx/dt) is low-pass filtered using appropriate initial conditions after the method described in Mudge and Lueck, 1994.

For pressure, you must pass both 'X' and 'X_dX' to this function. Both vectors are needed to make a good estimate of the initial conditions for filtering. Initial conditions are very important for pressure because the gain is usually ~ 20 s, and errors caused by a poor choice of initial conditions can last for several hundred seconds! In addition, the high-resolution pressure is linearly adjusted to match the low-resolution pressure so that the factory calibration coefficients can later be used to convert this signal into physical units.

The gains for all signal types are provided in the calibration report of your instrument.

Examples

```
>> C1_hres = deconvolve( 'C1_dC1', [], C1_dC1, fs_fast, setupfilestr )
```

Deconvolve the micro-conductivity channel using the diff_gain parameter from the 'C1_dC1' channel section found within the supplied configuration string. Because there is no channel without pre-emphasis, the argument is left empty.

```
>> T1_hres = deconvolve( '', T1, T1_dT1, fs_fast, 1.034 )
```

Deconvole the thermistor data using both channels with and without pre-emphasis. Using both channels improves the initial deconvolution accuracy and compensates for slight variations in the calibration coefficients between the two channels. Note the explicitly provided value for diff_gain. One typically provides the configuration string and channel name as shown in the previous example.

Mudge, T.D. and R.G. Lueck, 1994: Digital signal processing to enhance oceanographic observations, *J. Atmos. Oceanogr. Techn.*, 11, 825-836.

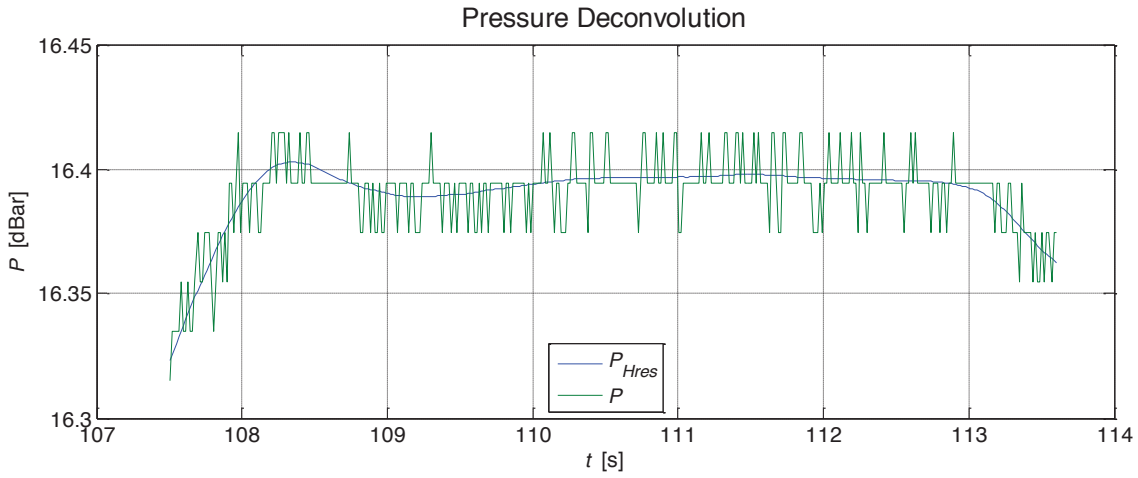

Figure 6.0.2: The green curve is from the normal pressure channel, and the blue curve is derived from the pre-emphasized pressure channel. This data is from a profiler that has impacted the soft bottom of a lake. Both signals are shown with full bandwidth (0 - 32 Hz) without any smoothing. The full-scale range of the pressure transducer is 500 dBar.

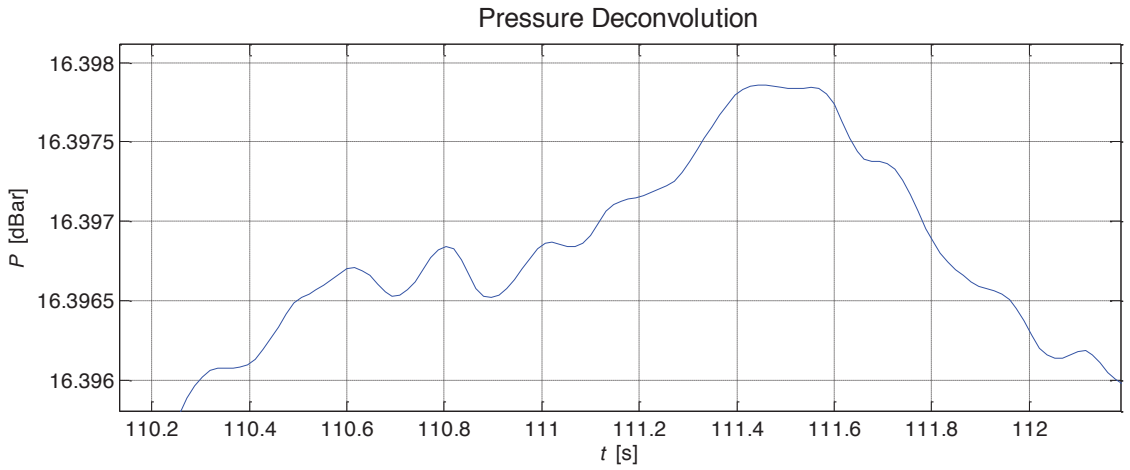

Figure 6.0.3: Same as previous figure but with zoom-in on only the high-resolution pressure. Again, full bandwidth without any smoothing.

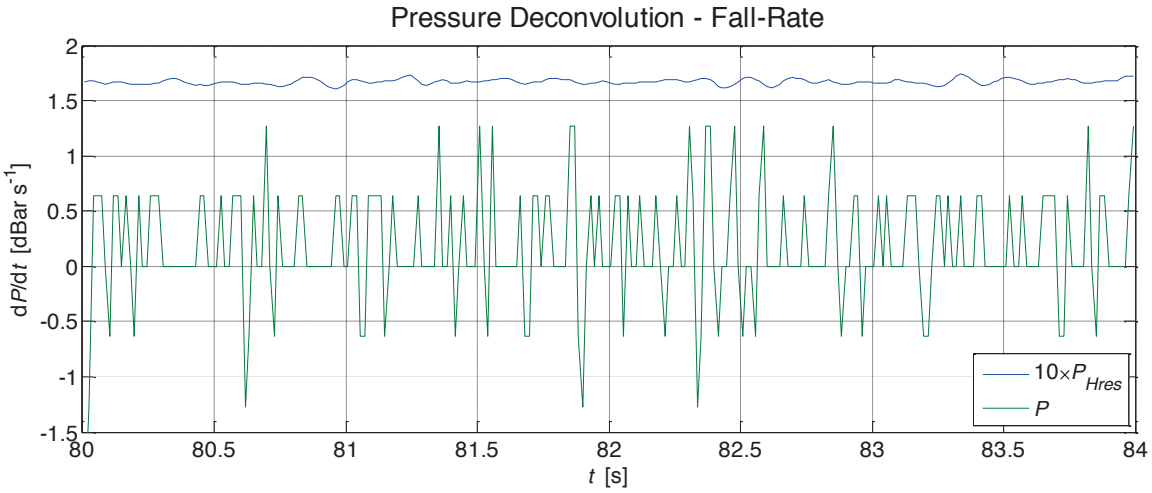

Figure 6.0.4: Full bandwidth signals of the rate of change of pressure. The normal pressure signal (green) produces a fall-rate that is useless without extensive smoothing because it is not even monotonic. The rate of change of the high-resolution pressure (blue) is smooth, always positive, and, therefore, the high-resolution pressure itself, can be used directly for plotting other variables as a function pressure (or depth). The high-resolution rate of change of pressure has been multiplied by 10 for visual clarity. The fall-rate is about $0.17\,\mathrm{m\,s^{-1}}$.

despike

Remove short-duration spikes from a signal.

Syntax

```
[y, spike, pass_count, ratio] = despike( dv, thresh, smooth, Fs, N, ...)
```

Argument	Description
`dv`	Signal to be despiked.
`thresh`	Threshold value for the ratio of the instantaneous rectified signal to its smoothed version. A value of 8 is a good starting point for a VMP.
`smooth`	The cut-off frequency of the first-order Butterworth filter that is used to smooth the rectified input signal. The time scale of smoothing is approximately 1/(2*smooth). A value of 0.5 is a good starting point for a VMP.
`Fs`	Sampling rate (Hz).
`N`	Spike removal scale. A total of 1.5*N data points are removed. N/2 points are removed before a spike, and N points are removed after a spike. The replaced data is an average of the adjacent neighbouthood. Averaging uses ~Fs/(4*smooth) points from each side of a spike.
`'-debug'`	Optional string that provides a plot of the data after every pass.
`'-single_pass'`	Optional string to force the function to make only one pass.
`y`	Despiked signal.
`spike`	Indices to the spikes.
`pass_count`	The number of iterations used to find and remove the spikes.
`ratio`	The fraction of the data replaced with a local mean.

Description

This function removes spikes in shear probe and micro-conductivity signals, such as those generated by a collision with plankton and other detritus. It identifies spikes by comparing the instantaneous rectified signal against its local psuedo standard deviation. The instantaneous rectified signal is obtained by:

- high-pass filtering the input signal, `dv`, at 0.5 Hz with a zero-phase, single-pole, Butterworth filter, and
- rectifying this signal by taking its absolute value.

The pseudo standard deviation is calculated by smoothing the rectified signal with a low-pass, single-pole, zero-phase, Butterworth filter with a cut-off frequency of `smooth` Hz.

The function applies itself iteratively. After identifying and replacing spikes, it tries again to identify and replace spikes, until no more spikes are detected. The maximun number of iterations is capped at 10.

You can restrict despiking to a single pass, which duplicates the behaviour of the function in previous versions, using the `'-single_pass'` string option. You can visualize the iterative operation of the function using the `'-debug'` string option.

Examples

```
>> [y,spike,pass_count,ratio] = despike(sh1,8,0.5,fs_fast,round(0.04*fs_fast))
```

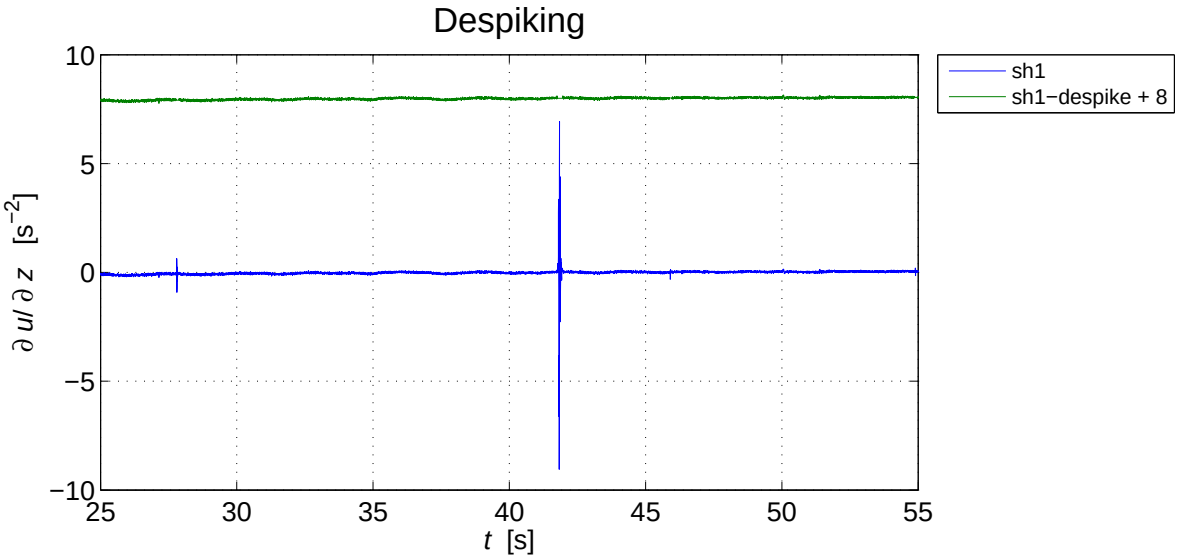

Figure 6.0.5: Despike function applied with despike(sh1, 7, 0.1, 512, 30). The smoothing scale, the inverse of 0.1 Hz, is long because this profiler was moving at only $0.17\,\mathrm{m\,s^{-1}}$. The units for shear should be $\mathrm{s^{-1}}$ and not $\mathrm{s^{-2}}$.

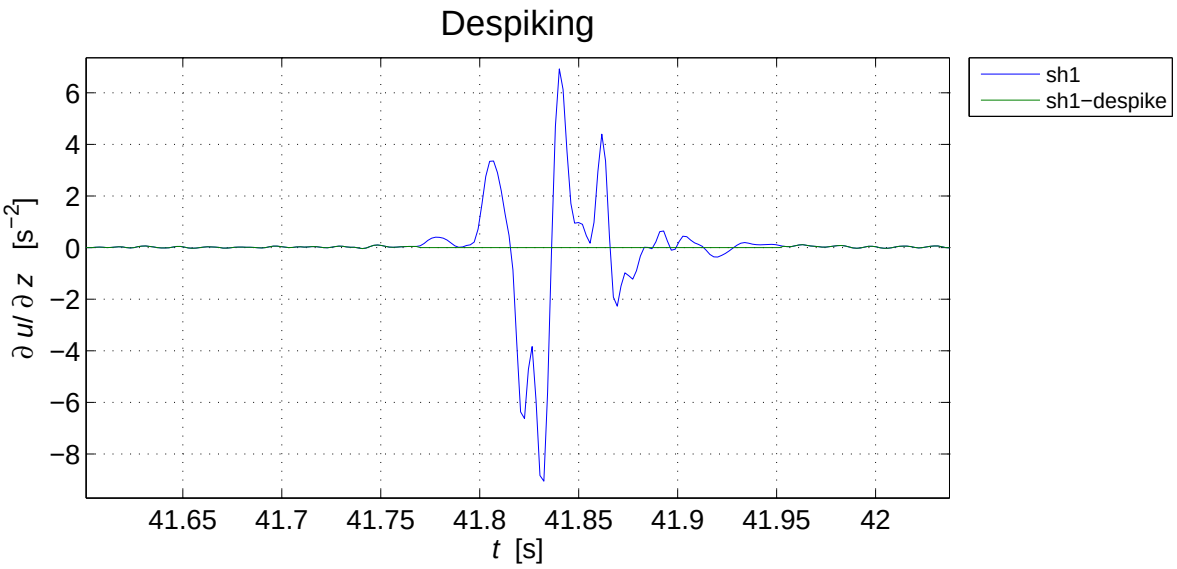

Figure 6.0.6: The same plot as the previous figure but with a close-up view.

The parameters for this function should be chosen with care and thoroughly tested using the `'-debug'` option.

The number of points around a spike that are removed is best expressed in terms of a duration because the ringing of the shear probe after a collision is not speed dependent. For example, `round(0.04*fs_fast)` will remove 40 ms of data after a spike and half that amount before a spike.

The parameter `smooth` is based on the notion that turbulence occurs over a path-length that exceeds some minimum, perhaps 0.5 m. Therefore, the smoothing time-scale and the cut-off frequency, `smooth`, of the filter are speed dependent. The value of `smooth` is proportional to speed.

The threshold should be chosen so that an extrama is indeed anomalous with respect to a neighbourhood defined by the parameter `smooth`. A value of 8 usually gives good result.

extract_odas

Extract a range of records from a RSI raw binary data file

Syntax

```
[newFile, successFlag] = extract_odas( fileName, startRecord, endRecord )
```

Argument	Description
`fileName`	Name of the binary data file, extension optional.
`startRecord`	Record number from which to start copying records.
`endRecord`	Record number of the last required record.
`newFile`	Name of the file created by this function. The first record contains the configuration string. This allows the extracted file to be processed like any other raw data file. All records from startRecord to endRecord are included in the new file.
`successFlag`	Integer representing status. 0 if operation successful, otherwise <= -1.

Description

Extract a range of records from a RSI raw binary data file and move them into a new file. The new file has the same name as the old file except that the range of the records is appended to the name. The original file is not altered.

Binary data files are sometimes overwhelmingly long. This function provides a means to segment a file into shorter pieces to speed up the data processing. The companion function 'show_P' can be used to extract the record-average pressure from a binary file which might be useful for determining the appropriate range(s) to be extracted.

This function can be used iteratively to segment a raw binary data file into an arbitrary number of smaller files. See also, `show_P.m`.

Examples

The following example shows how the large data file "DAT001.p" can be segmented to produce a smaller file. The user previously determined that the data file contained data of interest between records 5400 and 6900.

```
>> extract_odas( 'DAT001', 5400, 6900 )
ans = DAT001_5400_6900.p
```

The resulting file can be processed normally and, because it is smaller, will process faster.

Additional examples are in the top of the function itself.

extract_setupstr

Extract configuration string from a RSI raw binary data file.

Syntax

```
configStr = extract_setupstr( dataFileName, configFileName )
```

Argument	Description
`dataFileName`	Name of a version raw binary data file (version >= 6). (extension optional)
`configFileName`	Optional name of configuration file that should be generated. When excluded, no file is created but the configuration string is still returned.
`configStr`	Resulting configuration string.

Description

Extract the configuration string from a RSI raw binary data file. Use this function when one needs a copy of the configuration file used when the data file was created. Note that this function will only work on a v6 or greater data file.

If the `configFileName` is not provided and configStr not requested, this function will display the resulting configuration string. When `configFileName` is provided, a new configuration file named `configFileName` is created with the contents of the configuration string. When `configStr` is requested, the variable `configStr` is set to the configuration string.

Used in conjunction with `patch_setupstr`, this functions allows one to modify the calibration coefficients for the conversion of the data into physical units, and for other forms of data processing. First, extract the configuration string into a new file. Edit the file as required. Finally, use `patch_setupstr` to update the data file with the new configuration file.

Examples

```
>> configFile = extract_setupstr( 'data_005.p' )
```

Extract the configuration string from the data file `data_005.p` and store in the variable `configFile`.

```
>> extract_setupstr( 'data_005.p', 'setup.cfg' )
```

Extract the configuration string from `data_005.p` and store it in the file `setup.cfg`.

file_with_ext

Find files with default extensions

Syntax

```
[P,N,E,fullName] = file_with_ext( file, extensions, errorStr )
```

Argument	Description
`file`	Name of file with or without extension.
`extensions`	Cell array of acceptable extensions. ex, {'.p' '.P' ''}
`errorStr`	Error string used if error is to be triggered. (optional)
`P`	Full path to the requested file.
`N`	Name of requested file.
`E`	Extension of requested file.
`fullName`	Full path and file name with path seperators. (optional)

Description

Find files with default extensions. Convenience function that allows users to input file names without an extension.

This function searches for a file with the name `file` and an extension from the array `extensions`. When found, it returns the file components `[P,N,E]` along with the fully qualified name, `fullName`. If not found, the function triggers the error `errorStr`. Should `errorStr` be empty then no error is triggered and empty values are returned for `[P,N,E,fullName]`.

The return value `fullName` is provided to simplify addressing the resulting file. Alternatively, this value can be constructed from the `[P,N,E]` values.

The `extensions` array lists the acceptable file extensions for the requested file. Each extension should consist of a string value prepended with a period (`.`). To allow for the extension to be declared within the `file` variable, prepend the array with an empty string. See the example for details.

Be sure to include both possible case values within `extensions` - for example, `{'.p', '.P'}`. Some operating systems are case sensitive while others are not.

Examples

```
>> file_err = 'No valid input file was found..';
>> [P,N,E] = file_with_ext( file, {'' '.p' '.P'}, file_err );
```

Typical usage where `file` is a string provided by the user.

```
>> [P,N,E] = file_with_ext( file, {'' '.p' '.P'} );
>> if isempty(N), %processing error...;  end
```

Errors are optional. If `errorStr` is not provided, this function will not trigger an error if the file is not found.

```
>> [P,N,E,fName] = file_with_ext( fName,
                                  {'' '.p' '.P'},
                                  ['Unable to find file: ' fName] );
>> dos(['mv ' fName ' /dev/null']);
```

Record the full matching file identifier in `fName`. This example demonstrates how the function can be utilized in a single call. The subsequent `dos` command will effectively delete the file on a UNIX system. Using the full name and path ensures the wrong file does not get deleted.

fix_bad_buffers

Fix badly corrupted records in a RSI raw binary data file

Syntax

```
[badRecords, chop, truncate] = fix_bad_buffers( fileName )
```

Argument	Description
`fileName`	Full name of a RSI raw binary data file including the '.p' extension.
`badRecords`	Vector of the record numbers of bad records that were detected and fixed in the file 'fileName' after the fixing. The record numbers may shift if the program chops records off the front of the file.
`chop`	Number of bad records chopped off the front of the file. Set to -1 if the file can not be opened.
`truncate`	Number of bad records truncated from the file. Set to -1 if the file can not be opened.

Description

Repair bad records in RSI raw binay data files. It sometimes happens that there is a communication failure with an instrument. This is indicated by the 'Bad Buffer' message during data acquisition and its occurrence is flagged in the header of the affected record. The result is some skewed data in the binary file that can lead to huge errors if the skew is not fixed before processing the data.

When errors are detected, this function replaces the entire record with interpolated data. All significant data in the record will be lost but the resulting record can still be graphed without disrupting neighboring records. However, because there will be a time gap between a record with a bad buffer and the next record, the deconvolution function may produce erroneous results. The error will be transitory and decay with an e-folding time equal to the gain of the differentiator used to pre-emphasize a signal.

If a file ends with bad records, those records are truncated. If a file starts with bad records, those records are also truncated. If a bad record has already been fixed successfuly using `patch_odas`, the record is not changed.

IMPORTANT: Bad buffers should first be fixed using `patch_odas` because it preserves more of the original data. Should this fail, `fix_bad_buffers` can be used to patch the data file.

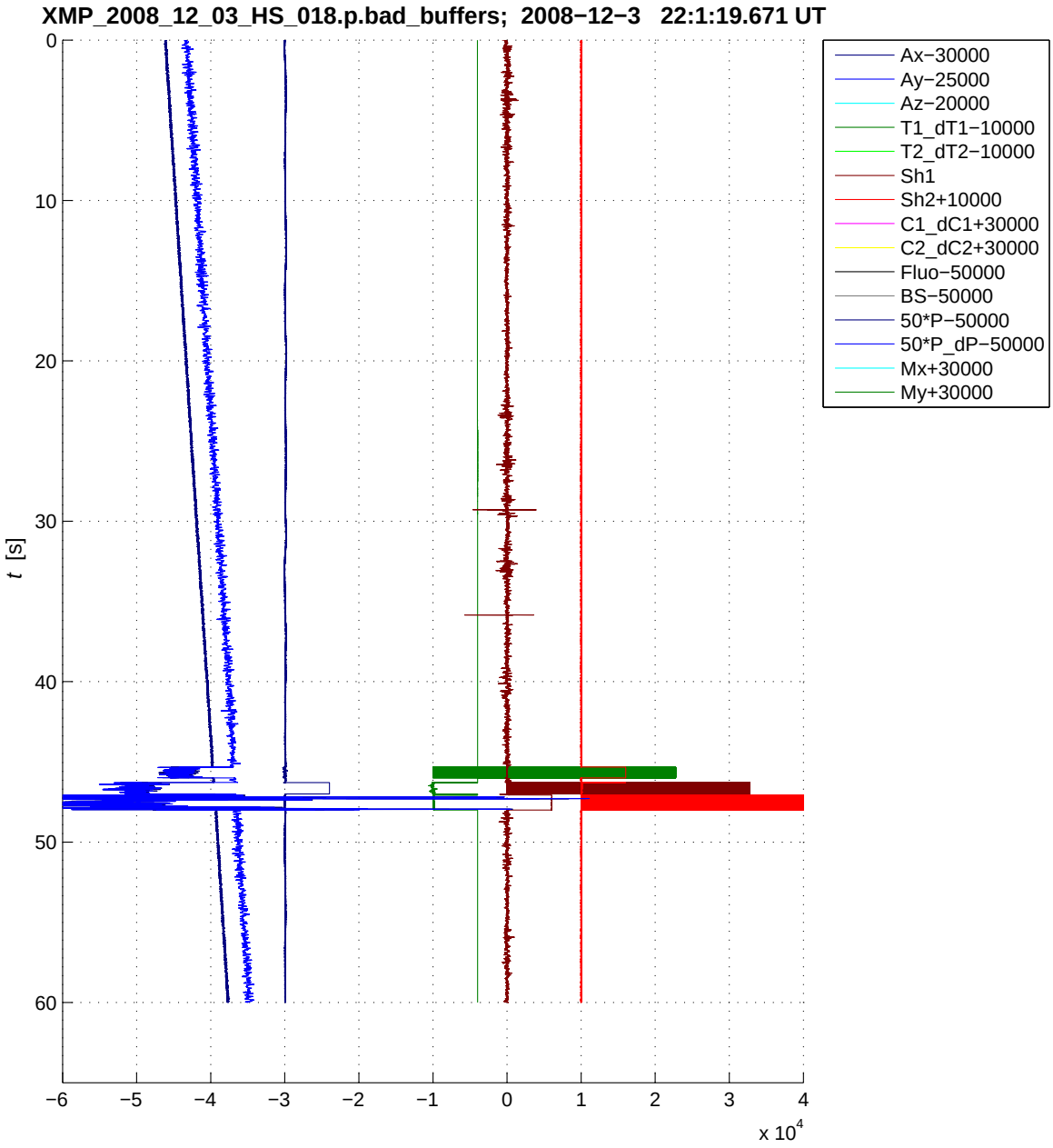

Figure 6.0.7: A corrupted data file that has 'Bad Buffers' from about 45 to 48 seconds. This sample is from an Expendable Microstructure Profiler (XMP) which was notorious for communication failuresn in the early stages of its development. Records 45 through 48 are skewed and appear garbled as a result of communication failures.

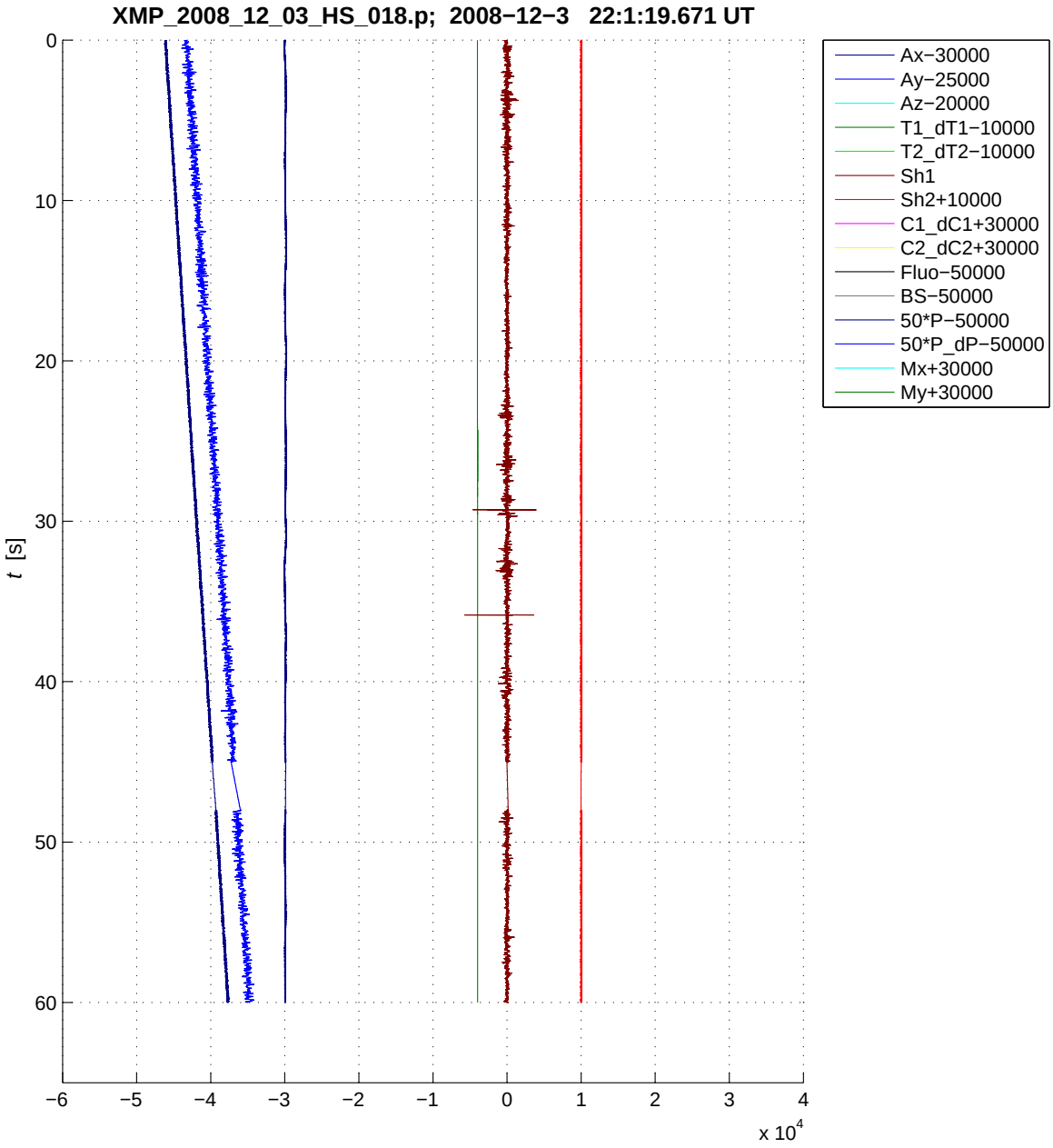

Figure 6.0.8: Same data as in the previous figure after being processed by `patch_odas.m` and then by `fix_bad_buffers.m`. Three records have been replaced with interpolated data. Interpolated data should be used cautiously.

This function will make a backup copy, but the user is advised to always make a backup of the data file in a secure directory before using this function. This function should be called **after** `patch_odas` because less data is lost if `patch_odas` is run first.

fix_underscore

Legacy function replaced with texstr

Description

Depreciated function. Please use `texstr` in place of this function.

fopen_odas

Open a RSI raw binary data file of unknown endian type

Syntax

```
[fid, errorMsg] = fopen_odas(fileName, permission)
```

Argument	Description
`fileName`	Name of the requested RSI raw data file.
`permission`	Permission string, usually 'r', or 'r+'. Consult Matlab documentation for the command 'fopen'.
`fid`	File pointer to 'fileName'. Values less then 3 indicate errors. For error values, consult Matlab documentation for the command 'fopen'.
`errorMsg`	String indicating the endian flag error. Empty if file failed to open or no error was observed.

Description

Used to open a RSI raw binary data file of unknown endian type. This function observes the endian flag in the header to determin the correct endian format of the file.

When the endian flag matches the endian format deduced by reading the file, the file is opened without returning any error messages.

When the endian flag is empty (equal to 0), we assume the file is in 'little' endian format. We open the file and return the appropriate error message.

Should the endian flag indicate the opposite endian value as that deduced by reading the header, the file is opened using the deduced value. An error message is returned indicating the error.

All other values of the endian flag are considered invalid and result in the file not being opened and the appropriate error message being returned.

Acceptable endian flags in the header are as follows:

- 0 = unknown
- 1 = little endian, the usual value with Intel and compatible processors
- 2 = big endian, the usual value with ODAS5-IR

Possible *error_msg* Values:

- 0: endian flag not found, assume 'little'.
- 1: endian flag does not match file - ignore flag and assume 'little'.
- 2: endian flag does not match file - ignore flag and assume 'big'.
- 3: endian flag invalid - byte offsets (127,128) = `byte127, byte128`.

Error string 0 occurs when the endian flag is zero. When this happens we assume the file to be in 'little' endian format, open the file, and report the error.

Error strings 1 and 2 occur when the endian flag in the header does not agree with the observed endian format of the file. In these scenarios we open the file with the observed endian format and report the error.

Error string 3 occurs when the endian flag is invalid. This is most likely the result of a corrupted or invalid data file. The two bytes representing the endian flag are displayed. The file is closed.

get_diss_odas

Calculate a profile of dissipation rate.

Syntax

```
diss = get_diss_odas( shear, A, info )
```

Argument	Description
`shear`	Matrix of shear probe signals, in physical units, one shear probe per column.
`A`	Matrix of acceleration signals, one acceleromter per column. Does not have to be in physical units. Number of rows must be the same in shear and A.
`info`	Structure containing additional process-control parameters.
`diss`	Structure of results including the profile of the rate of dissipation of turbulent kinetic energy, for each probe, and related information.

Description

This function calculates a profile of the rate of dissipation of turbulent kinetic energy for each shear probe using the method described in RSI Technical Note 028. The shear data should be despiked (using the function despike), and high-pass filtered at a frequency [Hz] that is about one-half of the inverse of the length [s] of the fft-segments. There is no limitation on the number of shear-probe and accelerometer signals. All results are returned in a structure.

Input Structure

Argument	Description
fft_length	length of the FFT segment [samples]. ex, 1024
diss_length	length of shear data over which to estimate the rate of dissipation [samples]. Must be >= 2 * fft_length, recommended >= 3 * fft_length.
overlap	Distance [samples] of the overlap of successive dissipation estimates. Should be 0 <= overlap <= diss_length. Suggested value: overlap = diss_length/2. overlap = 0 means no overlap.
fs_fast	Sampling rate in Hz for shear and acceleration.
fs_slow	Sampling rate in Hz for all slow channels in Data_slow.
t_fast	Time vector for fast channels.
speed	Profiling speed used to derive wavenumber spectra. Must have length equal to 1 or the number of rows in shear.
T	Temperature vector used when calculating kinematic viscosity. Length must match shear.
P	Pressure data. Must have same length as the shear-probe data.

Optional Input Parameters

Argument	Description
Data_fast	Data matrix of fast channels, one per column. These will be averaged over the time interval of each dissipation estimate. Must have the same number of rows as the shear matrix. Default value = [ ].
Data_slow	Same as Data_fast but assumes slow channel data. Each column should have length = size(shear,1)*fs_slow/fs_fast. Default value = [ ].
fit_order	Order of the polynomial fit to the shear spectra, in log-log space, that is used to estimate the spectral-minimum wavenumber. Default = 3.
f_AA	Cut-off frequency of the anti-aliasing filter. Default = 98 Hz.
fit_2_isr	Value of dissipation rate at which the estimation will be derived from a fit to the inertial subrange rather than by spectral integration. Default fit_2_isr = 1.5e-5 W/kg.
f_limit	Maximum frequency to use when estimating the rate of dissipation. Use this parameter if you have vibrational contamination that is not removed by the Goddman coherent noise removal function (clean_shear_spectrum). Default value is f_limit=inf.
UVW	Matrix where every column is a velocity component. Must have the same number of rows as the shear matrix. It will be used to estimate the maximum angle of attack of the flow for each dissipation estimate. Default = [ ].

Output Structure

A structure of the rate of dissipation estimated from the shear probes, along with a lot of other information. The elements are:

Argument	Description
`e`	Rate of dissipation [W/kg]. One row for every shear probe and one column for every dissipation estimate.
`K`	Wavenumbers [cpm, or cycles per metre] with one column for every dissipation estimate. The number of rows equals the number of wavenumber elements in the cross-spectra.
`K_max`	A matrix of the maximum wavenumber used for each dissipation estimate. Its size is [N P] where N is the number of shear probes and P is the number of dissipation estimates.
`sh_clean`	Wavenumber spectrum for each shear probe signal at each dissipation estimate. This is a 4-dimensional matrix of size [M N N P] where N is the number of shear probes and P is the number of dissipation estimates. M is the number of wavenumber elements in each cross-spectrum. The "diagonal" elements of the N by N submatrix are the auto-spectra.
`sh`	same as sh_clean but without cleaning by the Goodman coherent noise removal algorithm.
`AA`	the cross-spectral matrix of acceleration signals for each dissipation estimate. This is a 4-dimensional matrix of size [M N N P] where N is the number of acceleration signals and P is the number of dissipation estimates. M is the number of wavenumber elements in each cross-spectrum. The "diagonal" elements of the N by N submatrix are the auto-spectra.
`UA`	the cross-spectral matrix of shear probe and acceleration signals. Its size is [M N_s N_a P] where N_s is the number of shear probes and N_a is the number of acceleration signals.
`Nasmyth`	a 3-dimensional matrix of the Nasmyth spectra for each shear probe and each dissipation estimate. Its size is [M N P] where M is equal to the number of rows in diss.K and each wavenumber element corresponds to diss.K, N is the number of shear probes and P is the number of dissipation estimates.
`F`	frequencies, with one column for every dissipation estimate. Currently every column is identical.
`speed`	a column vector of the mean speed at each dissipation estimate.
`nu`	column vector of kinematic viscosity [m^2/s] at each dissipation estimate.

Argument	Description
P	column vector of pressure at each dissipation estimate. Derived directly from the input.
t	column vector of time at each dissipation estimate. Derived directly from the input.
T	column vector of temperature at each dissipation estimate. Derived directly from the input.
Data_fast	a matrix where every column is a fast vector that has been averaged over the interval of each dissaption estimate. The number of rows equals the length of diss.P. This matrix can be empty.
Data_slow	same as diss.Data_fast except that it is for the slow vectors.
method	a matrix indicating the method used to make each dissipation estimate. A value of 0 indicates the variance method and a value of 1 indicates the inertial subrange fitting method. Its size [N P] where N is the number of shear probes and P is the number of dissipation estimates.
DOF	a matrix of the degrees of freedom in each dissipation estimate. Its size [N P] where N is the number of shear probes and P is the number of estimates.
MAD	A matrix of the mean absolute deviation of the base-10 logarithm of the spectral values from the Nasmyth spectrum over the wavenumber range used to make each dissipation estimate. DOF and MAD are used for quality control of the dissipation estimates.
AOA	a vector containing the maximum angle of attack of the velocity field, if you passed the 3 velocity components U V W. The output is empty if you did not pass the velocity vectors.

get_latest_file

Get the name of the latest RSI raw binary data file in current folder.

Syntax

```
latest_file = get_latest_file( match_file )
```

Argument	Description
`match_file`	Expression to filter matching files. Defaults to "*" for accepting all files.
`latest_file`	Name of the latest RSI raw binary data file in the current directory. Empty if no ODAS binary data files exist.

Description

Retrieve the name of the latest RSI raw binary data file in the local directory. It is useful for near real-time data processing and plotting because such operations are frequently conducted on the latest data file recorded in the local directory.

get_profile

Extract indices to where an instrument is moving up or down

Syntax

```
profile = get_profile( P, W, pMin, wMin, direction, minDuration, fs )
```

Argument	Description
`P`	Pressure vector
`W`	Vector of the rate-of-change of pressure.
`pMin`	Minimum pressure, for a valid profile.
`wMin`	Minimum magnitude of the rate-of-change of pressure, for a valid profile.
`direction`	Direction of profile, either 'up' or 'down'.
`minDuration`	Minimum duration, for a valid profile [s].
`fs`	Sampling rate of P and W in samples per second.
`profile`	2 X N matrix where each column contains the start- and end-indices of a profile, according to the input definition. Empty if no profiles were detected.

Description

Extract the sample indices to the sections in a data file where the instrument is steadily moving in `direction` (up or down) for at least `minDuration` seconds, faster than `wMin` and at a pressure greater than `pMin`.

Call this function separately for ascents and descents.

Can be used with data files collected with any vertical profiler or glider.

get_scalar_spectra_odas

Calculate spectra of scalar signals for an entire profile.

Syntax

```
sp = get_scalar_spectra_odas( scalar_vectors, P, t, speed, ... )
```

Argument	Description
scalar_vectors	Matrix in which each column is a scalar vector.
P	Column vector of pressure that will be used to calculate the average pressure of the data used for each spectrum. Its length must match the number of rows in scalar_vectors.
t	Column vector of time that will be used to calculate the average time of the data used for each spectrum. Its length must match the number of rows in scalar_vectors.
speed	Scalar or vector of profiling speed that is used to derive wavenumber spectra. If it is a vector, it must have a length equal to the number of rows in scalar_vectors.
...	Structure, or parameter / value pair, of optional parameters.
sp	Structure containing resulting spectra and anciliary information. Exact contents of the structure are discussed within the description section.

Description

Computes the spectra of scalar signals such as the gradients of temperature and micro-conductivity. It is usually used in conjunction with get_diss_odas, which returns the spectra of shear signals.

Optional Input Parameters

Argument	Description
`diff_gain`	Vector of differentiator gains used to pre-emphasize the signals. It is used to make a small correction to the spectra due to the frequency characteristics of the deconvolve function. Must be empty or its length must equal the number of columns in scalar_vectors, if gradient_method = 'first_difference'. Default = [ ].
`gadient_method`	Method used to create the scalar gradient vectors. 'first_difference' corrects for the deconvolution and the first difference operator used to estimate the scalar gradient. 'high_pass' applies no correction. Default = 'first_difference'
`fft_length`	FFT segment length [samples] used to estimate the auto-spectrum of scalar_vectors. Default fft_length = 512.
`spec_length`	Length of data used for each spectrum. It should be at least 2 times fft_length. The recommended value is larger than 2 times fft_length, for statistical reliability. Default = 3*fft_length.
`overlap`	Number of samples by which the auto-spectral estimates overlap. Should be between 0 and spec_length. The recommended value is overlap = spec_length/2, or the value used for the function get_diss_odas.
`fs`	Sampling rate in Hz. Default value = 512.
`f_AA`	Cut-off frequency of the anti-aliasing filter. Default = 98 Hz.

Resulting Output Structure

Argument	Description
K	Wavenumbers [cpm, or cycles per metre]. Contains one column for every spectral estimate.
F	Frequencies with one column for every spectral estimate. Every column is identical.
scalar_spec	Wavenumber spectrum for each scalar vector. This is a 3 dimensional matrix of size [M N L]. Every column is a spectrum – one for each signal. Thus, N equals the number of signals. Each layer (the third index, L) corresponds to a spectral estimate. The number of layers, L, is determined by the values of spec_length, and overlap, and the length of the signals. The number of rows, M, is the number of wavenumber values (from 0 to the Nyquist wavenumber). Usually, M = 1 + fft_length/2. M equals the number of rows in K.
speed	Column vector of the mean speed for each spectral estimate. Derived directly from the input.
P	Column vector of the mean pressure for each spectral estimate. Derived directly from the input.

hotelfile_Nortek_vector

Generate a hotel-file from Nortek Vector dat- and sen-files

Syntax

```
dat = hotelfile_Nortek_vector(file_name, sampling_rate )
```

Argument	Description
file_name	Name of data file to read.
sampling_rate	The rate of sampling. If absent, the *.hdr file in the local directory is used to determine the sampling rate.
dat	Optional. When specified, the vector data are return in this structure. Otherwise, the data are written into a hotel file.

Description

Generate a hotel-file containing the data from a Nortek Vector ADV `*.dat` and `*.sen` files. If a `*.hdr` file exists, it will be used to determine the sampling rate of the data in the `*.dat` file.

The hotel file is a mat-file with the same base name as the data files. The hotel file can be used with `odas_p2mat` and `quick_look` to incoprporate the ADV data into an RSI data mat-file.

Notes

- Nortek Vector instruments with outdated firmware can generate data files containing "(*null*)" entries. This function can not process such files. Replacing all "(*null*)" entries with "0" solves this problem. Ensure the Nortek Vector firmware is properly updated to prevent this problem from occuring.

hotelfile_Remus_mat

Generate hotel file from Remus MAT files

Syntax

`User Script: hotelfile_Remus_mat`

Argument	Description
`file_num_range`	Scan input files within the specified range.
`file_base_name`	Base name of the input files.
`output_file_name`	Hotel file created with the specified name. Default value generates a name derrived from the input file names.

Description

Generate a hotel-file from the Matlab mission-files of a Remus AUV that were generated by Hydroid-supplied software. The resulting hotel file contains the subset of data from the

mission-files that are useful for converting a RSI data file into physical units and for other purposes.

The speed information in the hotel-file is used by `odas_p2mat` to convert data into physical units. All vectors that are extracted from the mission-file are interpolated to the RSI `t_fast` and `t_slow` time vectors. The mission data vectors are renamed using `_fast` and `_slow` postfixes. A direct comparison of mission-file data and RSI recorded data is then possible.

Examples

The hotel-file can be generated by navigating to the directory containing the mission mat-files. This script is then executed.

```
>> hotelfile_Remus_mat
```

The resulting hotel-file can then be used with odas_p2mat. In this example, a RSI raw binary data file ending in `16` is used.

```
>> odas_p2mat('*16', 'vehicle', 'auv', ...
              'hotel_file', 'my_hotel_file_name.mat');
```

If there is a time offset between the RSI data and the data supplied within in the mission-file, a time offset can be added to the RSI instrument time.

```
>> odas_p2mat('*16', 'vehicle', 'auv', ...
              'hotel_file', 'my_hotel_file_name.mat', ...
              'time_offset', -5);
```

This is a script that you must edit. The parameters `file_num_range`, `file_base_name` and `output_file_name` are near the top of the function. The names of the desired data in the mission-file, and the names that they will attain in your RSI data mat-file are tabulated just below the parameter names.

hotelfile_seaglider_netcdf

Generate a hotel-file from the NetCDF files generated by a Seaglider

Syntax

`User Script: hotelfile_seaglider_netcdf`

Argument	Description
`file_num_range`	Scan input NetCDF mission-files within the specified range.
`file_base_name`	Base name of the input NetCDF mission-files.
`output_file_name`	Hotel-file created with the specified name. The default name is derived from the input-file names.

Description

Generate a hotel-file from the NetCDF mission-files of a Sea-glider that were generated by Kongsberg-supplied software.

The resulting hotel-file contains the subset of data from the NetCDF mission-files that are useful for converting RSI binary data files into physical units and for other purposes. All of the vectors that are extracted from the mission-files are interpolated to the RSI `t_fast` and `t_slow` time vectors. The mission data vectors are renamed using the `_fast` and `_slow` postfixes. A direct comparison of mission-file data and RSI recorded data is then possible.

Examples

The hotel-file can be generated by navigating to the directory containing the NetCDF files. This script is then executed.

```
>> hotelfile_seaglider_netcdf
```

The resulting hotel-file can then be used with `odas_p2mat`. In this example, a RSI raw binary data file ending in `16` is used.

```
>> odas_p2mat('*16', 'vehicle', 'sea_glider', ...
              'hotel_file', 'my_hotel_file_name.mat');
```

If there is a time offset between the RSI data and the data supplied within in the mission-file, a time offset can be added to the RSI instrument time.

```
>> odas_p2mat('*16', 'vehicle', 'sea_glider', ...
              'hotel_file', 'my_hotel_file_name.mat', ...
              'time_offset', -5);
```

This is a script that you must edit. The parameters `file_num_range`, `file_base_name` and `output_file_name` are near the top of the function. The names of the desired data in the mission-file, and the names that they will attain in your RSI data mat-file are tabulated just below the parameter names.

hotelfile_slocum_netcdf

Generate a hotel-file from NetCDF files generated by a Slocum Glider

Syntax

`User Script: hotelfile_slocum_netcdf`

Argument	Description
`depl_date`	Optional deployment date. Leave empty for all files.
`glider`	Glider name. If left empty the name will be extracted from the NetCDF files.
`hotelfile_date_glider.mat`	Script generated hotel-file with unique name constructed from the NetCDF file names. The user can, optionally, force the name to a desired value.

Description

Generate a hotel-file from the NetCDF mission-files of a Webb Slocum glider that were generated by Webb-supplied software.

The resulting hotel-file contains the subset of data from the NetCDF mission-files that may be used for a variety of purposes directly from within a RSI data mat-file. All of the vectors that are extracted from the mission-files are interpolated on to the RSI `t_fast` and `t_slow` time vectors. The mission data vectors are renamed using `_fast` and `_slow` postfixes. A direct comparison of mission-file data and RSI recorded data is then possible.

Examples

The hotel-file can be generated by navigating to the directory directly above the directory containing the NetCDF files. This script is then executed.

```
>> hotelfile_slocum_netcdf
```

The resulting-hotel file can then be used with `odas_p2mat`. In this example, a RSI raw binary data file ending in `16` is used.

```
>> odas_p2mat('*16', 'vehicle', 'slocum_glider', ...
              'hotel_file', 'hotelfile_08-18-2015_bob_mr_test.mat');
```

If there is a time offset between the RSI data and the data supplied within in the mission-file, a time offset can be added to the RSI instrument time.

```
>> odas_p2mat('*16', 'vehicle', 'slocum_glider', ...
              'hotel_file', 'hotelfile_08-18-2015_bob_mr_test.mat', ...
              'time_offset', -5);
```

This is a script that you must edit. The parameters for the directory holding the NetCDF files, the deployment date, the glider name and the output file name, are near the top of the function. The names of the desired data in the mission-file, and the names that they will attain in your RSI data mat-file are tabulated just below the parameter names.

Using this script is entirely optional for a MicroRider mounted on a Slocum glider because the pressure record, pitch angle (`-Incl_Y`) and the angle-of-attack (`aoa`) will be automatically used to determine the speed of profiling. That is all that is needed to convert your data into physical units. This function is necessary if you want to compare mission data, such as pressure, pitch and roll against the same data recorded by the MicroRider. Also, if you have an uncalibrated FP07 thermistor, then the CTD data in the mission-file can be used to calibrate the FP07 using in-situ data (see the function `cal_FP07_in_situ`).

Credit

- Anatoli Erofeev, Oregon State University. The code used in this script is inspired by the scripts and functions developed by Anatoli.

make_scientific

Convert number into a string in scientific notation

Syntax

```
scString = make_scientific( N, Digits )
```

Argument	Description
`N`	Number to convert to scientific notation
`Digits`	Number of significant digits required in the output
`scString`	String representation of 'N' in scientific notation. When 'N' is a vector of length greater than 1, 'scString' is a cell array of strings.

Description

Converts the number `N` into a string representation in scientific notation. The resulting string is optimized for use in Matlab plotting. When `N` is a matrix, each value is converted, and returned in a cell array of strings.

Examples

```
>> plotLabel = make_scientific( pi/100, 5 )
```

Returns Pi/100 with 5 significant digits:

```
3.1416 \times 10^{-2}
```

Rendered in a MATLAB plot as:

3.1416×10^{-2}

median_filter

Used for removing flyers from ADV data

Syntax

```
[result, points] = median_filter( dIn, threshold, filterLen,
                                  extraPoints, stDev, k )
```

Argument	Description
`dIn`	Matrix of column vectors to clean. Typically three columns – one for each velocity component.
`threshold`	Data points are defined as bad when they differ by more than this value from the median. Can be empty if using the stDev option.
`filterLen`	Length of data segment [samples] used to calculate the median and standard deviation.
`extraPoints`	Number of points, adjacent to bad points, to also replace with interpolated values.
`stDev`	empty string '', or 'std_dev'.
`k`	empty, or the scale factor for the stDev option.
`result`	Copy of dIn with bad data points replaced with interpolated values.
`points`	Vector of indicies to the bad data samples. Length of points equals the number of bad samples detected.

Description

Use a median filter to detect and remove flyers or outliers - points that are obviously wrong because they differ greatly from a local median.

This function is directed at the cleaning of ADV (acoustic doppler velocimeter) data using only velocity data without any information about correlation coefficients or other metrics. Can be used with other data.

The function detects erroneous data by calculating the median and standard deviation of segments of length `filterLen`. Data points that deviate by more then `threshold` from the median are flagged as bad. The value of `threshold` can be specified explictly, in which case the values of `stDev` and `k` are ignored. If `threshold` is empty, and `stDev = 'st_dev'`, the threshold is set to `k` times the standard deviation. That is, the threshold may change from segement to segment. The standard deviation of a segment may be highly elevated due to a

flyer, and it may be prudent to clean the data using two passes of this function. First with a very high value of `k`, say ~ 8, to remove extreme flyers, and then with a lower value of ~ 4.

When dIn has multiple columns, a bad data point in one column invalidates the corresponding data points in all other columns. This behaviour is suited to cleaning ADV data, for which an error in one velocity component implies an error in the other components. If this behaviour is undesirable, use the function with only a single column vector.

The input data may be from the analog output voltage of an ADV, in which case it is frequently oversampled. A short neighbourhood adjacent to the bad points may then also be contaminated by a flyer. Use the `extraPoints` option to remove such points. A value of 3 seems to work well for analog ADV data sampled at 512 Hz, when the ADV itself was updating its output at a rate of $4\,\mathrm{s}^{-1}$.

nasmyth

Generate a Nasmyth universal shear spectrum

Syntax

```
[phi, varargout] = nasmyth( varargin )
```

Argument	Description
`e`	Rate of dissipation in units of watts per kg.
`nu`	Kinematic viscosity in units of metre-squared per second. Default value is 1e-6.
`N`	Number of spectral points. Default value is 1000.
`k`	Wavenumber in cpm.
`phi`	Nasmyth spectrum in units of per seconds-squared per cpm. The length of phi is N, or the length of k.
`k`	Wavenumber in cpm.

Description

This function generates a Nasmyth universal shear spectrum. There are four basic forms of this function:

```
1) [phi,k] = nasmyth( e, nu, N);
2)  phi    = nasmyth( e, nu, k);
3) [phi,k] = nasmyth( 0, N );
4)  phi    = nasmyth( 0, k );
```

Form 1: Return the Nasmyth spectrum for the dissipation rate `e` and viscosity `nu`. The length of the returned spectrum `phi` is `N` and `k` is the wavenumber in cpm. Default values are `nu` $= 1 \times 10^{-6}$ and `N` $= 1000$.

Form 2: Same as form 1) except that the Nasmyth spectrum is evaluated at the wavenumbers given in the input vector `k` [in cpm]. `k` must be a vector.

Form 3: Return the non-dimensional Nasmyth spectrum (the G2 function in Oakey, 1982) of length `N` points. The wavenumber is $k = k'/ks$ [where k' is in cpm, $k_s = (\epsilon/\nu^3)^{1/4}$ (see Oakey 1982)] and runs from 1×10^{-4} to 1.

Form 4: Same as form 3) except that the non-dimensional spectrum is evaluated at the wavenumbers given in the input vector `k` (in k'/k_s).

Note

For forms 1) and 2), the dissipation rate can be a vector, e.g. `e` = [1e-7 1e-6 1e-5], in which case `phi` is a matrix whose columns contain the dimensional Nasmyth spectra for the dissipation rates specified in `e`.

The form of the spectrum is computed from a fit by Lueck that is documented in McMillan, et al (2016) and is based on the Nasmyth points listed by Oakey (1982).

Examples

Form 1:

```
>> [phi,k] = nasmyth( 1e-7, 1.2e-6, 512 )
>> [phi,k] = nasmyth( 1e-7, 1.2e-6 )
>> [phi,k] = nasmyth( 1e-7 )
```

Form 2:

```
>> phi = nasmyth( 1e-7, 1.2e-6, logspace(-1,3,512) )
```

Form 4:

```
>> phi = nasmyth( 0, logspace(-3,0,512) )
```

References:

- Oakey, N. S., 1982: J. Phys. Ocean., 12, 256-271.

- McMillan, J.M., A.E. Hay, R.G. Lueck and F. Wolk, 2016: Rates of dissipation of turbulent kinetic energy in a high Reynolds Number tidal channel. J. Atmos. and Oceanic. Techno., 33, TBD.

odas_p2mat

Convert a RSI raw binary data file into a Matlab mat-file

Syntax

```
result = odas_p2mat( fname, ... )
```

Argument	Description
`fname`	Name of the raw binary RSI data file to process (extension optional).
`...`	Optional parameters provided as a structure and-or list of parameter name-value pairs. See below for a listing of optional parameters.
`result`	Return structure with different contents depending on the context in which the function is called. When odas_p2mat is called with no input arguments, the result structure contains a set of default input arguments and no processing is performed. When odas_p2mat has fname specified, the structure contains the results traditionally saved within the resulting mat-file and no mat-file is generated. The mat-file is only generated if result is not explicitly used.

Description

Loading, converting, and processing of raw data files (`.p`) is performed by the `odas_p2mat` function. This function works with all RSI data files and automates much of the work involved in processing data files.

This function reads and processes a raw binary `p`-file to generate data vectors converted into physical units. Data vectors are either saved within a `mat`-file named from the `p`-file or returned within the `result` structure.

Optional input parameters control the data-conversion process. Each parameter has a default value that is used if the parameter is not explictly specified. To simplify using these parameters, a structure containing all optional parameters, set to their default values, is returned when this function is called without input arguments. For example,

```
>> default_parameters = odas_p2mat()

default_parameters =
      [MF_extra_points] 0
                  [MF_k] 4
              [MF_k_mag] 1.7
                [MF_len] 256
             [MF_st_dev] 'st_dev'
          [MF_threshold] []
                    [aoa] []
         [constant_speed] []
          [constant_temp] []
             [hotel_file] []
           [speed_cutout] 0.05
              [speed_tau] []
            [time_offset] 0
                [vehicle] ''
```

You can then change the parameters (which are the fields within the returned structure `default_parameters`) to values suited to your particular processing requirements.

Vehicle Specification

An instrument is identified by the `vehicle` parameter.

Argument	Description
`vehicle`	String identifying the vehicle that carries the RSI instrument. Typically empty [ ].

The vehicle should be identified in the `[instrument_info]` section of the configuration-file. The function searches the configuration string and tries to identify the vehicle. If it cannot find the vehicle, it will read this parameter from the input arguments. If the input parameter is empty, it will assume that the vehicle is `vmp`. You must specify this parameter only if it is not in the configuration-string (and the data is not from a VMP), or if you want to explicitly override the value in the configuration string. This is bad practice but useful for testing purposes. Ultimately, you should correct the configuration string using the functions `extract_setupstr` and `patch_setupstr`.

The recognized `vehicle`-types are listed in the `default_vehicle_attributes.ini` file included in the ODAS Matlab Library.

Profiling Speed

Parameters used to determine the speed of profiling speed are;

Argument	Description
constant_speed	Speed used to generate gradients and convert shear probe data into physical units. Leave empty if a constant speed is not desired. Must be empty or positive. Default = [ ]. Units are m/s.
hotel_file	Name of the mat-file containing speed information from a source outside of the raw RSI data file. Default = [ ]
speed_tau	Time-scale [s] of smoothing filter applied to speed data. Default is vehicle dependent. See the default_vehicle_attributes.ini file.
speed_cutout	Minimum profiling speed, [m/s], used to convert data into physical units. Slower speeds are set to this value. Default = 0.05.

The constant_speed parameter takes precedence over all other methods of deriving the speed of profiling. Specifying a constant_speed is useful if you have unreliable speed data, for example, when profiling through large turbulent up- and down-drafts. Or, if you have a vehicle for which it is not possible to calculate the speed using the raw RSI data file, and you do not yet have a hotel-file.

If constant_speed is empty, the speed of profiling is calculated from data in the RSI data file, or optionally, from data in a hotel-file. The algorithm used to calculate speed is determined by the vehicle parameter (see above). If the speed algorithm requires data vectors not available in the current data file, the required data vectors must be provided within a hotel-file. You can also use a hotel-file if you prefer an alternative method of estimating the speed. See the "hotel"-scripts for more information.

The purpose of speed_cutout is to avoid division by a small number. The raw shear probe signals and all gradients are generated by dividing them by the speed of profiling. The estimated speed is very small when an instrument is not actually profiling, such as a VMP while it is suspended near the surface before being released to fall freely. Shear probe and gradient data are meaningless, during such occasions, and could be extremely large if the speed is not set to a cut-out value.

Median Filter

Argument	Description
MF_len	Length of data segment [samples] used to calculate the median and standard deviation. Default = 256.
MF_threshold	Data points are defined as bad when they differ by more than this value from the median. Default = [ ].
MF_st_dev	empty string, or 'std_dev'.
MF_k	empty, or the scale factor for the MF_st_dev option. Default = 4.
MF_k_mag	Similar to MF_k but applied to the magnetometer signal. Default = 1.7
MF_extra_points	Number of points, adjacent to bad points, to also replace with interpolated values. Default = 0.

A median filter is used remove flyers, and other erroneous data, from the ADV (acoustic doppler velocimeter) velocity data. See the function `median_filter` for a more detailed description of these parameters. The median filter can be disabled by setting either `MF_threshold` or `MF_k` to `NaN` or `inf`.

Other parameters

Argument	Description
aoa	Angle-of-attack for a glider [degrees]. It is the difference between the glide-path angle and the pitch angle. Default = 3.
time_offset	Time [s] added to the instrument time. The resulting absolute time is offset from the time recorded within the data file header. This facilitates time synchronization with other sources of data and a correction of an erroneously configured instrument clock.
constant_temp	Temperature used for calculating the kinematic viscosity of water when measured temperature is unreliable, or unavailable. Default = [ ].

patch_odas

Find and fix bad buffers in a RSI raw binary data file that can be corrected by a simple shift and interpolation

Syntax

```
[bad_record, fix_manually] = patch_odas( file )
```

Argument	Description
`file`	Name of RSI raw binary data file with bad records.
`bad_record`	Vector of indices to bad records.
`fix_manually`	Vector of indices to records that can not be fixed.

Description

It sometimes happens that a communication failure with an instrument results in some data loss. This is indicated by "Bad Buffer" messages during data acquisition and its occurrence is flagged in the header of the effected record. The result is some skewed data in the binary file that can lead to huge errors if the skew is not fixed before applying the standard data processing tools.

This function is the first step when attempting to correct bad records in a RSI binary data file. It corrects small errors in a record without damaging the surrounding data, by locating the missing value and inserting a "best guess" approximation for that datum.

This only works when damage to a data file is minor. If the damage in a record is extensive, the algorithm leaves such records unchanged. The indices to such records are returned in the `fix_manually` vector.

Records that can not be fixed with this function can be fixed with the `fix_bad_buffers` function. The `fix_bad_buffers` function replaces all data within a record using linear interpretation.

Warning, this function changes the data file. Make a back-up copy of your data file before using this function.

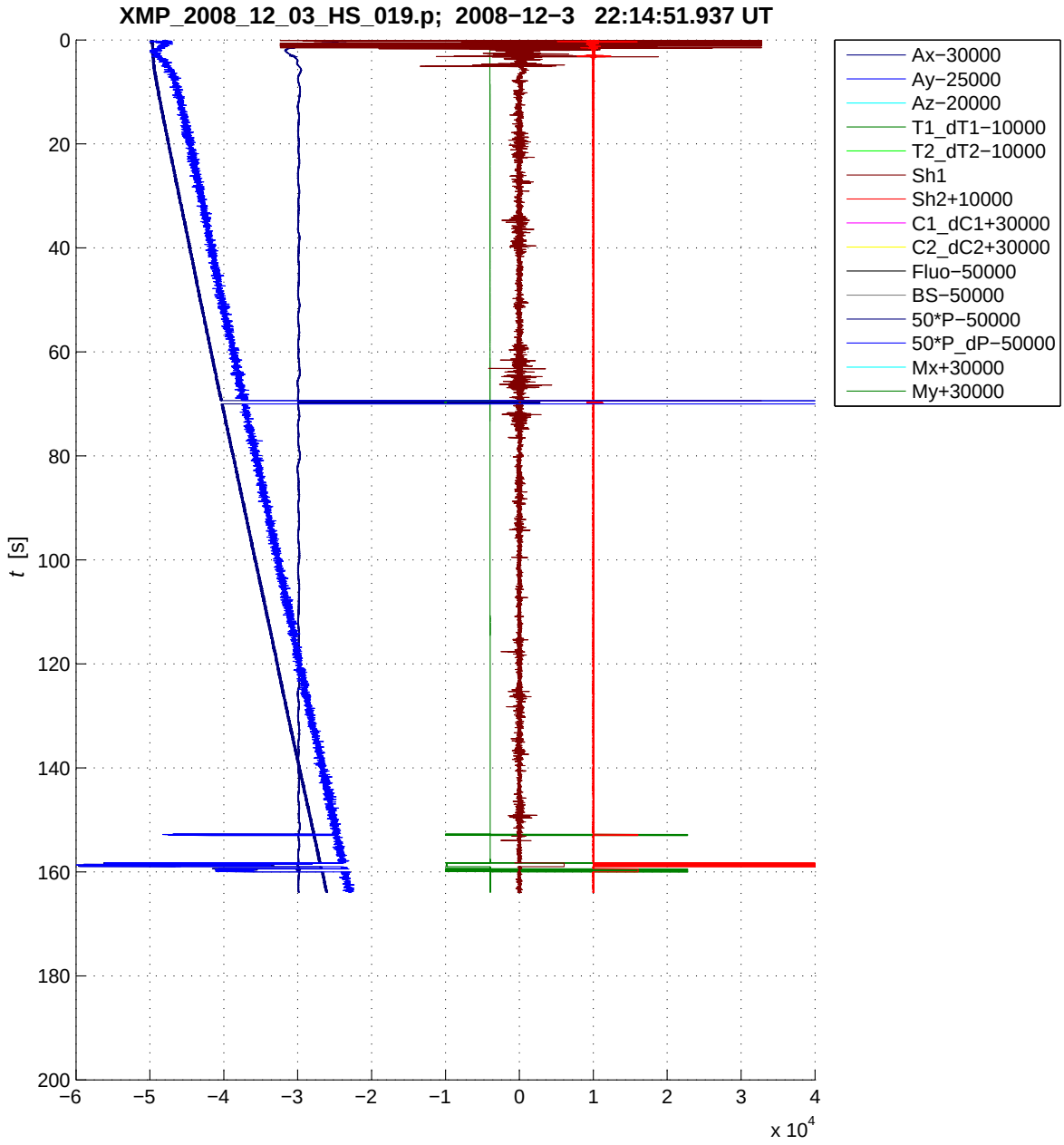

Figure 6.0.9: An example of a raw binary data file that has 4 bad records [70 153 159 160], plotted using `plot_VMP`. This data is from a prototype Expendable Microstructure Profiler. The legend is a wish list of channels. Only a subset exist in this instrument.

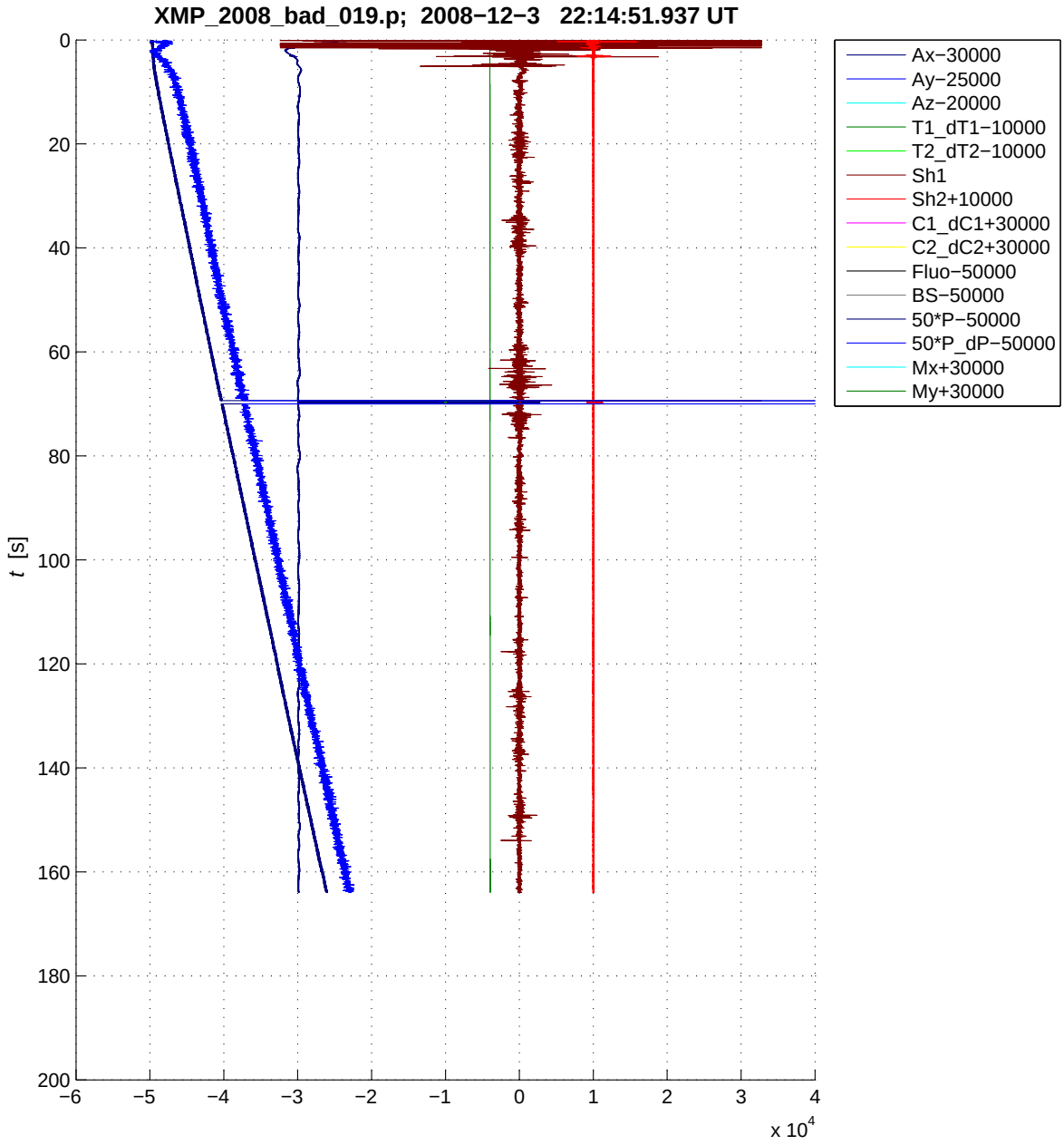

Figure 6.0.10: Plot of the previous figure after using patch_odas. The data in records [153 159 160] had their skew corrected by inserting the missing datum. Corrected records can now be used for data processing. However, record 70 was not fixed because it had too many missing data points. The function fix_bad_buffers can patch this record with interpolated data but must be used cautiously. Such records should not be used for dissipation calculations.

Examples

```
>> copyfile( dataFile, 'original_data_file.p' );
```

```
>> [bad_records, fix_manually] = patch_odas( dataFile );
>> if ~isempty( fix_manually ),
>>     fix_bad_buffers( dataFile );
>> end
```

Repair a data file with bad buffers. First repair the file with `patch_odas`, and then with `fix_bad_buffers`, if necessary.

patch_salinity

Clean, lag and match-filter conductivity data from a JAC-CT sensor

Syntax

```
str = patch_salinity( filename, ... )
```

Argument	Description
`filename`	Name of the mat-file containing JAC-CT data.
`...`	Structure or list of key/value pairs containing input parameters used to control the patching procedure.
`result`	Structure containing pressure, temperature, conductivity, corrected conductivity, salinity, fall-rate, and time data for use with further processing or visualization. When patch_salinity is called with no input parameters, default parameter values are returned within a structure.

Description

Read a mat-file containing JAC conductivity and temperature data and use it to generate a clean and corrected conductivity signal suitable for computing salinity. The mat-file is then updated with the corrected conductivity and salinity values.

A structure containing default parameters values is returned when patch_salinity is called with no input arguments. This structure can be examined, edited, and used as an input to the patch_salinity function. Structure fields that are not recognized are silently ignored.

Optional Inputs

Argument	Description
`lag`	The lag of the conductivity signal relative to the temperature signal, in units of samples. Default = 1.
`f_CT`	The half-power response frequency [Hz] of the thermometer relative to the conductivity cell. Default = 0.6.
`P_min`	Only use data with a pressure greater than P_min, in units of dbar. Default = 1.
`speed_min`	Only use data that has a magnitude of the rate-of-change of pressure greater than speed_min, in units of dbar per second. Default = 0.50.
`MF_len`	Length, in samples, of the segments used by the median filter to remove spikes in the conductivity signal due to plankton and other detritus. See the function median_filter for more detailed information on parameter values. Default = 64.
`MF_threshold`	The threshold for detection of conductivity spikes. Default = [ ].
`MF_st_dev`	Flag that indicates the use of the standard deviation to set the threhold. Default = 'st_dev'.
`MF_k`	The scale factor to apply to the standard deviation to determine the threshold. Default = 4.
`MF_extra_points`	The number of extra points around a spike to also replace with a linear interpolation. Default = 0.

The function goes through the folowing steps. First, it uses the median filter to remove conductivity spikes. It then lags the conductivity signal by `lag` samples. Next, it low-pass filters the conductivity signal using a first-order Butterworth filter with a cut-off frrequency of `f_CT` Hz to match this signal to the temperature signal. It then computes the salinity using the processed conductivity and the original temperature and pressure. Finally, it appends to the data file the low-pass filtered conductivity signal, `JAC_C_LP`, the salinity, `JAC_S`, and the indices, `JAC_C_bad_points`, to the data removed by the median filter.

patch_setupstr

Patch a configuration string into an existing RSI raw binary data file

Syntax

```
patch_setupstr( rawDataFile, configFile, ['-force'], ['-revert'] )
```

Argument	Description
`rawDataFile`	Name of the data file into which 'configFile' should be embedded.
`configFile`	Name of the configuration file to embed.
`'-force'`	Force the patch. Use with extreme caution.
`'-revert'`	Revert to the original configuration file. All previously applied patches will be lost.

Description

Patch an external configuration string into the first record of a RSI raw binary data file.

This function is used to modify the configuration string in a data file. The configuration file may have erroneous conversion parameters that must be corrected, before the file is turned into a mat-file in physical units. It can also be used to upgrade an older-style data file that does not have a configuration string, so that it can be used with this Matlab Library.

When updating a data file with a new configuration string, the original configuration string is commented out and appended to the new configuration string. This retains the contents of the original string, making it possible to revert to the original one.

When patching a data file, the parameters that directly affect data acquisition **MUST NOT** be changed. These values include;

- rate,
- recsize,
- no-fast,
- no-slow,
- matrix.

The `'-force'` bypasses all safety checks. Use with caution.

The `'-revert'` option reverts the configuration string to its original value.

Examples

```
>> patch_setupstr( 'data_005.p', 'setup.cfg' );
```

Embed the configuration string in the file `setup.cfg` into the data file `data_005.p`.

```
>> extract_setupstr( 'data_005.p', 'data_005.cfg' );
>> edit data_005.cfg
>> patch_setupstr( 'data_005.p', 'data_005.cfg' );
```

Example of how `patch_setupstr` can be used with `extract_setupstr` to modify the calibration coefficients embedded within a data file. It is assumed that changes will be made by the `edit` command.

```
>> patch_setupstr( 'data*', '-revert' )
```

Removes all patches to the specified data files. Note the use of a wild card character to specify more than a single file.

Older-style data files, that do not have a configuration string, can be upgraded with a configuration-string, so that the data file can be processed with this version of the ODAS Matlab Library. The upgrade replaces the first record with a new header and the configuration string. No data is lost because there is no data in the first record of older-style data files.

plot_HMP

Real-time or replay plotting of data collected with a horizontal microstruture profiler (HMP).

Syntax

```
plot_HMP( fileName, setup_fn )
```

Argument	Description
`fileName`	name of the data file
`setup_fn`	the name of the channel/data setup file

WARNING

This function is very old and does not make use of modern data files. It is included for users who already use the function and is not recommended for new users.

Description

Function for the real-time (or post-time) plotting of data collected with a horizontal profiler. The plotting style is in the form of horizontal traces in a stack of subplots as shown in the next figure. Up to 16 channels can be plotted. Each screen consists of 50 records of data (which are usually 50 seconds long). All graphical data are displayed in units of counts. The channels that are plotted and the names placed on the y-axis labels are determined by entries in the configuration file. In addition to the time-series, the function will also print on to the plot the average temperature from a Sea-Bird thermometer, the pressure, tow speed, and heading, if these are available. The averages are for the first record on the figure. The name of the data file, the date and time when the data were collected and when they were plotted are also annotated into the figure. Plotting continues for as long as new data are available in the source file. The function is somewhat interactive and the user can choose to have the local printer produce a hardcopy of each 50-record figure. The user can also select to look at a particular 50-record segment.

Examples

The next Figure was produced with the following parameters located in the setup file:

```
plotting: 1,2,3,7,49,50,51,10,11,53,54,14,56,55,16,17,18,19
plotnames: Ax,Ay,Az,T2\_dT2,F_R,F_N,F_C,P,P\_dP,U,V,T7\_dT7,Alt_error,
           Alt,SBT2E,SBT2O,SBC2E,SBC2O
plotaverages: 16,17,10,53,54,32,33
averagenames: SBT2E,SBT2O,Pres,U,V,Mx,My
```

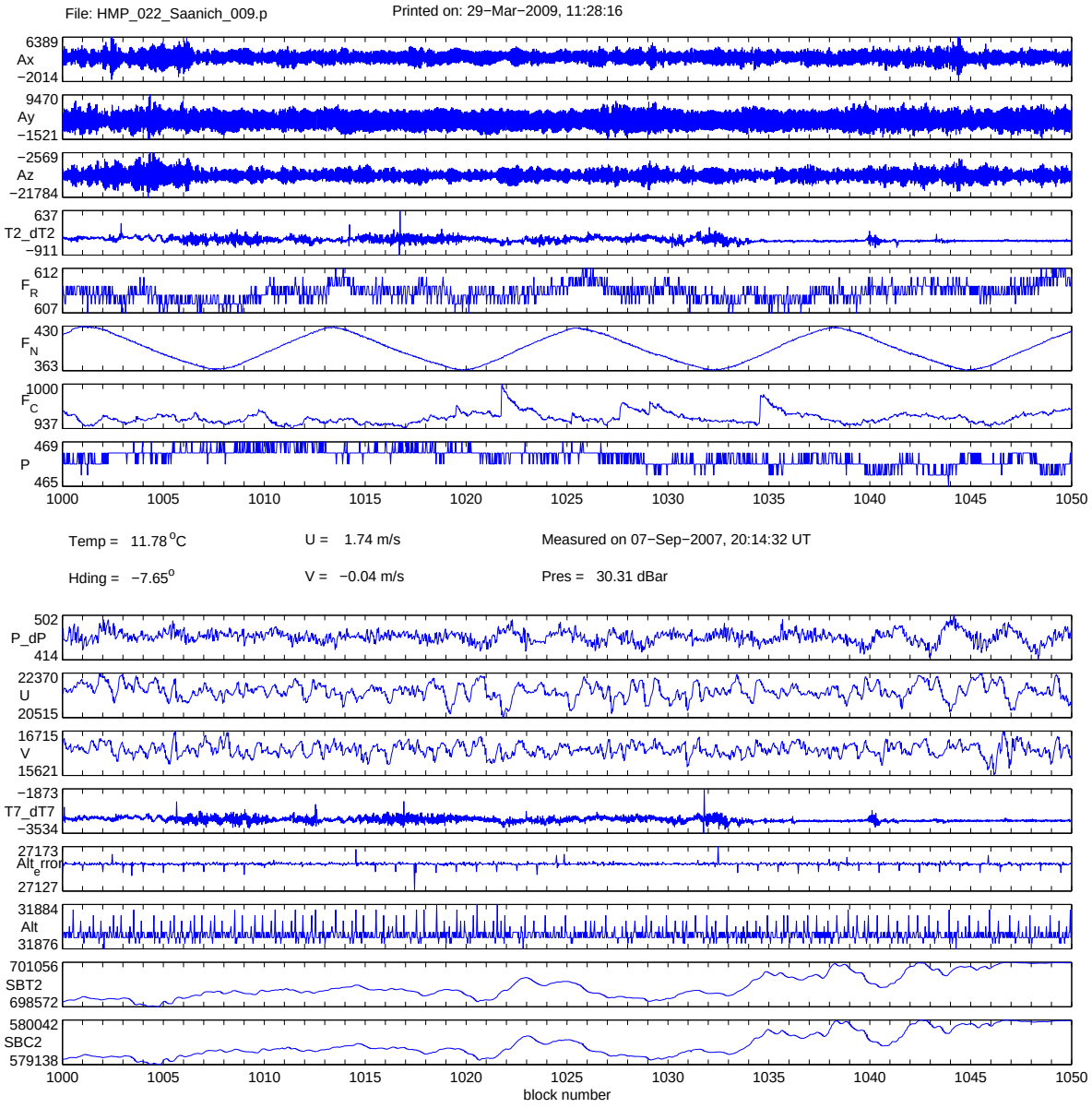

Figure 6.0.11: Example output produced by `plot_HMP`. Data courtesy of Eric Kunze.

Note: This function has not been fully converted to support ODAS v6 and higher. In the meantime, please note the following guidelines and limitations (only for ODAS v6 and higher):

* There is no need to supply a setup file name. All plotting related

```
  parameters are hardcoded at the top of this function or obtained
  from the setup file string.
* Channel averaging is only supported for Pressure at this time.
* Channel averaging for Pressure is configured by hardcoding three
  variables (found at the top of this m-file) as follows:
      ch_mean_seq = [10];
      ch_mean_names = {'Pres'};
      ch_mean_sections = {'pres'};
* Plotting of channels is accomplished by hardcoding the variable
  ch_plot_seq at the top of this function, for example:
      ch_plot_seq = [1 2 3 4 5 6 7 8 9 10 11]';
```

plot_spec

Plot frequency and wavenumber spectra of shear, scalar gradients and acceleration.

Description

This is a legacy function that has been renamed to show_spec. Call show_spec instead of plot_spec in future usage.

plot_VMP

Simple real-time plots of raw data collected with a VMP

Syntax

```
plot_VMP( fileName )
```

Argument	Description
`fileName`	name of the data file to display

WARNING

This function is very old and does not make use of modern data files. It is included for users who already use the function and is not recommended for new users.

Description

Provides a simple but effective preview of the data from a vertical profile. Plotted on a time vs count graph, the data forms descending traces where each trace represents a channel. Visualizing the data in this manner lets one see what is happening to the instrument in either real-time or during a previously recorded acquisition.

When run, the user is asked for the figure duration. The duration is the length of the vertical axis in seconds. The function will plot the requested channels over this duration as a single figure. When the end of the figure is reached, the plot will pause before clearing the graph and continuing from where it left off at the top of the graph.

Data is plotted in units of counts - essentially raw values from the analog to digital converter. When plotted, the resulting traces tend to overlap and are difficult to see. To solve this problem one should apply scalar and offset values to position traces to ensure they can be viewed.

The channels to display, along with their respective scalar and offset modifiers, are controlled by variables defined near the top of this function. The section looks similar to what is shown below:

```
% Variables to be plotted, and their channel numbers
fast_vars     = {'Ax','Ay','Az','T1_dT1','T2_dT2','Sh1','Sh2','C1_dC1',
                 'C2_dC2','Fluo','BS'};
fast_var_nums = [ 1  2  3  5  7  8  9  12  13  14  15 ];
```

```
slow_vars     = {'P','P_dP', 'Mx', 'My'};
slow_var_nums = [10   11      34    33];

set(0,'Defaultaxesfontsize',10,'Defaulttextfontsize',10);
Ax_scale        = 1 ;    Ax_offset         =  -30000;
Ay_scale        = 1 ;    Ay_offset         =  -25000;
            Continued in function....
```

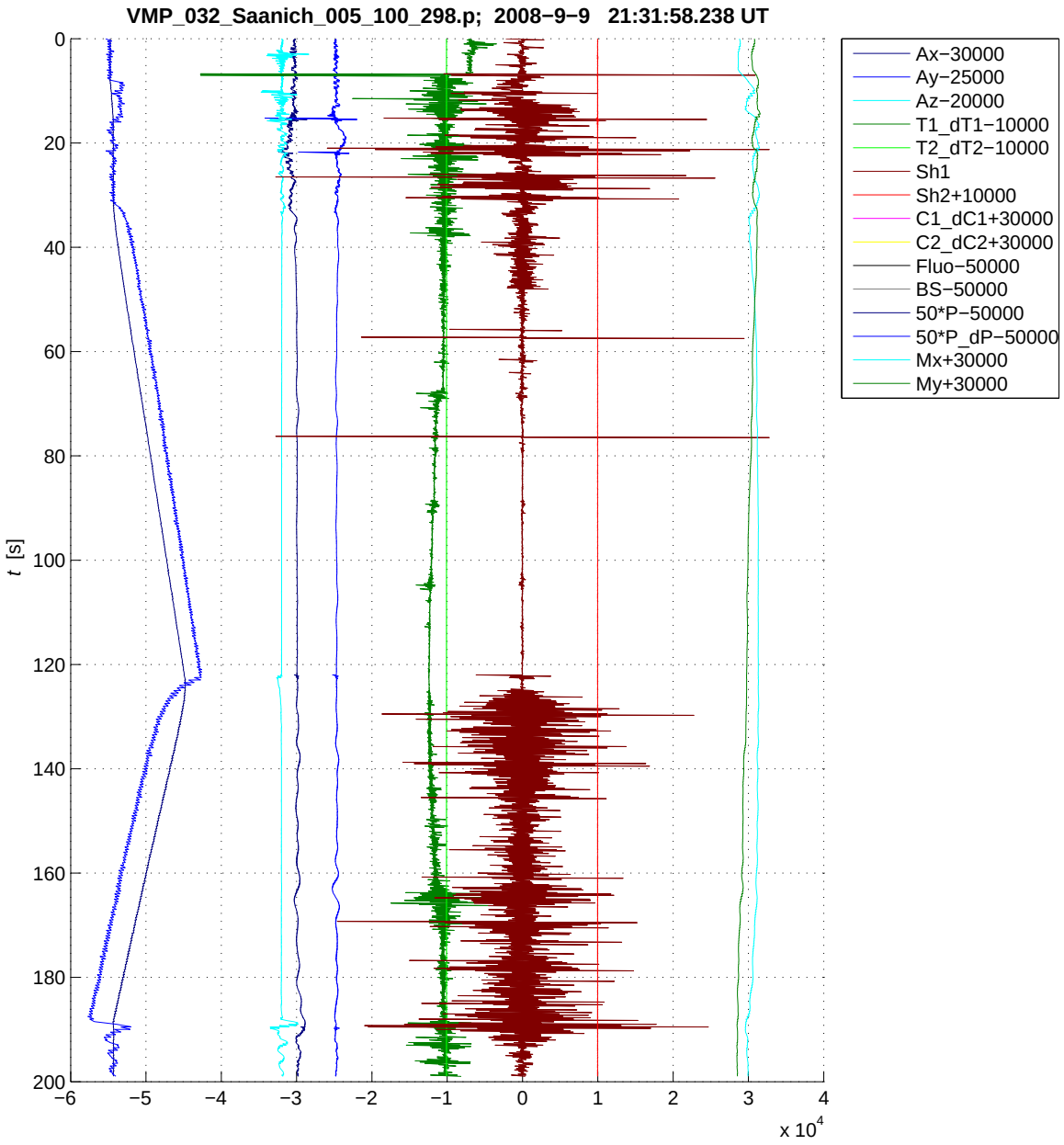

Figure 6.0.12: Example output from `plot_VMP`. A detailed explanation of the plot is found within the function description.

The example plot shows the dark-blue pressure trace on the left side of the figure. The pre-emphasized pressure trace is blue and is next to the pressure trace. At t = 31 s, the pre-emphasized pressure signal separates from the pressure signal because the instrument is falling and has a significant rate of change of pressure. This instrument stopped descending at about t = 123 s. This is evident by the change in the pre-emphasized pressure, a step in the accelerometer signals (the bluish and cyan traces around -30000), and a pulse in the signal from shear probe 1. Shear probe #2 is not installed, nor is thermistor #2.

The main purpose of this function is to give the user a real-time graphical view of the data being collected by a profiler, with no conversion to physical units or other types of processing that might obscure potential problems. The data is shown with all of its warts, such as the frequent spikes in the shear signals due to collisions with plankton (t = 76 s). But the function can also be used to get a quick view of data downloaded from an internally recording instrument. Data courtesy of Manuel Figueroa.

The function shows only the minimum and maximum of consecutive segments of the data, in order to plot the data rapidly. The length of the segments is based on the pixel resolution of your screen. What is shown is an accurate representation of the data. However, the zoom-in function will show details that are not real.

query_odas

Report which channels are in a RSI raw binary data file.

Syntax

```
[ch_slow, ch_fast] = query_odas( fileName )
```

Argument	Description
`fileName`	RSI raw binary data file name (.p) with optional extension
`ch_slow`	vector of slow channels from the address matrix
`ch_fast`	vector of fast channels from the address matrix

Description

Return a list of channel `id` numbers of the data in a RSI raw binary data file, a `p`-file. This function reads the address `[matrix]` in the configuration string, and returns a list of the

channels in the matrix. If two output variables are specified, then there is a separate list for slow and fast channels. If you specify a single output variable, then the slow and fast channels are listed together.

Examples

```
>> query_odas
```

Extract the channel `id` numbers from a data file. The user will be prompted for the file name.

```
>> ch_list = query_odas( 'myfile' );
```

Channel `id` numbers for all of the channels within the file `myfile` are stored into the `ch_list` variable.

```
>> [ch_slow,ch_fast] = query_odas( 'myfile.p' );
```

Channel `id` numbers for the slow and fast channels within the file `myfile` are listed separately in `ch_slow` and `ch_fast`, respectively.

quick_bench

Quick evaluation of a data file collected while the instrument is on a bench.

Syntax

```
quick_bench( 'dataFileName', 'serialNumber' )
```

Argument	Description
`dataFileName`	String. The name of the file to be processed (extension optional).
`serialNumber`	Serial number of the instrument as a string, or any other useful information. This string is placed into a line of the title of each figure.

Argument	Description
`empty`	No return parameters but this function produces two figures.

Description

This function generates two figures from data collected with a RSI instrument, that you wish to test. The instrument should be on a bench, or just standing in a laboratory. Dummy probes should be installed when collecting data. Data should be collected for a few minutes. This function processes the data to produce time-series and spectra of some of the channels in the instrument. The resulting graphs allow the user to determine if an instrument is working correctly.

The graphs are primarily used to detect excessive noise within an instrument. This helps identify problems such as corroded connections and other faults which would otherwise go unseen. For example, you can compare the spectra produced by `quick_bench` against the noise spectra in the calibration report for your instrument.

For real profiles taken in the ocean, use quick_look.m to verify that your instrument is working correctly.

Examples:

```
>> quick_bench( 'data_001.p', '43' )
```

Plot the data in file `data_001.p` collected with the instrument that has the serial number 43. The serial number is not required, but the string will be added to the title of the figures.

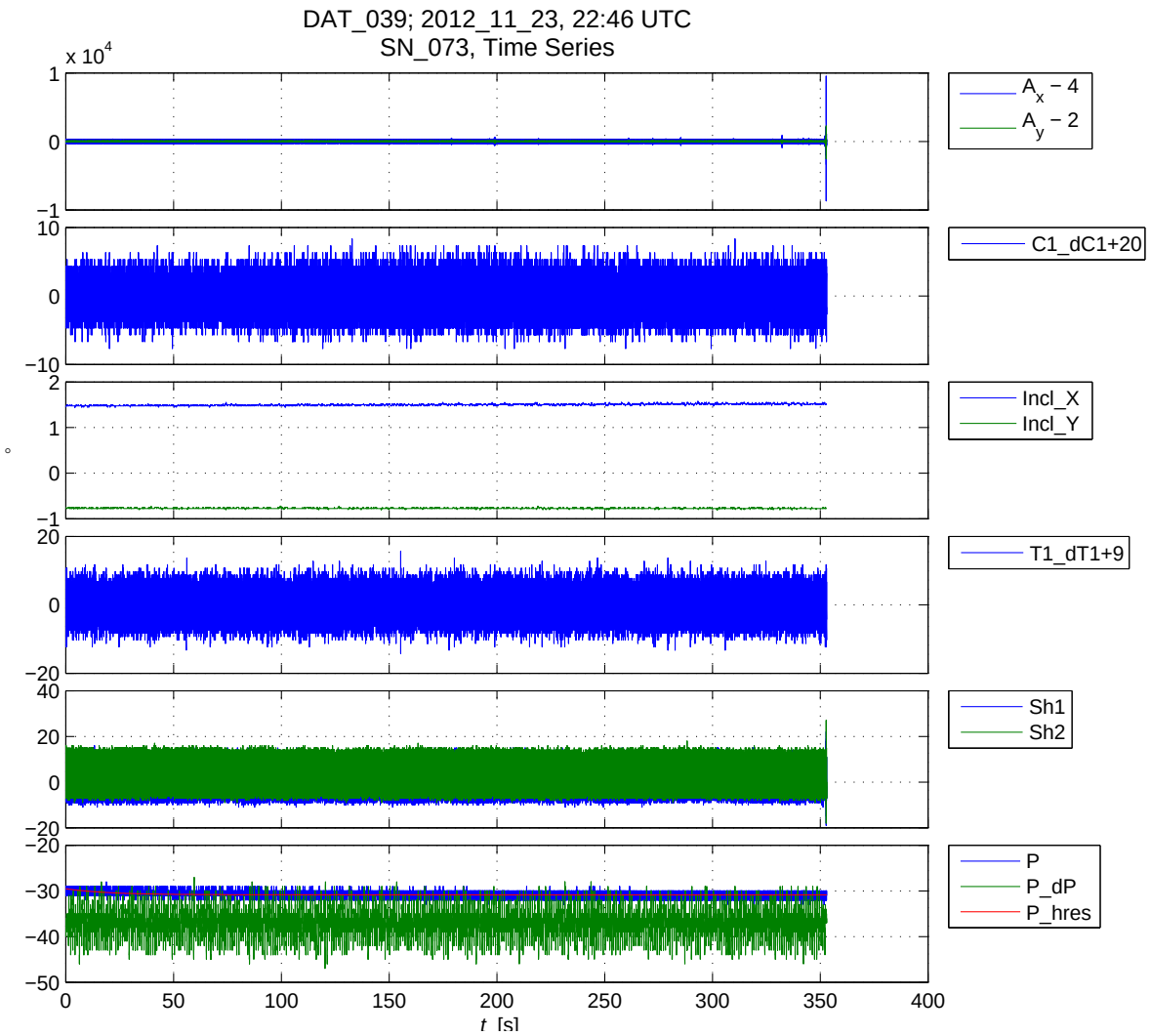

Figure 6.0.13: The time-series output from the `quick_bench` function. Quick_bench tries to plot most of the variables in your raw data file. These include accelerometers, pressure signals, shear probes, thermistors, magnetometers, inclinometers, voltage-output oxygen sensors, micro-conductivity sensors, and JAC -T, -C, -Turbidity and -Chlorophyll sensors. For some signals, the function subtracts the mean and this is indicated in the legend on the right-hand side. Only the inclinometer signals are converted into physical units. All others remain in raw units of counts.

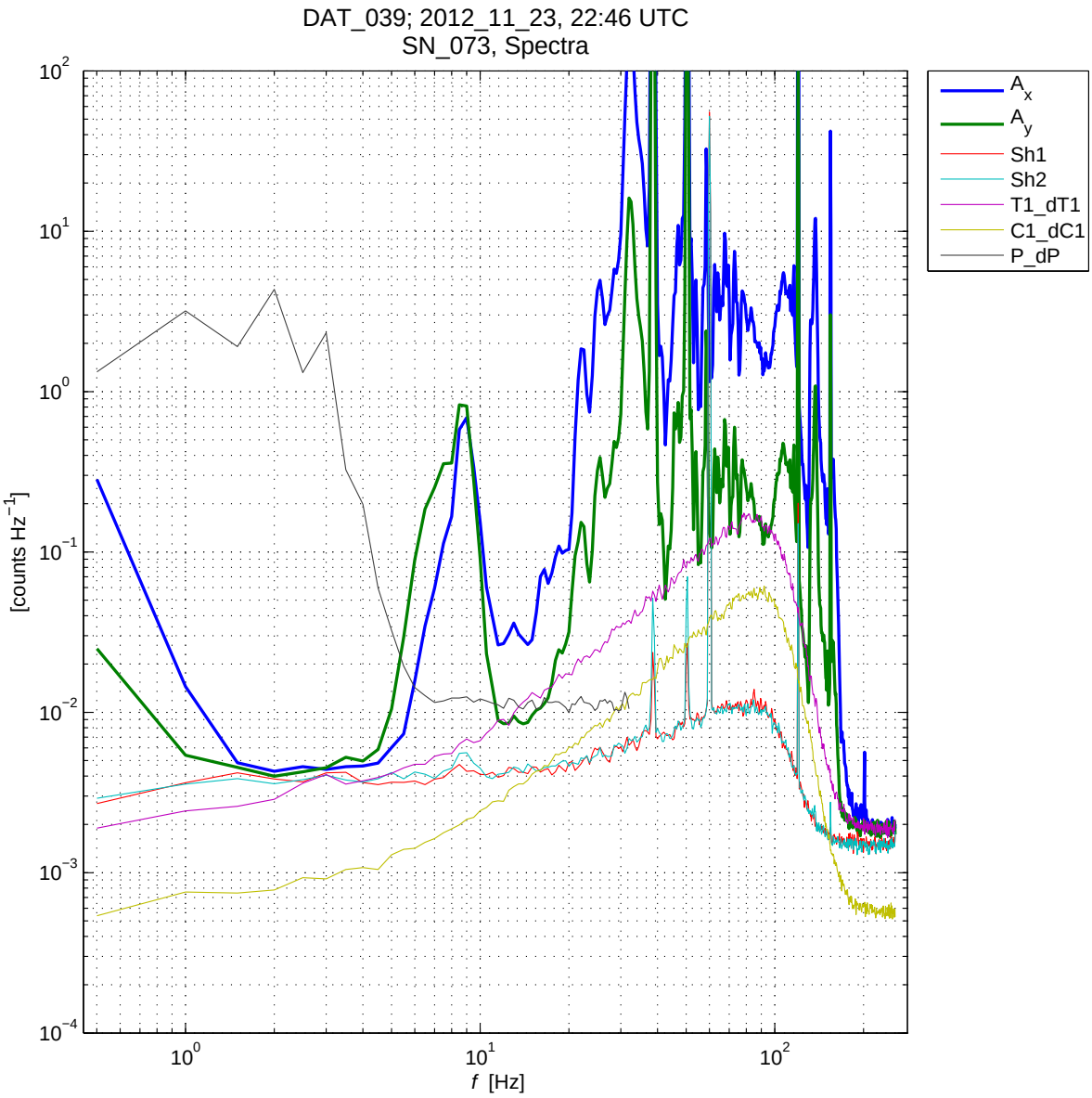

Figure 6.0.14: Spectra of some of the signals shown in the time-series figure. The instrument should be well cushioned to minimize its vibrations. Even so, it is nearly impossible to suppress the output from the extremely sensitive accelerometers. AC power line frequency (50/60 Hz) contamination is also difficult to suppress and may show up as narrow spectral peaks. Dummy probes have been installed in place of the shear probes, thermistor and micro-conductivity sensors. Their spectra can be directly compared against those in your instrument calibration report to check if the noise level is close to that observed at RSI before your instrument was shipped.

quick_look

Visualize contents of a RSI data file, compute spectra for a selected range, and return a profile of the rate of dissipation of kinetic energy.

Syntax

```
diss = quick_look (fname, P_start, P_end, ql_info, ...)
```

Argument	Description
`fname`	Name of the binary data file to process (extension optional).
`P_start`	Starting point, in pressure, of the segment to be used for displaying a spectrum of all shear, scalar-gradient, and acceleration signals. Can be empty, if P_end is also empty, to suppress the display of the spectra.
`P_end`	End point, in pressure, for the segment used to show spectra. Can be empty, if P_start is also empty.
`ql_info`	Structure containing configuration parameters. A template is generated and returned when quick_look is called with no input parameters. The parameters are described below.
`...`	Optional configuration parameters to supplement, or override, those values included within ql_info.
`diss`	Dissipation estimates for the specified profile. A default ql_info structure is returned when quick_look is called with no input parameters.

Description

This function generates a variety of figures that visualize the data in the file `fname`. The function works in three stages. In the first stage, data is converted into physical units using the `odas_p2mat` function, if a mat-file does not already exist. In the second stage, the data are plotted in several figures. In the third stage, the function computes a profile of the rate of dissipation of kinetic energy from the shear probe data.

Converting data into physical units is performed by the `odas_p2mat` function. `quick_look` calls this function if a mat-file with the same name as the data file does not exist. Or, if the parameters for the `odas_p2mat` function (which may be included in the call to the `quick_look` function) are not the same as those that were used to create the mat-file. You will be prompted for permission to overwrite the mat-file. Therefore, input parameters used with `quick_look` can include the parameters required to convert your data into physical units, but these parameters are only required if the mat-file does not exist.

If `quick_look` is called without input arguments, it returns the default parameters used by `quick_look` and by `odas_p2mat`, within a single structure, that you can customize to your particular processing requirements. For example,

```
>> ql_info = quick_look

ql_info =
                HP_cut: 0.4000
                LP_cut: 30
                  YD_0: 0
             despike_A: [8 0.5000 0.0400]
             despike_C: [10 1 0.0400]
            despike_sh: [8 0.5000 0.0400]
           diss_length: 8
                  f_AA: 98
               f_limit: Inf
            fft_length: 2
             fit_2_isr: 1.5000e-05
             fit_order: 3
          make_figures: 1
               op_area: 'open_ocean'
               overlap: 4
         profile_min_P: 1
         profile_min_W: 0.2000
 profile_min_duration: 20
           profile_num: 1
      MF_extra_points: 0
                  MF_k: 4
              MF_k_mag: 1.7000
                MF_len: 256
             MF_st_dev: 'st_dev'
          MF_threshold: []
                   aoa: []
        constant_speed: []
         constant_temp: []
            hotel_file: []
          speed_cutout: 0.0500
             speed_tau: []
           time_offset: 0
               vehicle: ''
```

The configuration parameters (fields) within the structure `ql_info` that control the behaviour of `quick_look`, are listed below. They are grouped for clarity, but are all part of the single structure `ql_info`, including those required only by `odas_p2mat`.

Parameters that specify a profile

A single data file may contain multiple profiles. For example, a verticle profiler that was raised and lowered multiple times while recording data into a single file. `quick_look` detects profiles using 5 specifications. These are:

Argument	Description
`profile_num`	Index to the requested profile from the set of detected profiles. The first profile, 1, is the default value.
`profile_min_P`	The minimum pressure of a profile, [dbar]. Default = 1.
`profile_min_W`	The minimum vertical speed of profiling, [dbar/s]. Default = 0.2
`profile_min_duration`	The minimum duration in which the minimum pressure and speed must be satisfied [s]. Default = 20.

The `direction` of profiling is the implict 5th specification and is determined by the `vehicle` used for profiling. See the file `default_vehicle_attributes.ini` for the direction of each recognized `vehicle`.

Parameters that control the calculation of dissipation rate

The rate of dissipation of turbulent kinetic energy, ϵ, is estimated using the function `get_diss_odas` and follows the method described in RSI Technical Note 028. The controlling parameters are:

Argument	Description
diss_length	The time span, in seconds, over which to make each estimate of the rate of dissipation. Default = 8.
overlap	The overlap, in seconds of each dissipation estimate. Default = 4.
fft_length	The length, in seconds, of the fft-segments that will be ensemble-averaged into a spectrum. Default = 2.
fit_2_isr	The rate of dissipation, in W/kg, above which the function will switch from the method of spectral-integration to the method of fitting to the inertial subrange, to estimate the rate of dissipation. Default = 1.5e-5.
fit_order	The order of the polynomial to fit to the shear spectrum, in log-log space, to estimate the wavenumber at which the spectrum has a minimum. This is one of several constraints on the upper limit of spectral integration. Default = 3.
f_limit	The upper frequency limit, in Hz, to be used for estimating the rate of dissipation. Default = inf (no unconditional limit).
f_AA	The cut-off frequency, in Hz, of the anti-aliasing filter in your instrument. Default = 98. This value is instrument dependent but is almost always 98, unless you have an instrument that has been customized to samples at rates faster than 512 per second.

Parameters that control the processing of microstructure signals

The parameters that control the processing of the microstructure signals pertain to the high-pass filtering of the shear-probe signals, and the despiking of the shear-probe, acceleration and micro-conductivity signals.

Argument	Description
HP_cut	The cut-off frequency, in Hz, of the high-pass filter applied to the shear-probe signals. Default = 0.4.
despike_sh	The triplet of parameters for the despike function applied to the shear-probe signals. The first value is the threhold. The second value is the cut-off frequency, in Hz, of the low-pass smoothng filter. The third value is the duration, in seconds, of data to remove around a spike. Default = [8 0.5 0.04]. See the function despike for more information on the parameters. You can suppress the despike function by specifying an infinite threshold, for example [inf 0.5 0.07].
despike_A	The triplet of parameters for the despike function applied to the accelerometer signals. For data collected with a glider, it may be necessary to supress despiking. The intermittent vibrations from battery movement and fin actuators creates short duration vibrations that are easily confused with spikes, but such data is needed for coherent-noise removal. Default = [8 0.5 0.04].
despike_C	The triplet of parameters for the despike function applied to the microconductivity signals. Default = [10 1.0 0.04].

Parameters for other purposes

Argument	Description
LP_cut	The cut-off frequency, in Hz, of the low-pass filter applied to the microstructure profile signals, for graphical display only. It does not affect the estimation of the rate of dissipation. Default = 30.
YD_0	The year-day subtracted from the time axis of figures. It is currently not used. Default = 0.
op_area	The operational area of your instrument. Recognized values are 'open_ocean' and 'tidal_ch'. It controls the scale on certain figures. Default = 'open_ocean'.
make_figures	The parameter that determins if figures are generated. make_figures = false suppresses the generation of figures to speed up the data processing. The default is make_figures = true.

Parameters for the odas_p2mat function

The parameters starting with MF_extra_points are used by the odas_p2mat function, to convert your data into physical units. They are described in the section for that function.

The diss structure output

The output structure from `quick_look` depends slightly on the channels in your instrument. The typical fields are shown below, in groups. The first group is associated with the calculation of the profile of the rate of dissipation, ϵ, and is described in the section for `get_diss_odas`.

```
                  [e]  [2x217 double]
              [K_max]  [2x217 double]
            [warning]  [2x217 double]
             [method]  [2x217 double]
        [Nasmyth_spec]  [513x2x217 double]
                 [sh]  [4-D double]
           [sh_clean]  [4-D double]
                 [AA]  [4-D double]
                 [UA]  [4-D double]
                  [F]  [513x217 double]
                  [K]  [513x217 double]
          [Data_fast]  [18x217 double]
          [Data_slow]  [19x217 double]
              [speed]  [217x1 double]
                 [nu]  [217x1 double]
                  [P]  [217x1 double]
                  [T]  [217x1 double]
                  [t]  [217x1 double]
               [AOA]  []
              [f_AA]  88.2000
            [f_limit]  Inf
          [fit_order]  3
        [diss_length]  4096
           [overlap]  2048
         [fft_length]  1024
```

The next group is associated with the despiking of the shear-probe, micro-conductivity and acceleration signals.

```
            [spikes_A]  {[4x1 double], []}
         [pass_count_A]  [2x1 double]
           [fraction_A]  [2x1 double]
            [spikes_sh]  {[247x1 double], [269x1 double]}
       [pass_count_sh]  [2x1 double]
         [fraction_sh]  [2x1 double]
            [spikes_C]  {}
         [pass_count_C]  [0x1 double]
           [fraction_C]  [0x1 double]
```

The indices to the spikes located in the signals from the accelerometers, are given in `spikes_A`. The number of passes of the despike function used to remove the spikes is in `pass_count_A`. The fraction of data removed by the despike function is in `fraction_A`. Similarly for the shear probe and the micro-conductivity signals.

The next group is associated with the scalar signals.

```
          [scalar_spectra]  [1x1 struct]
      [scalar_vector_list]  {'gradT1'  'gradT2'}
             [scalar_info]  [1x1 struct]
```

The structure `scalar_spectra` (a structure within a structure) is described in the section for the function `get_scalar_spectra_odas`. The processing parameters are in `scalar_info`. The names of the scalar vectors are in `scalar_vector_list`.

The remaining fields are:

```
                  [fs_fast]  511.9454
                  [fs_slow]  63.9932
    [profiles_in_this_file]  1
             [speed_source]  'Rate of change of pressure'
                [fast_list]  {1x18 cell}
                [slow_list]  {1x19 cell}
                  [ql_info]  [1x1 struct]
               [ql_info_in]  []
```

`fs_fast` and `fs_slow` are the actual fast and slow sampling rates of the data. The number of profiles detected in this data file is given in `profiles_in_this_file`. The source of the speed of profiling is identified in `speed_source`.

All column vectors that have a length matching the length of the time vector `t_fast` are combined into a single matrix and passed to the dissipation function for averaging over the interval of each dissipation estimate. Similarly for all column vectors that match `t_slow`. Each row of `Data_fast` and `Data_slow` hold the values from a single vector. The names of the signals are identified in the cell arrays `fast_list` and `slow_list`. In this example there are 19 fast vectors and 19 slow vectors. Use the Matlab `find` and `strcmpi` functions to identify the row of a particular signal.

The structure that was input to this function is saved in field `ql_info_in` and is empty in this example because the call used default values. The structure that was actually used to process the file, in this case a structure of default values, is given in the field `ql_info`.

read_odas

Convert a RSI raw binary data file into a raw mat-file

Syntax

```
[ch_list, outstruct] = read_odas( fname )
```

Argument	Description
`fname`	Name of the RSI raw binary data file. Optional, you are prompted, if omitted.
`ch_list`	List of channels found within the resulting mat-file.
`outstruct`	Structure holding contents of the mat-file. If this output argument is used, the mat-file is not created.

Description

Convert a RSI raw binary data file (.p) into a Matlab mat-file, without conversion into physical units. The raw data file is read, demultiplexed, and assembled into Matlab data vectors. All channel data, along with other relavent information such as the configuration string, is converted into a Matlab-readable format.

If created, the mat-file is given the same name as the input file with the `.p` extension replaced by `.mat`. If this file already exists, you are queried for an alternate name.

When the `outstruct` variable is used, a mat-file is not generated. Instead, all values that would have been placed into the mat-file are returned as fields in the structure `outstruct`.

The vectors are not converted into physical units. However, 1e-12 is added to every vector so that Matlab is forced to save them as true double-precision floating-point numbers.

Examples

```
>> read_odas
```

Extract all data from a `.p` file and store inside a newly generated mat-file. Input file requested from the user with the most recent data file being the default value.

```
>> my_vars = read_odas('my_file.p')
```

Extract data from `my_file.p` and store them in the file `my_file.mat`. A list of the extracted channels is in `my_vars`. The `.p` extension is optional.

```
>> [myVar, d] = read_odas( '*03' )
```

Read the first raw data file that has a name ending in `03`. Do not create a mat-file. All data are returned within the structure named `d`.

salinity

Compute salinity at a specified pressure, temperature and electrical conductivity

Syntax

```
sal = salinity( P, T, C )
```

Argument	Description
`P`	Pressure [dbar]
`T`	Temperature [C]
`C`	Conductivity [mS/cm]
`sal`	Salinity in practical salinity units [PSU]

Description

Compute salinity from vectors of pressure, temperature and electrical conductivity.

checkvalue: c = 42.914, s = 35 psu, t = 15.0 deg C , p = 0

Based on Fortran routine developed at WHOI.

save_odas

More efficient way to [re]write a vector in an existing mat-file.

Syntax

```
success_flag = save_odas(file_name, vector_name, vector)
```

Argument	Description
`file_name`	Destination mat-file into which the vector should be saved.
`vector_name`	Name of the vector being saved.
`vector`	Vector to save.
`success_flag`	Result of operation, see following table:

Possible values for success_flag:

- 0: success, vector found in .mat file is of equal length to the vector being saved;
- 1: success, vector not found in .mat file so the vector is saved using the '-append' flag;
- -1: failure, vector found in .mat file is of different length to the vector being saved. The .mat file was not modified.

Description

This function greatly reduces the time required to save a modified real vector back into a mat-file.

The basic idea of this function is that the entire data file does not have to be rewritten when the vector that you are saving has the same name and length as one that is already in your mat-file.

NOTES:

- This function works only with real vectors.
- This function only works with MATLAB Version 5 (save -v6) or earlier file formats. Newer file formats are compressed and can not take advantage of this shortcut.

Example

```
>> load  my_file  my_vector
>> my_vector  =  5 * my_vector;
>> save_odas( 'my_file.mat',  'my_vector',  my_vector);
```

Extract one, of possibly many, long vectors from a mat-file. Modify it without changing its length. Then save it.

segment_datafile

Break data file into segments around bad buffers

Syntax

```
segment_datafile( file_name, min_record_length )
```

Argument	Description
file_name	Name of file with or without extension. Wildcard characters accepted in the same format as the dir() function.
min_record_length	Minimum size of a valid data segment. Default = 20.

Description

Segment a raw RSI data file with BAD BUFFERS into smaller data files without BAD BUFFERS.

Interpolation of bad buffer data works well for patching over short bad buffer segments within a data file. This is the default behaviour performed by the quick_look function. When bad buffers persist for multiple records, simply patching the bad buffers with interpolated values can induce notable errors in data channels with pre-emphasis.

This function removes bad buffer data by segmenting the original data file into segments where each segment contains no bad buffers. This prevents interpolated values from being used when deconvolving pre-emphasised channel data.

Each resulting data file has a minimum length of min_record_length records (seconds). The new files are named by appending _XX to the original file name where _XX is a counter that

increments for each observed segment. If the orginal file does not have a BAD BUFFER then the only file created is _00. This allows subsequent functions and scripts to work regardless of the number of BAD BUFFERS in a file.

setupstr

Read attributes of a configuration string from a RSI raw binary data file.

Syntax

```
varargout = setupstr( varargin )
```

Argument	Description
`varargin`	see Usage
`varargout`	see Usage

Description

This function will extract requested attributes from an input string. The string should be a configuration string that was extracted out of a RSI data file. This string can also be obtained by reading a configuration file (.cfg).

Usage

The setupstr function is called with up to four arguments. Depending on the number and type of arguments used, different results are returned.

The first argument is the configuration string, either extracted from a data file or loaded from a configuration file. When this is the only argument, the `setupstr` function returns an indexed structure containing the contents of the configuration string. This indexed structure can be used in place of the configuration string, for subsequent function calls, to greatly increase the speed of this function.

```
>> obj = SETUPSTR('config_string')
```

Parse the string `'config_string'` and return `obj`, a data structure containing the values in `config_string`. The returned `obj` can be used in subsequent calls to the function.

```
>> S = SETUPSTR( obj, 'section' )
```

Find sections within `obj` that match the value `'section'`. See Notes for additional information regarding search queries. The returned value `S` is a cell array containing the matching section identifiers in string format.

```
>> V = SETUPSTR( obj, 'section', 'parameter' )
```

Find parameter values within `obj` that are located in `'section'` and have the parameter name `'parameter'`. See Notes for additional information regarding search queries. The returned value `V` is a cell array containing the matching string values.

```
>> [S, P, V] = SETUPSTR( obj, 'section', 'parameter', 'value')
```

Find values within `obj` that are located in `'section'`, have the parameter name `'parameter'`, and the value `'value'`. See Notes for additional information regarding search queries. The returned values are a tuple of cell arrays containing the matching sections, parameter names, and parameter values.

Argument	Description
`config_string`	Configuration string.
`obj`	A previously returned structure containing indexed values from a configuration string.
`section`	Search query for a matching section identifier.
`parameter`	Search query for a matching parameter name.
`value`	Search query for a matching parameter value.
`obj`	Structure containing indexed values from a configuration string, that can be used in subsequent calls to this function to speed the search.
`S`	Matching sections as an array of cells.
`P`	Matching parameter names as an array of cells.
`V`	Matching parameter values as an array of cells.

Notes

The `[root]` section in a configuration file is usually not declared explicitly. It exists implicitly and consists of all content before the first explictly declared section. To access the parameters within the `root` section, use `'root'` for the section identifier.

Examples

```
>> cfg = setupstr( setupfilestr );
>> value = setupstr( cfg, 'P', 'coef0' )
```

From the variable `setupfilestr`, which is a string containing the entire configuration file used for data acquisition, query the value of the parameter `'coef0'` for the pressure channel.

```
>> sections = setupstr( setupfilestr, '' )
```

Find all sections within a configuration file.

```
>> cfg = setupstr( setupfilestr );
>> [S,P,V] = setupstr( cfg, '', 'id', '' )
```

Find all sections that have a parameter with name `id`, and place the section names into the cell-array `S`. The values of the `id`-parameters are placed into `V`, and `P` is filled with the value `'id'`. `S`, `P`, and `V` have identical length.

show_P

Extract the record-average pressure from a RSI raw binary data file.

Syntax

```
[P, record] = show_P( fileName )
```

Argument	Description
`fileName`	- String containing the name of a RSI raw binary data file.
`P`	- Vector of the record-average pressure in units of dbar.
`record`	- Vector of the record numbers corresponding to P.

Description

Calculate the record-average pressure from a RSI raw binary data file. This function is used to quickly glimpse the pressure-time history of an instrument. The pressure channel address must be present in the address `[matrix]` and there must be a `[channel]` section containing the coefficients for converting raw pressure data into physical units.

The returned vectors are suitable for plotting the pressure history in a data file.

Examples

```
>> [P records] = show_P( 'my_data_file.p' );
>> plot(P); set(gca, 'YDir', 'reverse'); grid on;
```

Generate a pressure (depth) vs. record (time) line plot to reveal the descents and ascents of your instrument.

Use

```
>> plot(diff(P)); grid on;
```

to see the rate-of-change of preesure (vertical velocity).

show_spec

Plot frequency and wavenumber spectra of shear, scalar gradients and acceleration.

Syntax

`show_spec( diss, titleString )`

Argument	Description
`diss`	Structure of dissipation and other information returned by, for example, the functions quick_look, get_diss_odas, or get_scalar_spectra_odas.
`titleString`	Optional string to prepend to each figure title. Use it to help identify the current profile.

Description

Function to plot the spectra of shear, scalar gradients and acceleration that are embedded in the structure `diss`.

The structure `diss` contains sets of spectra, one set for every dissipation estimate within the structure. This GUI plots all spectra (shear, Nasmyth, scalar-gradients, and acceleration) from a chosen set, and lets you tranverse among the sets of spectra using a slider bar, or the arrow keys. The spectra are displayed in a single figure, with frequency spectra on the right, and wavenumber specra on the left. There are options to export a single set into a pdf-file. Optimal settings for PDF export are pre-selected and can be changed.

Quick Overview

The `show_spec` function visualizes spectra that was generated using a function such as `quick_look`. As such, before show_spec can be run the spectra must be generated.

```
>> diss = quick_look('DAT_001', 10, 20);
```

With the `diss` structure generated, `show_spec` can be launched.

```
>> show_spec( diss )
```

show_spec will now display the main window. The window displays the spectra from one segment used to estimate the rate of dissipation, ϵ. The segment is identified by its index number in the title, along with the segment-average pressure, speed and temperature. Use the arrow keys or scrollbar, to navigate among all segments in the `diss` structure.

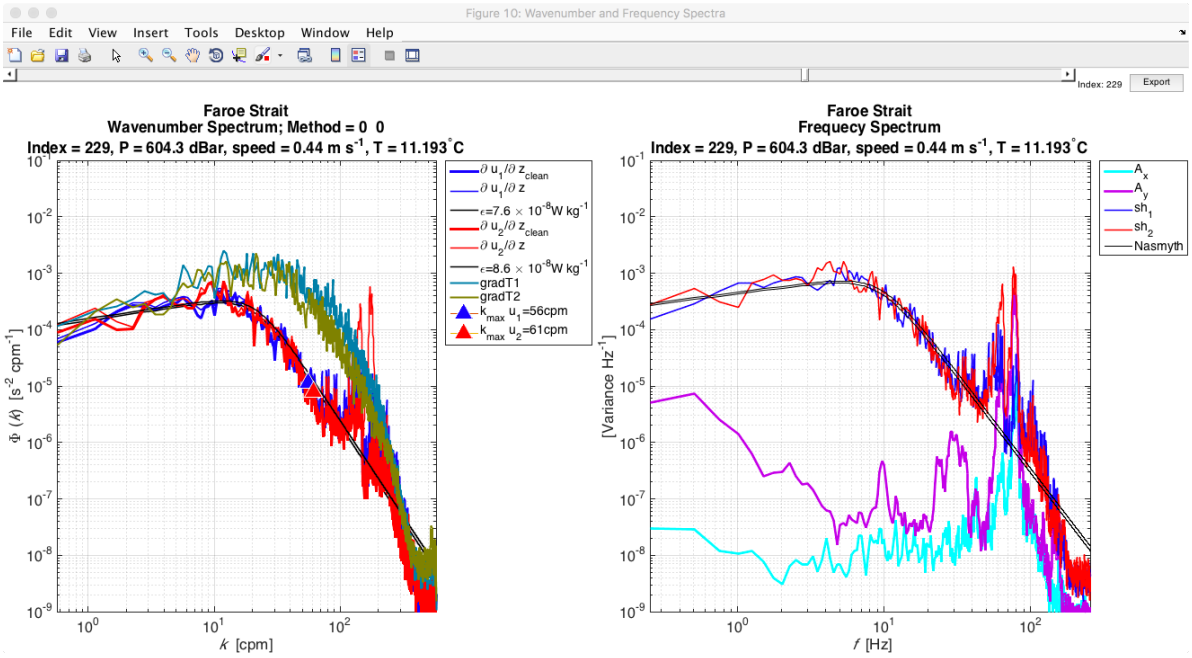

Figure 6.0.15: View of the `show_spec` main window. You navigate between segments by either using the arrow keys, changing the scroll bar at the top of the window, or by entering a specific index into the text field located in the top right corner of the window. Data courtesy of Ilker Fer, University of Bergen.

Spectra of interest are exported using the "Export" button found at the top of the screen. This button opens a dialog box that simplifies the saving of a plot into a file.

Plots are generated using the third-party package `export_fig`. This package bypasses the MATLAB PDF rendering engine to use a locally installed version of Ghostscript to generate PDF files that are attractive and well suited for publication.

You must manually download and install `export_fig` before `show_spec` can export images. The package is free, open source, and available from the MATLAB file exchange website.

`export_fig` requires a local version of Ghostscript. A dialog box with instructions is displayed if a local version of Ghostscript is not found. Users of LaTeX likely have Ghostscript already installed.

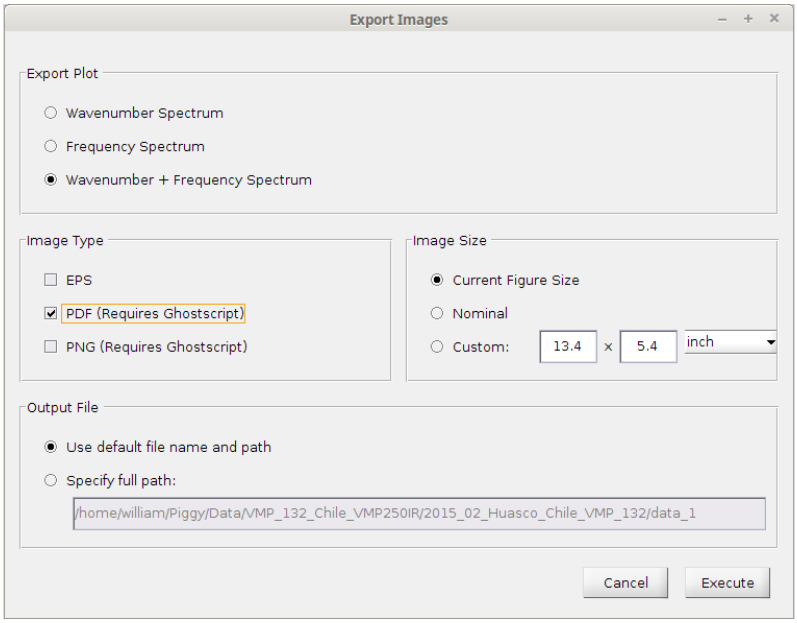

Figure 6.0.16: Exporting images is simplified by using the export dialog - activated by the "Export" button found on the main window.

Notes

- Default font sizes are used when generating plots. On some computers the default values might be too small. To change the default font size for all MATLAB plots, put the following into your startup.m configuration file.

  ```
  >> set(0,'DefaultAxesFontSize',14)
  ```

- This function requires Matlab version 2014b and higher.

texstr

Format a string for use by a TeX interpreter

Syntax

```
result = texstr( input, escChar )
```

Argument	Description
`input`	String, cell, or cell array to reformat.
`escChars`	String containing characters to interpret literally. Default is '_\'.
`result`	Input with all specified characters prepended with a forward slash '\'. Output type is the same as the input type – string, cell, or cell array.

Description

Parse the input string and escape specified characters with a forward slash '\'. This allows the strings to be used with third party applications such as LaTeX and was originally written to facilitate underscore characters within Matlab plot labels.

TI_correction

Derive the corrected temperature of the fluid in a conductivity cell.

Syntax

```
T_cell = TI_correction( T, alpha, beta, fs )
```

Argument	Description
`T`	Environmental temperature measured by a thermometer.
`alpha`	Amplitude coefficient for the thermal inertial of the conductivity cell.
`beta`	Inverse relaxation time for the thermal inertia of the conductivity cell, in units of 1/s.
`fs`	Sampling rate of the data, in samples per second.
`T_cell`	Water temperature in the cell. This value should be used for salinity calculations but not for density calculations.

Description

Adjust the temperature measured with a thermometer to account for the temperature anomaly in a conductivity cell due to the thermal inertia of the cell. This function was originally written to process data collected with the Sea-Bird SBE3F and SBE4C sensors, but it can be used with any conductivity cell. This function is based on Lueck (1990), Lueck and Picklo (1990) and Morrison et al (1994). Familiarity with these papers is nearly a pre-requisite for using this function.

The algorithm originally proposed by Lueck and Picklo (1990) tries to adjust the measured conductivity to account for the thermal inertial of the SBE4C cell. That is, it tries to adjust the measured conductivity to what it would be in the absence of thermal inertia. However, this is indirect and problematic because the temperature coefficient of conductivity:

$$\left.\frac{\partial C}{\partial T}\right|_{P,S}$$

is pressure-, salinity-, and temperature-dependent, and a single value can not be used for an entire profile. The algorithm proposed by Morrison et al (1994) is more direct and requires no further coefficients. They argue that the conductivity reported by the cell is the correct value for the fluid in the cell. The problem is that its temperature is different from that

reported by the thermometer, and is wrong by the amount proposed by Lueck (1990). So a much simpler approach is to estimate the temperature in the cell and use it, with the measured conductivity and pressure, to calculate the salinity. The density is then calculated with this salinity, the real temperature in the water, and the pressure.

The coefficients `alpha` and `beta` are pump-speed dependent. The original estimates by Lueck and Picklo (1990) have `alpha` = 0.021, and `beta` = 1/12 for an un-pumped system. A pumped system will have a larger `beta`. The user has to play with these coefficients to get the best correction but even poor values reduce significantly the errors in the calculated salinity. These errors are frequently called 'salinity spikes'.

This function corrects only the thermal inertia. The short-term mismatch between the measured temperature and conductivity must be corrected before using this function.

- Lueck, R.G., 1990, Thermal inertia of conductivity cells: Theory, J. Atmos. Oceanic. Technol., 7, 741-768.
- Lueck, R.G. and J.J. Picklo, 1990, Thermal inertia of conductivity cells: Observations with a Sea-Bird cell, J. Atmos. Oceanic Technol., 7, 756-768.
- Morrison, J., R. Anderson, N. Larson, E, D'Asaro and T. Boyd, 1994: The correction for thermal-lag effects in Sea-Bird CTD data., J. Atmos. Oceanic Technol., 11, 1151-1164.

visc00

The kinematic viscosity of freshwater, $S = 0$.

Syntax

```
v = visc00( T )
```

Argument	Description
`T`	temperature in degrees Celsius
`v`	kinematic viscosity, in metres-squared per second

Description

Returns an approximation of the kinematic viscosity, ν, in units of $\mathrm{m^2\,s^{-1}}$, based on temperature, in units of °C. The kinematic viscosity is derived from a 3$^{\mathrm{rd}}$-order polynomial fit of ν

against T for salinity 0. The error of the approximation is less than 1 % for $0 \leq T \leq 20\,^{\circ}\mathrm{C}$ at atmospheric pressure.

(see also viscosity)

visc35

The kinematic viscosity of seawater for $S = 35$.

Syntax

```
v = visc35( T )
```

Argument	Description
`T`	temperature in units of degrees Celsius.
`v`	viscosity in units of metres-squared per second.

Description

Return an approximation of the kinematic viscosity, ν, in units of $\mathrm{m^2\,s^{-1}}$, based on temperature, in units of °C. The kinematic viscosity is derived from a 3$^{\mathrm{rd}}$-order polynomial fit of ν against T for salinity 35. The error of the approximation is less than 1 % for $0 \leq T \leq 20\,^{\circ}\mathrm{C}$ and $30 \leq S \leq 40\,\mathrm{PSU}$ at atmospheric pressure.

(see also viscosity)

viscosity

The kinematic and dynamic viscosity of sea-water

Syntax

```
[nu, mu] = viscosity( S, T, r )
```

Argument	Description
`T`	temperature in units of degrees Celsius
`S`	salinity in practical salinity units
`r`	density in units of kg per cubic-metre
`nu`	the kinematic viscosity in units of metres-squared per second
`mu`	the dynamic viscosity in units of kg per metre per second

Description

Calculates the kinematic viscosity, ν [$\mathrm{m^2\,s^{-1}}$], and dynamic viscosity, μ [$\mathrm{kg\,m^{-1}\,s^{-1}}$], of sea-water following Millero (1974). The range of validity is $5 \leq T \leq 25\,^{\circ}\mathrm{C}$ and $0 \leq S \leq 40\,\mathrm{PSU}$ at atmospheric pressure.

Check values are: `T = 25, S = 40, r(T,S): mu = 9.6541e-4`.

References:

- Millero, J. F., 1974, The Sea, Vol 5, M. N. Hill, Ed, John Wiley, NY, p. 3.
- Peters and Siedler, in Landolt-Bornstein New Series V/3a (Oceanography), pp 234.

A Data and Log File Structure

RSI data acquisition software produces at least two files - a data file and a log file. A second data file is generated, for real-time telemetering instruments, if the data acquisition software (ODAS-RT) is configured to access an external serial device.

A.1 Data File

Data files hold the unprocessed (raw) data observed by an instrument. These binary files are generated by the data acquisition software and are given the extension "`.p`".

A.1.1 Data File Format

Data files are structured as a series of records (Figure 2.2.1). The first record is a configuration string constructed by prepending a header to the configuration string. Subsequent records are data records constructed by prepending a header to a data block.

A.1.2 Header

Both configuration and data records start with a 128 byte header comprised of 64, 16-bit words. This header contains the information needed to correctly read the data file. The header from every record is saved into the matrix named `header` in your mat-file with every row representing a single header.

Most entries in the header are self explanatory, see table A.1.1, but some entries require the additional details described below.

- The date-time entries (items 4 – 10) are set to the time of the completion of a record. This time can fluctuate by about 0.2 s because the buffer status is checked about 5 times per second. The value is truncated to 0.001 s for a real-time instrument and to the nearest second for internally recording instruments.
- Item 16 indicates a "Bad Buffer" because the check of the special character for channel 255 failed. If the check failed, the next record will have the restart (item 17) set to 1. It can take about 0.2 s for the acquisition of data to restart.
- Items 21 and 22 give the actual aggregate sampling rate (see equation 1). It may differ from the requested rate if it is not a whole-number divisor of the 24MHz reference clock. If so, the data acquisition software selects the nearest rate. The reference clock is accurate to 1 part per million. The sampling rate of fast channels is the aggregate sampling rate divided by the number of columns in the address `[matrix]`. The number of columns in the address matrix equals the sum of items 29 and 30.
- Items 29 to 31 are the dimensions of the address `[matrix]`. They are used to demultiplex the data records into individual channels.

Table A.1.1: Description of fields that constitute a record header.

Location	Description
1	File number[1] - the three-digit number appended to the name of the data file.
2	Record number
3	Record number of the input on the RS-232 port, for real-time telemetering instruments (ODAS-RT only).[2]
4	Year
5	Month
6	Day
7	Hour
8	Minute
9	Second
10	Millisecond
11	Header version (MSB: major version, LSB: minor version)
12	Configuration string size in bytes
13	Product ID (0=legacy 1=odas5ir, 2=odasrt, 3=odas4ir)
14	Build number (unique revision number from SVN)
15	Time zone as minutes from UTC[1]
16	Buffer status (1 if special character check fails, 0 otherwise)
17	Restarted (1 if data acquisition was restarted due buffer status = 1. 0 otherwise)
18	Record header size in bytes (128)[1]
19	Data record size in bytes (header + data block)[1]
20	Number of records written to the current file.
21	Truncated frequency of the sampling clock (Hz)(see also equation 1)[1]
22	Fractional part of the frequency of the sampling clock (to 0.001 Hz)[1]
23-28	not used[1]
29	Number of fast columns in the address `[matrix]`[1]
30	Number of slow columns in the address `[matrix]`[1]
31	Number of rows in the address `[matrix]`[1]
32-62	not used[1]
63	Profile (0=Vertical, 1=Horizontal). Not used.
64	Data type[1] (0=unknown, 1=little endian, 2=big endian)

[1] These fields are the same for all records in a file.
[2] This field will be zero if the COM port is not used.

- The profile flag (item 63) indicates the direction an instrument travels. It is no longer used.
- The final item, 64, gives the endian format of the data file. Programs that read a data file directly should use this item to read the data correctly.

A.1.3 Configuration Record

The first record within a data file is the configuration record. The record consists of a header and a copy of the configuration file used for data acquisition. The copy is a single string of ASCII characters – the configuration string. The record number is always 0. The size of this record depends on the size of the configuration string.

To read the configuration record, open the data file and read the first 64 words (128 bytes).

Use the last word, item 64, to determine the endian format of the data file. Close the file and reopen it with the correct endian, if necessary. (The function `fopen_odas` does this in one call.) Use items 18 and 12 to read the first header and the configuration string.

A.1.4 Data Record

Data records consist of a header and a data block. They follow the configuration record. The header is described in section A.1.2 and the data block consists of unprocessed data from an instrument. Data words (samples of 2-byte numbers) are stored in the order provided by the address matrix. The number of words within a data block is a multiple of the size of the address matrix. The total size of a data record (header + data block) is stored in item 19 of the record header. All data records will be the same size.

To read the data records, start by reading the configuration record. The file pointer will be at the start of the first record. Use items 18 and 19 to read the header and data for the first data record. Continue reading data records until you reach the end-of-file.

A.2 Log-file

The log-file contains a record of the events that occurred during data acquisition. The log file is named `logfile.txt`. Example events from a log file are shown below:

```
50d8b024 0b 0000 dat_001.p --- Mon Dec 24 11:42:28 2012 - "acquisition start" Setup File:'H:/data/setup.cfg'
50d8b071 0c 004e dat_001.p --- Mon Dec 24 11:43:45 2012 - "acquisition stop" manual stop
50d8b09a 0b 0000 dat_002.p --- Mon Dec 24 11:44:26 2012 - "acquisition start" Setup File:'H:/data/setup.cfg'
50d8b1fd 0d 0162 dat_002.p --- Mon Dec 24 11:50:21 2012 - "bad buffer"
50d8b25e 0c 01c4 dat_002.p --- Mon Dec 24 11:51:58 2012 - "acquisition stop" manual stop
```

A.2.1 Log-file Format

Each line in a log-file describes a single event. The first part of each line is in a machine-readable form to support automated scripts. The second part of each line is in human-readable form.

Events are recorded using the following format:

```
          EVENT = {MACHINE_FORMAT} --- {HUMAN_FORMAT}

 MACHINE_FORMAT = {time} {event_type} {record#} {data_file_name}
   HUMAN_FORMAT = {human_time} - "{event_desc}" [additional_info]
```

The above fields are defined as follows:

Field	Description
`time`	Time of the event, as given by the UNIX "`time`" function. Value is in hexadecimal format and always uses the first 8 characters.
`event_type`	Type of event identified by a two digit hexadecimal number. See section A.2.2 for a list of event types.
`record#`	The record number during which the event occurred. Formatted as a 4-character hexadecimal value.
`data_file_name`	Name of the data file. If a data file has yet to be created, the value "`unknown`" is used.
`human_time`	Time of the event.
`event_desc`	The event type.
`additional_info`	Optional extra information for the event.

A.2.2 Event Types

The types of events that get recorded is instrument dependent (Table A.2.1).

Table A.2.1: Event codes that can be in a log file.

Hex Code	Event Name	Description
`01`	pressure release	Release triggered because the pressure exceeded `max_pressure` (see Table 3.5.2).
`02`	fall rate release	Release triggered because the fall rate was less than $0.2\,\mathrm{dbar\,s^{-1}}$ for 4 consecutive records.
`03`	timeout release	Release triggered because the duration of data acquisition exceeded `max_time` (see Table 3.5.2).
`04`	power is bad	Release triggered due to insufficient power-supply (battery) voltage. The "power-good" signal from the power-supply board went to low.
`05`	power good signal	The "power-good" signal from the power-supply board went high.
`06`	switch is off	The power ON/OFF switch was set to OFF.
`07`	manual stop	Acquisition was stopped manually by an operator.
`09`	maximum file size	Data file reached its maximum allowed size.
`0a`	out of disk space	Disk space was insufficient for data storage.
`0b`	acquisition start	Data acquisition was started. Name of the configuration file is provided for additional info.
`0c`	acquisition stop	Data acquisition had stopped.
`0d`	bad buffer	Record contains a bad buffer.

Index