ROCKLAND SCIENTIFIC

ODAS MATLAB Library

Version 3.1.1 (2013-02-26)

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

Contents

1 History **7**

2 Introduction **8**
2.1 What is new in Version 3.1? . . . 9
2.2 Data file structure . . . 10

3 Configuration Files **11**
3.1 Anatomy of a Configuration File . . . 11
3.2 Adding Values . . . 12
3.3 Extracting Values . . . 14
3.3.1 `setupstr` Examples . . . 16
3.3.2 Regular Expressions Explained . . . 18

4 Sections and Parameters **19**
4.1 `[root]` Section . . . 21
4.2 `[matrix]` Section . . . 23
4.3 `[channel]` Section . . . 25
4.4 Channel Names . . . 27
4.4.1 name = `Ax, Pitch, Tilt` . . . 28
4.4.2 name = `Ay, Roll` . . . 28
4.4.3 name = `Az` . . . 28
4.4.4 name = `C1` . . . 28
4.4.5 name = `C1_dC1` . . . 28
4.4.6 name = `C2` . . . 28
4.4.7 name = `C2_dC2` . . . 29
4.4.8 name = `Incl_T` . . . 29
4.4.9 name = `Incl_X` . . . 29
4.4.10 name = `Incl_Y` . . . 29
4.4.11 name = `O2_43F` . . . 29
4.4.12 name = `P` . . . 29
4.4.13 name = `P_dP` . . . 29
4.4.14 name = `SBC1` . . . 30
4.4.15 name = `SBT1` . . . 30
4.4.16 name = `Sh1` . . . 30
4.4.17 name = `Sh2` . . . 30
4.4.18 name = `T1` . . . 30
4.4.19 name = `T1_dT1` . . . 30
4.4.20 name = `T2` . . . 31
4.4.21 name = `T2_dT2` . . . 31
4.4.22 name = `V_Bat` . . . 31
4.5 Channel Types . . . 31
4.5.1 type = `accel` . . . 32
4.5.2 type = `gnd` . . . 32

4.5.3 type = `inclt` . . . 33
4.5.4 type = `inclxy` . . . 33
4.5.5 type = `magn` . . . 34
4.5.6 type = `o2_43f` . . . 35
4.5.7 type = `poly` . . . 36
4.5.8 type = `sbc` . . . 37
4.5.9 type = `sbt` . . . 38
4.5.10 type = `shear` . . . 39
4.5.11 type = `therm` . . . 40
4.5.12 type = `ucond` . . . 42
4.5.13 type = `voltage` . . . 42

5 Common Tasks **44**
5.1 Extract Configuration String . . . 44
5.2 Patch / Upgrade Data File . . . 44
5.3 ODAS conversion example . . . 46
5.4 ODAS deconvolution and conversion example . . . 46

6 ODAS Functions **48**
adis . . . 48
check_bad_buffers . . . 49
clean_shear_spec . . . 49
conductivity_TPcorr . . . 51
convert_odas . . . 52
csd_odas . . . 54
csd_rolf . . . 56
deconvolve . . . 56
despike . . . 59
extract_odas . . . 61
extract_setupstr . . . 62
fig2pdf . . . 63
file_with_ext . . . 65
fix_bad_buffers . . . 66
fix_underscore . . . 70
fopen_odas . . . 70
get_diss . . . 71
get_latest_file . . . 74
get_profile . . . 75
inifile_with_instring . . . 75
make_scientific . . . 76
nasmyth . . . 76
odas_update . . . 78
patch_odas . . . 79
patch_setupstr . . . 83
patch_tomi . . . 84

plot_HMP 85
plot_VMP 87
plot_XMP 89
psd_rolf 90
query_odas 90
quick_bench 91
quick_bench_XMP 95
quick_look 95
quick_look_HMP 103
read_odas 109
read_tomi 111
save_odas 112
save_rolf 113
setupstr 113
show_P 116
TI_correction 117
visc00 118
visc35 119
viscosity 119

List of Figures

1 Data Flow Chart . . . 8
2 Old and new style data format . . . 10
3 Coherent noise removal applied to shear probe spectrum. . . . 51
4 Deconvolution example. . . . 58
5 Deconvolution example two. . . . 58
6 The rate of change of pressure derived from the normal pressure signal and the high-resolution pressure signals using the gradient function. . . . 59
7 Despike function . . . 61
8 Despike with zoomed in plot. . . . 61
9 Corrupted ODAS binary data file . . . 68
10 Corrupted ODAS binary data file after being repaired. . . . 69
11 Example data file with four bad records . . . 81
12 Repaired data file . . . 82
13 Example output produced by plot_HMP. . . . 86
14 Example output from plot_VMP. A detailed explanation of the plot is found within the function description. . . . 88
15 The time-series output from the quick_bench function. . . . 93
16 quick_bench spectra output . . . 94
17 Quick_look example for pitch and roll. . . . 98
18 Quick_look exmple plot of Sea-Bird data. . . . 99
19 Quick_look example of polynomial regression of the thermistor against the Sea-Bird thermometer. . . . 100
20 Quick_look example showing vertical gradients of the micro-structure signals. 101
21 Quick_look example spectra plot. . . . 102
22 Quick_look example spectra plot also showing the Nasmyth spectra. . . . 103
23 quick_look_HMP - example #1 . . . 104
24 quick_look_HMP - example #2 . . . 105
25 quick_look_HMP - example #3 . . . 106
26 quick_look_HMP - example #4 . . . 107
27 quick_look_HMP - example #5 . . . 108
28 quick_look_HMP - example #6 . . . 109

List of Tables

4.1.1 Required parameters for section `[root]`. 21
4.1.2 Optional parameters for section `[root]`. 22
4.2.1 Parameters for section `[matrix]` 23
4.2.2 Channel IDs and the sensors and signals that they commonly identify. 25
4.3.1 Parameters for section `[channel]` 26
4.4.1 Instrument required RSI defined channel names. 27
4.4.2 RSI defined channel names. 27
4.5.1 Channel types used for converting raw data into physical units. 31

Listings

1 Partial configuration file example 13
2 Example Configuration File 20
3 Example channel section for accel. 32
4 Example channel section for gnd. 33
5 Example channel section for inclt. 33
6 Example section for inclxy. 34
7 Example section for “magn” type. 35
8 Example channel section for o2_43f. 36
9 Example channel section for poly. 37
10 Example channel section for sbc. 38
11 Example channel section for sbt. 39
12 Example channel section for shear. 40
13 Example channel section for therm. 41
14 Example section for ’ucond’ type. 42
15 Example section for ’voltage’ type. 43

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

2 Introduction

The ODAS Library of Matlab functions provides the tools to process data collected with all microstructure 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.

The flow of information, from raw data to products that are useful for scientific work, is depicted in Figure 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 (ODASRT, ODAS4-IR, or ODAS5-IR). The result is an ODAS date file with a base-name that is determined by the 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 and 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.

There are two types of function. Type-A functions are those that will be used by everyone. For example, the conversion of a raw "p"-file into a mat-file is achieved with a type-A function. Type-B functions are special functions that we, at RSI, use to evaluate our instruments. Type-B functions can provide you examples of how to use the Type-A functions. This guidance accelerates the development of your own processing functions that you use to obtain specific scientific information according to your own interests, ideas and methods. An example of a Type-B function is one that derives the rate of dissipation of kinetic energy – there are many ways to do this and everyone has their preferred method.

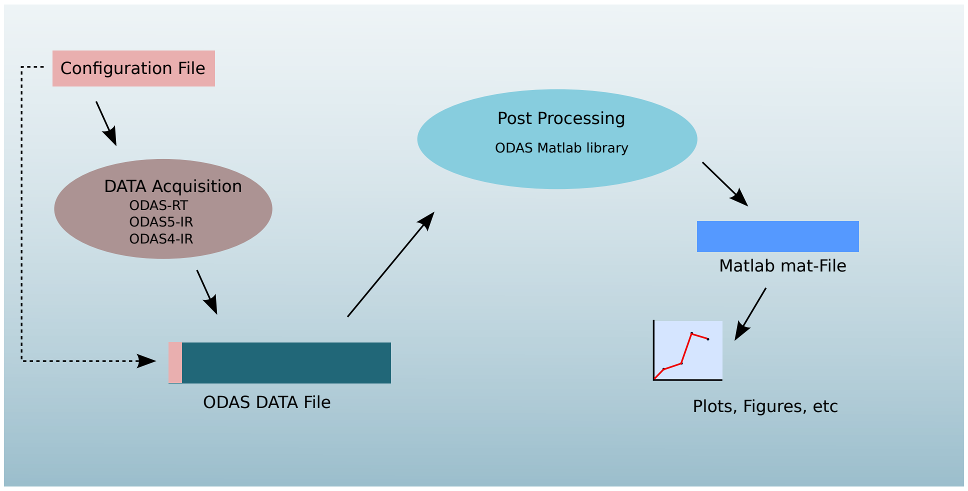

Figure 1: Data Acquisition and Post Processing Overview

2.1 What is new in Version 3.1?

We have:

- revised the discussion of the configuration-file to clarify its usage and provide detail examples,
- modified the functions that query the configuration-file string in the data file to make it easier to extract information,
- added new functions that allow you to modify the configuration information and to convert older-style (version-1) data files into the newer-style (version-6) data files, so that you can take advantage of the features that are available for the newer data files, and
- increased the speed of two important, and much used, functions (read_odas.m and plot_VMP.m) by factor of 100 to significantly reduce the processing time.

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 work with version-1 files. Also, version-1 files can be converted to version-6 files, using the function "patch_setupstr". We recommend highly that you convert the old-style version-1 files to version-6 files. In the future, version-1 files may not be supported for purposes other than conversion.

The internal structures of the old-style (version-1) raw binary data files and the new version -6 data files are illustrated in Figure 2. 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 properly configured. Therefore, the next sections provides details about the configuration file.

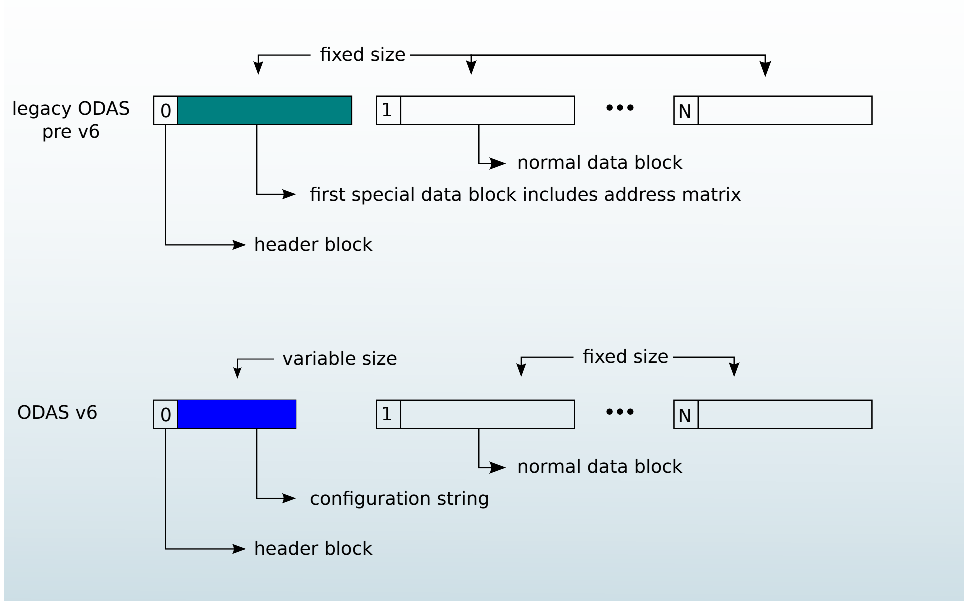

Figure 2: Old and new style data format. The configuration string is a copy of the configuration file used during data acquisition.

3 Configuration Files[1]

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 configure itself depending on the contents of the 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 - this is explained in the following paragraphs.

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 date and location of a profile can be added to a configuration file. When processing the resulting data files, reference to the date and location will be available and can be automatically extracted by your Matlab scripts.

RSI data acquisition software stores collected data within an RSI data file. All recent software (early 2013) produce v6 RSI data files that embed a copy of the configuration file used during data acquisition. As a result, information added to the configuration file will be included within the resulting data files. When processing the data files, this configuration file is extracted and stored as a *configuration string* - typically as the Matlab variable named "`setupfilestr`".

Contents of a configuration string are made accessible through use of the `setupstr` function. This function scans a configuration string for structured information and returns values that match a provided query. Configuration strings embedded within a data file can be modified using the `extract_setupstr` and `patch_setupstr` functions. This allows users to correct for some configuration file errors present at the time of data acquisition. These function, and some suitable examples, are described in the ODAS Matlab library manual.

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*.

[1]This section assumes you have some knowledge of Matlab and the ODAS Matlab library.
[2]Configuration files can only contain 7-bit characters - values from 0x00 to 0x7F.

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
```

Each line can have a maximum of one parameter with the first equal sign separating the name from the value. Extra white-space surrounding the name or value is ignored. As such, "`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 *section name* string enclosed by square brackets (`[ ]`).

```
[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 parameter named "`name`" should be unique. If a section does not contain a *name* parameter, the section name should be unique because it will be used as the section identifier.

Parameters declared before the first section declaration are defined as being in the *root* section. To extract these parameters, users should reference them as being in section "`root`".

Comments

All characters after a semicolon (`;`) are considered *comments* and ignored. These comments are still valuable and can assist people in understanding the meaning of the parameters and their values within a configuration file.

```
; 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. In addition, RSI instruments expect certain sections and parameters to exist so care should be taken when modifying a configuration file.

To ensure configuration files will work with an RSI instrument, it is recommended that users add parameters into newly created, uniquely named, sections. Parameters associated with existing sections, such as "`SN=M358`" in the following example, can be added so long as the parameter name is not used by RSI. The following example depicts a portion of a configuration file that has been modified by a customer 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 4.3.1 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 the section can be referenced. The name parameter acts as the section identifier.

To find the value of parameter name "`SN`" in the above example, the following query can be performed:

```
>> 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 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 3.3.

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 name.
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: 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.

Note: When section and parameter inputs are given as empty strings, (`''`), the `setupstr` function matches all values.

Note: 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 `'sections'` 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'
>> date = setupstr( obj, 'cruise', 'date' );
date =
    'April 14, 1912'
```

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. Note how 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 a user provided value so the function will only work if the user has added this value to their configuration file.

The query is 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.

3.3.2 Regular Expressions Explained

The `setupstr` function interprets section and parameter arguments as regular expressions. To avoid confusion, one should avoid the use of *metacharacters* in their section and parameter names. A list of metacharacters used by Matlab is shown in the "`regexp`" function documentation; type "`help regexp`" at the Matlab command prompt. As a general rule of thumb; all characters, numbers, and horizontal lines are safe.

To ensure exact matches to queried strings, the `setupstr` function automatically adds the "start of string" character "`^`" and "end of string" character "`$`" to every query. This ensures exact matches for those not familiar with regular expressions. Those familiar with regular expressions should take this into account when performing their own queries.

4 Sections and Parameters[3]

RSI uses many pre-defined sections and parameters to configure instruments and to allow for data conversion. Care must be taken to ensure user supplied sections and parameters do not conflict with those used by RSI. In addition, modifying RSI supplied sections and parameters may also be required. In this case, the following sections should be used as a guideline to ensure correct modifications are made.

[3]This section assumes you have some knowledge of Matlab and the ODAS Matlab library.

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 = 9
no-slow = 2

[pres]
id=    10
type=  poly
name=  P
units= [dbar]
coef0= 3.67
coef1= 0.062076
coef2= 4.724e-8
coef3= 0

[channel]
id=4
type=therm
name=T1
adc_fs=4.096
adc_bits=16
a=-6.4
b=0.99854
G=6
E_B=0.68247
beta=3143.55
T_0=289.301
units=[^\circ C]

[matrix]
num_rows=8
row01=255  0  1 2 3 5 7 8 9 12 13
row02=  4  6  1 2 3 5 7 8 9 12 13
row03= 10 11  1 2 3 5 7 8 9 12 13
row04= 16 17  1 2 3 5 7 8 9 12 13
row05= 18 19  1 2 3 5 7 8 9 12 13
row06=  4  6  1 2 3 5 7 8 9 12 13
row07=  0  0  1 2 3 5 7 8 9 12 13
row08=  0  0  1 2 3 5 7 8 9 12 13

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

4.1 `[root]` Section

The root section is not declared, but implied. It consists of all parameters not defined within an existing section. In listing 2, the root section starts with parameter "`rate`" and continues to section "`[pres]`". 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 software. Currently, the required parameters are listed in table 4.1.1;

Table 4.1.1: 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 4.2.
`disk`	full path to the directory where the data files are to be written. Use forward slashes ("/") to delimit directories regardless of operating system.
`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. Some operating systems, such as PICO-DOS, 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 because records must align with 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) connection speed between the UTRANS and instrument. 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 only exception to this rule is a value of 0 which is set to a rate of 2.0 Mbit/s. Most instruments are configured to use "3".

Optional `root` parameters are listed in table 4.1.2.

Table 4.1.2: 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 operating time (s) since the start of data acquisition after which the weight release is activated. This is used on instruments as a redundant condition for releasing a weight.
`profile`	direction which the instrument moves when collecting data. Possible values are *vertical* (default) and *horizontal*. This is used by some processing routines to orient data plots.
`brate`	(ODAS-RT) speed 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 the instrument will continue operating after the release mechanism is triggered. Default value of 0 implies the instrument will not stop.
`max_time`	(ODAS-IR) maximum profile duration before which the release is engaged.
`max_pressure`	(ODAS-IR) maximum pressure allowed for a profile. If exceeded, the release is engaged.

4.2 [matrix] Section

The matrix section defines the channel 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]", the frequency at which they are sampled.

A minimum of two parameters are required to define a matrix. One parameter provides the number of rows while the others each define a single row. The required parameters are detailed in table 4.2.1.

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 value 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 4.2.1: Parameters for section [matrix]

Instruments sample one row at a time, returning to the first row when they reach the bottom. 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, each individual row will be sampled 64 times a second.

Instruments traverse each row from left to right sampling the channels in sequence. When sampling the first row in listing 2, the channels 255, 0, 1, 2, 3, 5, 7, 8, 9, 12, and 13 will be sampled in the provided order. The duration between samples is made constant and set using a high-precision crystal oscillator. For the first row of the previous example, the time between any two adjacent samples will be,

$$\textit{channel period} = \left(\textit{rate}\left[\frac{\text{rows}}{\text{s}}\right]\cdot \textit{number of columns}\left[\frac{\textit{channels}}{\textit{row}}\right]\right)^{-1}.$$

For listing 2, this works out to be $(512 \cdot 11)^{-1} \approx 177.6\,\mu\text{s}$.

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, 9, 12, and 13 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" is defined as the number of fast channels within a single row – 9 in this example.

Channels that are not fast are called slow – they are not in each row of the channel matrix. As such, they are sampled at a lower sampling rate. This sampling rate can be calculated using the formula;

$$\textit{channel rate} = \frac{\textit{rate parameter}\left[\frac{\text{rows}}{\text{s}}\right]}{\textit{number of rows}\left[\frac{\text{rows}}{\text{matrix}}\right]} \cdot \textit{occurances in matrix}\left[\frac{\text{samples}}{\text{matrix}}\right].$$

For listing 2, channel 10 occurs once within the matrix resulting in a sampling rate of $\frac{512}{8} \cdot 1 = 64$ (samples/second). Channel 4 occurs twice and will have a sampling rate of $\frac{512}{8} \cdot 2 = 128$ (samples/second). The root parameter "`no-slow`" is defined as the number of slow channels within a single row. For this example, this value is 2.

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

The user must ensure all channels within the address matrix actually exist in their instrument. Requesting a channel that does not exist will result in corrupted data. However, a user can ignore a channel by excluding it from the channel matrix. Some of the common channel ids, and the signals to which they are associated, are listed in table 4.2.2.

ID	Possible Signals	Common Names
0	ADC ground signal.	Gnd
1	Accelerometer x-axis.	Ax, Pitch, Tilt
2	Accelerometer y-axis.	Ay, Roll
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 at the pressure transducer. Sometimes used for micro-conductivity sensor #1 with pre-emphasis.	PV, C1_dC1
13	Micro-conductivity sensor #2 with pre-emphasis.	C2_dC2
16,17	Sea-Bird SBE3F thermometer.	SBT
18,19	Sea-Bird SBE4C conductivity sensor.	SBC
32	Voltage of the main battery.	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
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
255	This special character should be present at the start of every channel matrix. It is used for error detection and correction.	not required

Table 4.2.2: Channel IDs and the sensors and signals that they commonly identify.

4.3 [channel] Section

Channel sections are used to describe a channel. Required parameters are used to reference the channel and to convert the raw channel data into physical units. Optionally, additional parameters can be added to enhance the plotting of data, such as the "`units`" parameter.

It should be noted that for backwards compatibility, the section name *channel* is not required. Previously, RSI software assumed each channel was given a specific name but this was both unnecessary and confusing. If you currently have a configuration file using alternative section names for channels, no changes are 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 4.5 for a complete list of additional parameters.

The required and optional parameters for a channel section are shown in table 4.3.1.

Parameter	Description
`id`	channel number. This value must be unique and is used within the matrix to identify the channel.
`name`	string reference to this channel. Both RSI instruments as well as the ODAS Matlab library have channel name requirements. Defined channel names can be found in table 4.4.2.
`type`	identifies how raw channel data should be converted into physical units. Possible values are listed in table 4.5.1. Depending on this value, additional parameters will be required. See section 4.5 for a complete list of additional required parameters.
`diff_gain`	defines the gain applied to a channel with pre-emphasis. This value is found within the instrument calibration report. These channels typically have the name "*X*_d*X*" where "*X*" is the channel name without pre-emphasis. This parameter is only used for channels with pre-emphasis.
`units`	(optional) string value for units. Used when generating plots using the ODAS Matlab library. Only ASCII characters should be used to ensure compatibility with RSI instruments. When special characters are required, users can use TEX typesetting commands.
`display`	(optional – ODAS-RT) boolean value that determines if a channel is displayed within the ODAS-RT gui. Default value is "`true`". Can also be declared as "`display in odasrt`".

Table 4.3.1: Parameters for section `[channel]`

4.4 Channel Names

Some RSI instruments may require channels to have to the following names. This allows for some data conversion allowing the instruments to, for example, perform a weight release at a specified depth. Table 4.4.1 defines the channel names some instruments require.

Table 4.4.1: Instrument required RSI defined channel names.

	Names	Description
4.4.12	`P`	Pressure transducer.

RSI instruments should operate and successfully gather data regardless of what most channels are named. However, the ODAS Matlab library does make assumptions as to the names of some of the channels. To minimize the work involved in data processing, it is recommended to use names from table 4.4.2 when possible.

Table 4.4.2: RSI defined channel names.

	Names	Description
4.4.1	`Ax, Pitch, Tilt`	Accelerometer, the X-axis
4.4.2	`Ay, Roll`	Accelerometer, the Y-axis
4.4.3	`Az`	Accelerometer, the Z-axis
4.4.4	`C1`	First micro-conductivity sensor
4.4.5	`C1_dC1`	First micro-conductivity sensor with pre-emphasis
4.4.6	`C2`	Second micro-conductivity sensor
4.4.7	`C2_dC2`	Second micro-conductivity sensor with pre-emphasis
4.4.8	`Incl_T`	ADIS inclinometer, temperature
4.4.9	`Incl_X`	ADIS inclinometer, x-axis
4.4.10	`Incl_Y`	ADIS inclinometer, y-axis
4.4.11	`O2_43F`	Sea-Bird `SBE43F` oxygen sensor
4.4.13	`P_dP`	Pressure transducer with pre-emphasis
4.4.14	`SBC1`	Sea-Bird `SBE4` conductivity sensor
4.4.15	`SBT1`	Sea-Bird `SBE3` thermometer
4.4.16	`Sh1`	First shear probe with pre-emphasis
4.4.17	`Sh2`	Second shear probe with pre-emphasis
4.4.18	`T1`	First thermistor
4.4.19	`T1_dT1`	First thermistor with pre-emphasis
4.4.20	`T2`	Second thermistor
4.4.21	`T2_dT2`	Second thermistor with pre-emphasis
4.4.22	`V_Bat`	Battery voltage monitor

4.4.1 name = `Ax, Pitch, Tilt`

RSI instruments typically have an embedded accelerometer[4] to indicate the pitch, tilt, or acceleration of the instrument. This channel is typically set to channel id 1 and is of type "`accel`". Different instruments can have different types of accelerometers so the channel calibration coefficients can differ. Older instruments typically have a linear accelerometer with the required calibration coefficients included in the calibration report. More recent instruments typically use a piezoelectric accelerometer and will have `coef0=0` and `coef1=1`.

4.4.2 name = `Ay, Roll`

RSI instruments typically have an embedded accelerometer to indicate the roll of the instrument. This channel is typically set to channel id 2 and is of type "`accel`". See section 4.4.1.

4.4.3 name = `Az`

Older RSI instruments have a linear accelerometer to measure vertical acceleration. This is typically set to channel id 3 and is of type "`accel`". Calibration coefficients are provided within the instrument calibration report.

4.4.4 name = `C1`

Newer instruments sample the micro-conductivity sensor both with and without pre-emphasis. The channel "`C1`" samples micro-conductivity without pre-emphasis. It is usually found on channel id 64 and is of type "`ucond`".

4.4.5 name = `C1_dC1`

The micro-conductivity signal with pre-emphasis is sampled as type "`ucond`". Depending on the instrument, this signal is either in channel id 65, or in channel id 12.

4.4.6 name = `C2`

A second micro-conductivity probe can be connected to an instrument. This one is on channel id 66 and is of type "`ucond`". See section 4.4.4.

[4]The x,y,z axes differ for horizontal and vertical profilers. See your instrument documentation for more details.

4.4.7 name = `C2_dC2`

See section 4.4.5. - channel id 67.

4.4.8 name = `Incl_T`

The ADIS16209 inclinometer provides a temperature signal. This signal is of type “`inclt`” and is on channel id 42.

4.4.9 name = `Incl_X`

The incline of an instrument (rotation around the x-axis) can be monitored using a ADIS16209 digital inclinometer. The x-axis is of type “`inclxy`” and is on channel id 40.

4.4.10 name = `Incl_Y`

Similar to the `Incl_X` channel, (section 4.4.9), but monitors the y-axis and is on channel id 41.

4.4.11 name = `O2_43F`

Oxygen can be monitored using a Sea-Bird SBE43F Dissolved Oxygen sensor. This channel will be of type “`o2_43f`” and is not located on any standard channel id. This sensor is similar to other Sea-Bird sensors in that two channels are used to transfer the signal. See section 4.4.14

4.4.12 name = `P`

The pressure transducer is typically of type “`poly`” and found on channel id 10. RSI instruments require the pressure channel to be named “P” and included within the channel matrix. Correct calibration coefficients are required if an RSI instruments is to properly utilize the pressure information to initiate a weight release.

4.4.13 name = `P_dP`

The pressure transducer signal with pre-emphasis is sampled on channel id 11. The channel is of type “`poly`”. See section 4.4.12.

4.4.14 name = `SBC1`

Sea-Bird SBE4C conductivity sensors can be connected to channel ids 18 and 19. Two channels are required because the sensor is sampled as a 32 bit value. The standard configuration places the lower 16 bits on channel id 18 and the upper 16 bits on channel id 19. The channel should be set to type "`sbc`", see listing 10 for an example channel.

4.4.15 name = `SBT1`

Sea-Bird SBE3F thermometers are typically connected to channel ids 16 and 17. As per section 4.4.14, a 32 bit value is returned. The channel should be set to type "`sbt`", see listing 11.

4.4.16 name = `Sh1`

Shear is measured using RSI shear probes. These probes measure the rate of change of transverse velocity (shear). The first shear probe is expected on channel id 8 and is of type "`shear`".

4.4.17 name = `Sh2`

The second shear probe is located on channel id 9 and is of type "`shear`". See section 4.4.16.

4.4.18 name = `T1`

Temperature is typically recorded using an RSI thermistor probe. The measurement is made using two channels, one with and one without pre-emphasis. The name "`T1`" refers to the channel without pre-emphasis, is located on channel id 4, and is of type "`therm`".

The instrument dependent calibration coefficients are provided in the instrument calibration report. This report does not include values for the thermistor probe. These values are either provided in a separate probe calibration sheet or determined by fitting the observed data against known values - typically from a Sea-Bird thermometer.

4.4.19 name = `T1_dT1`

The signal from thermistor #1 with pre-emphasis is the channel "`T1_dT1`". This signal is located on channel 5 and is of type "`therm`".

This channel differs from "`T1`" in that it requires the parameter "`diff_gain`" to be defined.

4.4.20 name = `T2`

RSI instruments can contain multiple thermistor probes. When two probes are provided, the second probe without pre-emphasis goes on channel 6 and is named "`T2`". See section 4.4.18 for a complete description.

4.4.21 name = `T2_dT2`

RSI instruments can contain multiple thermistor probes. The second probe, with pre-emphasis, goes on channel 7 and is named "`T2_dT2`". See section 4.4.19 for a complete description.

4.4.22 name = `V_Bat`

The battery voltage is monitored on the channel labelled "`V_Bat`". This channel is usually of type "`voltage`" and located on channel id 32. RSI instruments use this channel to monitor the on-board battery voltage.

4.5 Channel Types

In the following subsections, the variable N refers to the raw data value (counts) reported by an instrument. It is a 16-bit integer for all channels, except for two-channel sensors, such as Sea-Bird sensors, where it is a 32-bit integer.

Valid channel types are listed in table 4.5.1.

Section	Type	Description
4.5.1	`accel`	Linear accelerometer
4.5.2	`gnd`	Voltage at ADC ground
4.5.3	`inclt`	Temperature from an ADIS dual-axis digital inclinometer
4.5.4	`inclxy`	Angular data from an ADIS dual-axis digital inclinometer
4.5.5	`magn`	Data from a 3-axis magnetometer
4.5.6	`o2_43f`	Sea-Bird `SBE43F` dissolved oxygen sensor data
4.5.7	`poly`	Generic polynomial conversion of order ≤ 3
4.5.8	`sbc`	Sea-Bird `SBE4C` conductivity sensor data
4.5.9	`sbt`	Sea-Bird `SBE3F` thermistor data
4.5.10	`shear`	RSI shear probe data
4.5.11	`therm`	`FP07` thermistor data
4.5.12	`ucond`	RSI micro-conductivity data
4.5.13	`voltage`	Generic ADC voltage data

Table 4.5.1: Channel types used for converting raw data into physical units.

4.5.1 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;

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

The required parameters are as follows. See the example to view a completed channel section of type “`accel`”.

Parameter	Variable	Description
`coef0`	a_0	offset
`coef1`	a_1	slope

Listing 3: Example channel section for accel.

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

Note: The TEX formatted value for “`units`” will be rendered by Matlab using TEX when included within a plot.

The piezoelectric accelerometers in new instruments are linear but not calibrated. They should be assigned values of “`coef0=0`”, and “`coef1=1`”.

4.5.2 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. Please specify “`coef0`”, even though the value will get ignored. The returned value is in counts, just like the input value N.

$$Gnd = N$$

Parameter	Description
`coef0`	Value ignored.

Listing 4: Example channel section for gnd.

```
[channel]
id=0
type=gnd
name=gnd
coef0=1
units=[counts]
```

4.5.3 type = `inclt`

This type is used to convert the temperature data from an ADIS dual-axis digital inclinometer (see section 4.5.4).

The ADIS16209 inclinometer produces 16 bit words containing both data and status flags. Before converting data into physical units, the flags must be checked for errors and then stripped from the word.

The first two bits of the data word contain the status flags “new data” and “error” while the lower 12 bits contain the raw data as an unsigned integer. To convert this word to raw data, the function `adis` is used as follows;

$$T_{raw} = adis(N) \; .$$

This value can be converted into physical units with;

$$inclT = a_0 + a_1 \cdot T_{raw} \, [^{\circ}\mathrm{C}] \, .$$

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

Listing 5: Example channel section for inclt.

```
[channel]
id=42
type=inclt
name=Incl_T
units=[^{\circ}C]
coef0=624
coef1=-0.47
```

4.5.4 type = `inclxy`

This type is used to convert the angular data from an ADIS dual-axis digital inclinometer (see section 4.5.3)

The ADIS16209 inclinometer produces 16 bit words containing both data and status flags. Before converting into physical units, the flags must be checked for errors and then stripped from the word.

The first two bits of the data word contain the status flags “new data” and “error” while the lower 14 bits contain the raw data as a signed, two’s complement integer. To convert this word to raw data, the function `adis` is used as follows;

$$X_{raw} = adis(N) \ .$$

This value can be converted into physical units with;

$$inclX = a_1 \cdot X_{raw} + a_0 \, [\text{degree}] \, .$$

Similarly for `inclY`.

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

Listing 6: Example section for inclxy.

```
[channel]
id=40
name=Incl_X
type=inclxy
units=[^{\circ}]
coef0=0
coef1=0.025

[channel]
id=41
name=Incl_Y
type=inclxy
units=[^{\circ}]
coef0=0
coef1=0.025
```

4.5.5 type = `magn`

This type is used to convert data from a 3-axis magnetometer (such as PNI MicroMag3). The following formula, applies to all three axes, and provides the magnetic field in units of micro-tesla:

$$M = \frac{N - a_0}{a_1} \, [\mu\text{T}] \, .$$

The the heading, if desired, can be calculated with;

$$Heading = \frac{180}{\pi} angle(M_x + jM_y)\,[^\circ]\,.$$

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

Listing 7: Example section for "magn" type.

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

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

4.5.6 type = `o2_43f`

This type is used to convert data from a Sea-Bird SBE43F Dissolved Oxygen sensor and requires 4 coefficients.

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

From the Sea-Bird calibration of your sensor:

$$O_2 = 100 \cdot Soc\,(F_{offset} + f)\,[\%\ \text{saturation}]$$

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

Listing 8: Example channel section for o2_43f.

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

4.5.7 type = `poly`

This type applies a polynomial of order ≤ 3 to the channel data.

$$value = \sum_{i=0}^{3} a_i \cdot N^i \text{ [units sensor dependent]}$$

Parameter	Variable	Description
`coef0`	a_0	Least significant coefficient.
`coef1`	a_1	
`coef2`	a_2	
`coef3`	a_3	Most significant coefficient.

Listing 9: Example channel section for poly.

```
[pres]
id=10
units=[dbar]
type=poly
name=P
coef0=3.67
coef1=0.062076
coef2=4.724e-8
coef3=0

[channel]
id=88
units=[m^2]
type=poly
name=Junk
coef0=25
coef1=2.303
coef2=0
coef3=0
```

NOTE: You must supply all 4 coefficients.

In listing 9, the section name for the pressure channel is "`[pres]`". This name ensures the weight release on instruments with older firmware revisions will work. Instruments with updated firmware, and those without a weight release mechanism, can use "`[channel]`" as the section name. Please ensure you have the latest version of the ODAS software to ensure correct behaviour.

4.5.8 type = `sbc`

This type is used to convert data from a Sea-Bird SBE4C conductivity sensor and requires 7 coefficients.

Conductivity calculation:

$$f = \frac{n_p \cdot f_{ref}}{1000 \cdot N} \text{ [kHz]}$$

From the Sea-Bird calibration report of your sensor:

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

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

Listing 10: Example channel section for sbc.

```
[channel]
id_even=18
id_odd=19
units=[mS cm^{-1}]
type=sbc
name=SBC
coef0=-9.90912533e0
coef1=0
coef2=1.41415635e0
coef3=3.07973519e-4
coef4=5.32880619e-5
coef5=24e6
coef6=128
```

4.5.9 type = `sbt`

This type is used to convert data from a Sea-Bird SBE3F thermometer and requires 7 coefficients.

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

From the Sea-Bird calibration report of your sensor:

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

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

Listing 11: Example channel section for sbt.

```
[channel]
id_even=16
id_odd=17
units=[^{\circ}C]
type=sbt
name=SBT
coef0=4.35614236e-3
coef1=6.38892665e-4
coef2=2.11815756e-5
coef3=1.79451268e-6
coef4=1000.0
coef5=24e6
coef6=128
```

4.5.10 type = `shear`

This type is used to convert data from a shear probe sensors and requires 4 parameters. Conversion does not account for 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 is not in the odas data file. It has to be obtained from other sensors, such as the vehicle controller data file. To complete the conversion to physical units, you must divide *Sh* by the square of the speed of profiling.

$$Sh = \frac{N \cdot \frac{adc\_fs}{2^{adc\_bits}}}{2\sqrt{2} \cdot diff\_gain \cdot sens}$$

Parameter	Description
Parameters found in the Instrument Calibration report:	
`adc_fs`	Full scale voltage of the analog-to-digital converter, typically 4.096 or 5.0.
`adc_bits`	Number of bits in the analog-to-digital converter, typically 16.
`diff_gain`	Differentiator gain in seconds.
Parameters found in the Probe Calibration report:	
`sens`	shear probe sensitivity [V per m^2 s^{-2}] sensitivity.

Listing 12: Example channel section for shear.

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

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

4.5.11 type = `therm`

This type is used to convert data from an FP07 thermistor data into units of temperature, and requires 8 coefficients.

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

where N represents the raw data sample. The resistance ratio is then:

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

The final equation for Temperature in °C:

$$T = \left(\frac{1}{T_0} + \frac{1}{\beta} \cdot \ln\left(\frac{R_T}{R_0}\right)\right)^{-1} - 273.15\,[°\mathrm{C}]$$

Parameter	Description
Parameters found in the Instrument Calibration report:	
`a`	offset
`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 5.0.
`adc_bits`	Number of bits in the analog-to-digital converter, typically 16.
`diff_gain`	Only required for pre-emphasis channels - differentiator gain in seconds.
Parameters found in the Thermistor Calibration report:	
`beta`	Thermistor material constant
`T_0`	Temperature at which the resistance equals R_0

Listing 13: Example channel section for therm.

```
[channel]
id=4
type=therm
name=T1
units=[$^\circ$C]
a=-6.4
b=0.99854
G=6
E_B=0.68247
beta=3143.55
T_0=289.301
adc_fs=4.096
adc_bits=16

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

The above example is of a sensor that 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 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.

4.5.12 type = `ucond`

This type is used to convert data from a micro-conductivity data into physical units and requires 5 coefficients.

$$V_c = \frac{adc\_fs}{2^{adc\_bits}} \cdot N + \frac{adc\_fs}{2}$$

To obtain the conductance in units of Siemens:

$$Conductance = \frac{V_c - a}{b}$$

The final formula for conductivity in units of [mS cm^{-1}] is:

$$Conductivity = \frac{10 \cdot Conductance}{K} \left[\mathrm{mScm}^{-1}\right]$$

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 5.0.
`adc_bits`	Number of bits in the analog-to-digital converter, typically 16.
Parameters found in the Sensor Calibration report:	
`K`	sensor cell constant `[m]`

Listing 14: Example section for 'ucond' type.

```
[channel]
id=65
type=ucond
name=C1_dC1
units=[mS cm^{-1}]
a=1.216
b=196.7
K=1.03e-3
adc_fs=4.096
adc_bits=16
```

4.5.13 type = `voltage`

This type requires three coefficients and is used to convert voltage data into physical units of volts.

$$voltage = \frac{adc\_fs}{2^{adc\_bits}} \cdot \frac{N}{G}\ [\mathrm{V}]$$

Parameter	Description
`G`	Gain applied to signal before sampled by the analog-to-digital converter.
`adc_fs`	Full scale voltage of the analog-to-digital converter, typically 4.096 or 5.0.
`adc_bits`	Number of bits in the analog-to-digital converter, typically 16.

Listing 15: Example section for 'voltage' type.

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

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 presented to the analog-to-digital converter.

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 includes 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

Given a data file titled "`DAT_001.P`", the following code will save the embedded configuration string as a configuration file titled "`DAT_001.cfg`".

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

The following examples demonstrate three methods of calling the function and performing the same task. The examples extract the configuration string from the same data file and prints it to the screen.

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

Notice from the above examples how the file extension is optional. Note however that had the file been specified with a lower case extension, "DAT_001.p", then the function would fail and return a warning indicating that the requested file could not be found because the actual file name is "DAT_001.P".

When a data file is converted into a mat-file, the embedded configuration file is stored by the ODAS Matlab library in the variable named *setupfilestr*. If available, this variable can be referenced in place of using the `extract_setupstr` function.

5.2 Patch / Upgrade Data File

It is possible to patch a configuration file into an existing RSI data file. This is commonly done to make corrections to the calibration coefficients embedded within a data file. It is also used to upgrade a pre-v6 data file into a v6 data file. Once updated, users can take advantage of the modern data processing functions included with the ODAS Matlab library.

To perform the patch / upgrade of a data file, the ODAS Matlab library provides the `patch_setupstr.m` function.

Syntax

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

Only those parts of a configuration file used for post-processing data, such as calibration coefficients, can be changed. Parts that affect the structure of the resulting data file must not change and the function will return an error should this be attempted. For example, the channel matrix must never be changed. See section 6 for a list of elements that must not change.

Example - Patching a Data File

A data file named "`DAT_001.p`" and configuration file named "`corrected_config.cfg`" both exist in the current directory.

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

Patch the configuration file named "corrected_config.cfg" into the data file named "DAT_001.p". File extensions were not provided as they are optional. When not included, the `patch_setupstr` function will search for data files with either a "`.p`" or "`.P`" extension. Likewise, configuration file names are searched for matches with "`.cfg`" and "`.CFG`" extensions.

Example - Upgrading a Data File

Older data files can be upgraded to v6 data files by use of the `patch_setupstr` function. This is not a trivial task as it requires generating a new configuration file based on the previous setup file. 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.

```
%%  Assuming the 'DAT_002.cfg' file has been user generated.
>> patch_setupstr( 'DAT_002.P', 'DAT_002.cfg' );
```

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

5.3 ODAS conversion example

In this section we will show how ODAS Matlab Library uses the configuration file to convert the raw data samples into physical units, using the configuration-file example of Listing 2. See also the ODAS Matlab function reference for `convert_odas.m`.

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

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

where the vector to convert is the first argument in the function. The next argument is a string containing the "name". 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 try to find the one that has the parameter "`name=P`". It will then extract all calibration coefficients, the type-parameter, and the units-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.

The third argument, `'string'`, indicates that the calibration information is located in a string and the fourth argument, `setupfilestr`, is the string containing that information.

The final argument, `header_version`, indicates the format of the string containing the calibration information. This argument is optional and not used when processing legacy data files. For modern data files, this value is ≥ 6 and is typically found in the "`header_version`" variable produced by the ODAS library.

5.4 ODAS deconvolution and conversion example

Here we show how the channel entries for thermistor 1 (as shown in Listing 13) 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

$$T_1 + G_D \frac{\partial T_1}{\partial t}, \tag{1}$$

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 Matlab command,

```
>> T1_hres = deconvolve( 'T1_dT1', [], T1_dT1, fs_fast, ...
                         setupfilestr, header_version );
```

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`" as the vector `T1_hres`. The first argument is the name of the channel section in which the parameter "`diff_gain`" (G_D) is found. The second argument is only used for the pressure channel. For other channels, an empty matrix, `[]`, is used as a place-holder. The third argument is the vector to deconvolve. The fourth argument provides the sampling rate at which the vector should be deconvolved – the variable "`fs_fast`" is a standard variable in an ODAS mat-file. The fifth argument contains the configuration string from which to read parameters – the ODAS Matlab Library typically stores this in the variable "`setupfilestr`". The last argument is the header version and allows for the function to be backward compatible with the old-style data files. Its value is 6 and could have been entered explicitly.

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

```
>> T1_hres = convert_odas( T1_hres, 'T1', 'string', ...
                           setupfilestr, header_version );
```

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 for "`T1`", rather than "`T1_dT1`" because only the section for "`T1`" has the calibration coefficients for this thermistor. There are no redundant (or duplicate) coefficients in the section for "`T1_dT1`" (see Listing 13).

6 ODAS Functions

This section lists all functions that are part of the ODAS Matlab library. There are three types of functions:

1. Functions that work on raw binary data or on mat files (MATLAB binary file format containing saved variables). These are referred to as **Type A** functions.
2. Functions that are used by RSI internally to evaluate our instruments. Most functions in this category produce 2D plots. We call these **Type B** functions.
3. Depreciated functions - functions which should no longer be used. When used, they still work but display a warning. For each depreciated function, alternatives suggestions are provided within this document. These are called **Depreciated** functions.

adis – Type A

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.

check_bad_buffers – Type A

Find bad buffers within a data file

Syntax

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

Argument	Description
`file`	Name of file with or without the '.p' extension.
`badRecords`	Vector of bad record counts.

Description

Scan an ODAS 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 cooresponding data file is also flagged.

This fuction searches the headers within a ODAS binary data file for the bad buffer flag. The indexes of these records are collected and returned within the BadRecord value.

Examples

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

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

clean_shear_spec – Type A

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.

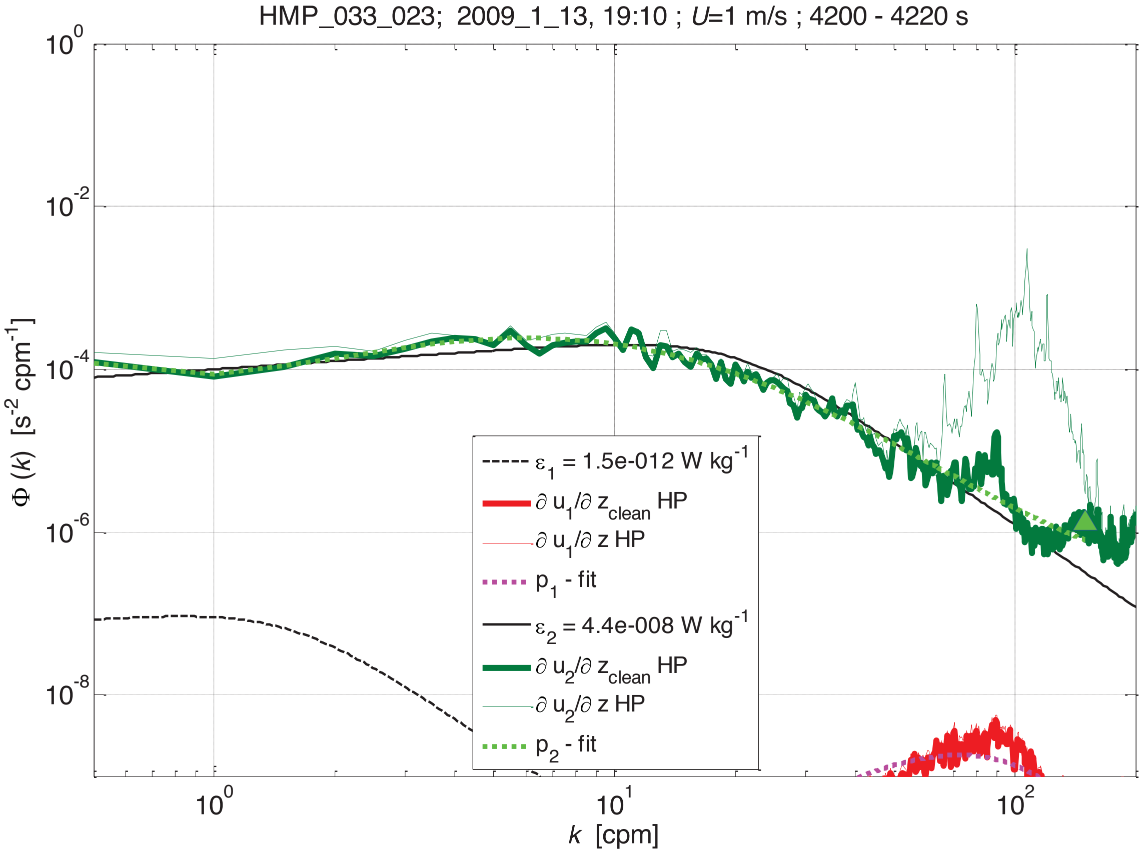

Figure 3: Original spectrum (thin green line) and after noise removal (thick green line). Data collected with Remus-600 in Buzzards Bay MA, courtesy of Tim Boyd , Scottish Association for Marine Science. Only shear probe 2 was installed, so, the spectrum for probe 1 falls off the deep end of the plot.

conductivity_TPcorr – Type A

Apply correction for thermal and pressure changes to 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 degrees Celsius.
`pres`	Pressure in units 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

A function to apply the small 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 – Type A

Convert from raw counts to physical units

Syntax

```
[phys, units] = convert_odas(var, sensor, coefSource, coef, ver, xmpSerNum)
```

Argument	Description
`var`	ODAS variable(s) in counts. Either a single vector or a matrix with a single variable in each column.
`sensor`	String array of names for the included ODAS variables. Names should match the channel names given in the setup / configuration file.
`coefSource`	Location of the coefficients (optional). Set to 'default' (use values within the routine), 'file' (load from setup file) or 'string' (load from configuration string). Most v6 users should use 'string' while v0 users will use 'file' or 'default'.
`coef`	Coefficient values. When coefSource is set to 'file', this is the name of the setup file. When coef_source is 'string', this is the variable that contains the configuration string (e.g., 'setupfilestr').
`ver`	Header version - always 1 for ODAS versions prior to v6.
`xmpSerNum`	String identifying the XMP serial number. Only used with XMP instruments. Should be of the form 'xmp_nnnnn' where nnnnn is the instrument's serial number with leading zeros. For example, 'xmp_00012'.
`phys`	Variable(s) converted into physical units.
`units`	String representation of the converted variable units.

Description

Converts ODAS data from recorded raw counts into physical units. Requires data as raw counts (var), the channel type used to collect the data (sensor), and the calibration coefficients needed to perform the conversion. For v6 data files, the calibration coefficients are found within the configuration string embedded within the data file. For previous v1 data files, one can use the default calibration coefficients found within this function, or calibration coefficients contained within a setup file. However, only a limited number of signals can be converted when using v1 data files.

Returns the data converted into physical units as double values along with the observed units as string values.

Examples (ODAS >= v6)

```
>> T2 = convert_odas(T2, 't2', 'string', setupfilestr, 6.0)
```

Convert data within 'T2' using coefficients from section 't2' found within the configuration file 'setupfilestr'. Notice that the second argument, 't2', will work with or without capital letters.

```
>> P = convert_odas(p, 'P', 'string', setupfilestr, 6.0, 'xmp_00012')
```

Convert a pressure vector from an XMP style instrument into physical units.

For more examples and detailed information regarding conversion from Raw data to physical units, please see section named 'Converting into Physical Units'.

Examples (ODAS == v1)

```
>> SBT1 = convert_odas( sbt1, 'sbt' )
```

Convert vector sbt1 into physical units using the Seabird temperature formula. Coefficients are not supplied so default values from within the function are used.

```
>> [SBC1, cUnits] = convert_odas( sbc1, 'sbc', 'file', 'setup.txt' )
```

Convert vector sbc1 into physical units using the Seabird conductivity formula. Coefficients will be extracted from the provided 'setup.txt' file. Both the converted values and the units of these values are returned.

```
>> [SBT1,SBC1,units] = convert_odas([sbt1 sbc1], {'sbt','sbc'}, 'default')
```

Multiple vectors can be supplied and converted in one call. The input vectors '[sbt1 sbc1]' are processed using their respective 'sbt' and 'sbc' types. Default coefficients from the convert_odas.m file are used for the conversion.

csd_odas – Type A

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

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 does not support the linear detrending of each segments 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:

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

csd_rolf – Depreciated

Legacy function replaced by csd_odas

Description

This is a legacy function. Please replace with the function 'csd_odas'.

This function calls 'csd_odas' using:

```
  [Cxy, Fs] = csd_odas( x, y, nFFT, rate, [], nFFT/2, 'linear' )
```

deconvolve – Type A

Deconvolve signal plus derivative to produce high resolution data

Syntax

```
xHires = deconvolve( dataType, X, XdX, fSample, gain, ver, xmpSerNum )
```

Argument	Description
`dataType`	String describing the data type to be deconvolved. For legacy ODAS and XMP, it can be one of 'pressure', 'temperature', or 'conductivity'. For ODAS v6 and up, this string identifies the channel to be deconvolved - a channel containing a 'diffGain' key. The channel is identified through the 'name' parameter or, when not found, the section title.
`X`	Low-resolution signals. Can be empty for data types other than pressure.
`XdX`	Pre-emphasized signals.
`fSample`	Sampling rate of the data in Hz.
`gain`	Differentiator gain as either a numeric value or as the configuration string from which it can be found.
`ver`	ODAS header version.
`xmpSerNum`	String identifying the XMP serial number. Only used with XMP instruments. Should be of the form 'xmp_nnnnn' where nnnnn is the instrument's serial number with leading zeros. For example, 'xmp_00012'.
`xHires`	Deconvolved, high-resolution signal.

Description

Deconvolve 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.

Examples

```
>> pHires = deconvolve( 'pressure', P, P_dP, 32, 20.5 )          (legacy)

>> tHires = deconvolve( 'temperature', [], T_dT, 512, 1)         (legacy)

>> pHires = deconvolve( 'pressure', P, P_dP, fs_slow, setupfilestr,
                        header_version, xmp_ser_num)           (xmp only)

>> t1Hires = deconvolve( 'T1_dT1', [], T1_dT1, fs_fast, setupfilestr,
                         header_version);                (odas v6 and up)
```

For temperature and conductivity, the initial conditions are estimated from the pre-emphasized signal only. Therefore, the vector X does not have to be passed to the function and can be given by '[ ]'.

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 seconds, 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 is provided in the calibration report of your instrument.

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

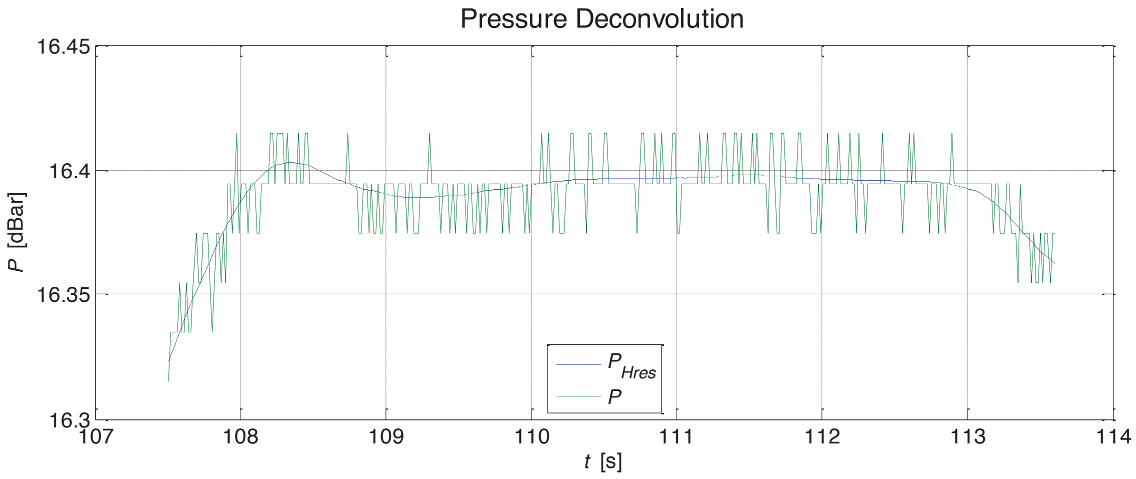

Figure 4: The green curve is from the normal pressure channel, and the blue curve is derived from the enhanced pressure channel. This data is from a profiler that has impacted the soft bottom of a lake. Both signals are shown with full bandwidth (0 - 32 Hz) without any smoothing. The full-scale of the pressure transducer is 500 dBar.

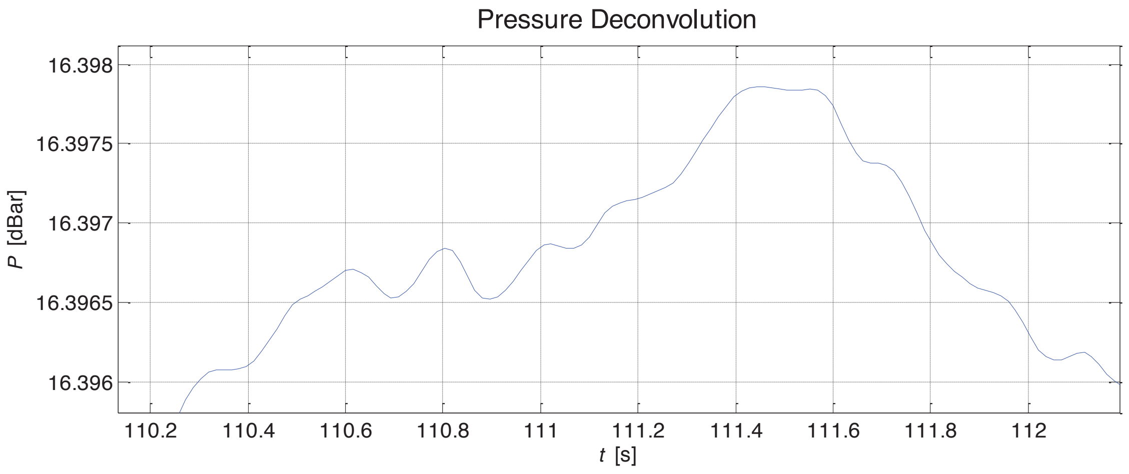

Figure 5: Same as previous figure but with zoom-in on only the high-resolution pressure. Again, full bandwidth without any smoothing.

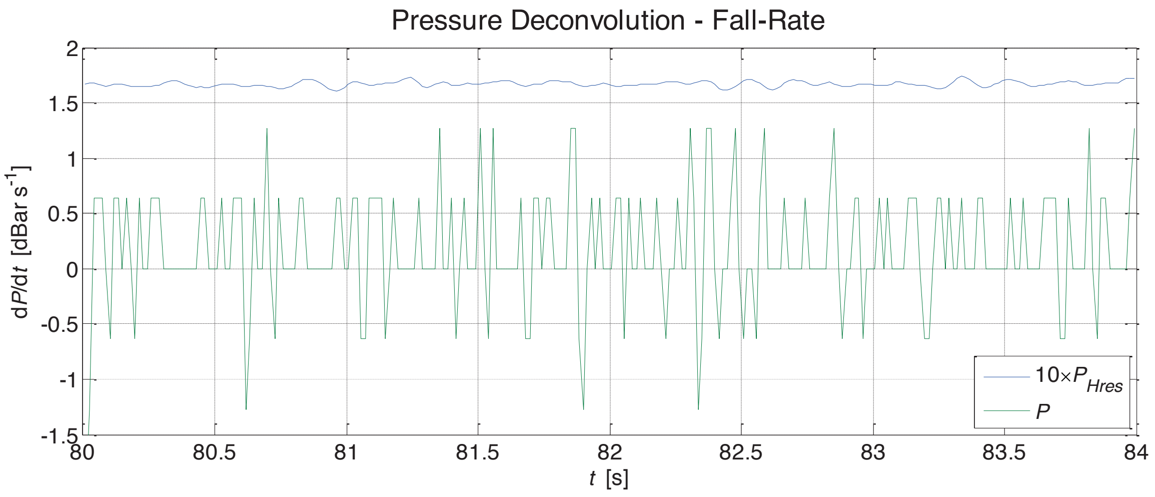

Figure 6: Full bandwidth signals. The normal pressure signal produces a fall-rate that is useless without extensive smoothing. It even has negative values, which means that the normal pressure record is not even monotonic. The rate of change of the high-resolution pressure is smooth, always positive, and the high-resolution pressure, itself, can be used directly for plotting other variables as a function pressure (or depth). Notice that the high-resolution rate of change of pressure has been multiplied by 10 for visual clarity. The fall-rate is about 0.17 m/s.

despike – Type A

Remove short-duration spikes from a signal.

Syntax

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

Argument	Description
`dv`	Signal to be despiked.
`thresh`	Threshold value for ratio of instantaneous signal to rectified, smoothed signal. A value of 7 is a good starting value. (default: 5)
`smooth`	Inverse smoothing time-scale (Hz) for the low-pass filtered that is applied to the high-pass and rectified version of dv to derive an estimate of the standard deviation of dv. (default: 0.5)
`Fs`	Sampling frequency (Hz). (default: 512)
`N`	Half-width of a spike. The amount of data that gets replaced is 2N+1, with half on each side of the spike. The replaced data equals the local mean which is estimated after exclusion of the identified spikes. (default: 5)
`y`	Despiked signal.
`spike`	Index to the identified spikes. It's length is a useful indicator of the function's veracity. (optional)

Description

Designed to despike all kinds of signals. It identifies spikes by comparing the instantaneous rectified signal against its local standard deviation. It obtains a measure that approximates the local standard deviation by:

- high-pass filtering the input signal,
- rectifying it (absolute value), and
- smoothing it with a low-pass zero-phase filter.

Based on a program originally written for shear probe despiking (enhance.m).

Examples

```
>> [y, spike] = despike(dv)
>> [y, spike] = despike(dv, thresh)
>> [y, spike] = despike(dv, thresh, smooth)
>> [y, spike] = despike(dv, thresh, smooth, Fs)
>> [y, spike] = despike(dv, thresh, smooth, Fs, N)
```

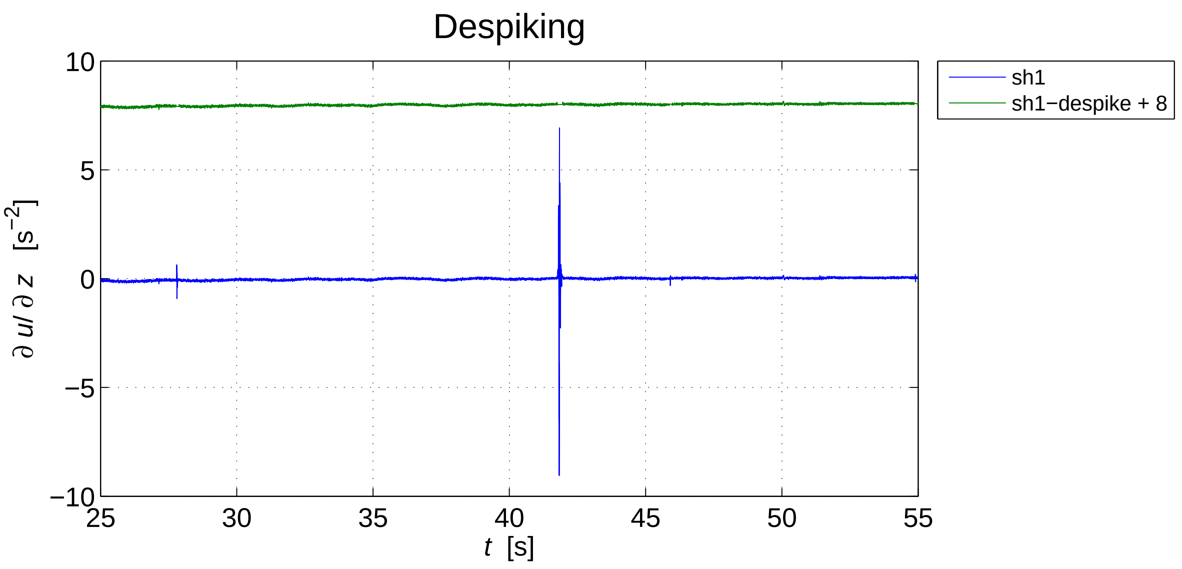

Figure 7: applied with despike(sh1, 7, 0.1, 512, 30). The smoothing scale, the inverse of 0.1 Hz, is long because this profiler was moving at only 0.17 m/s.

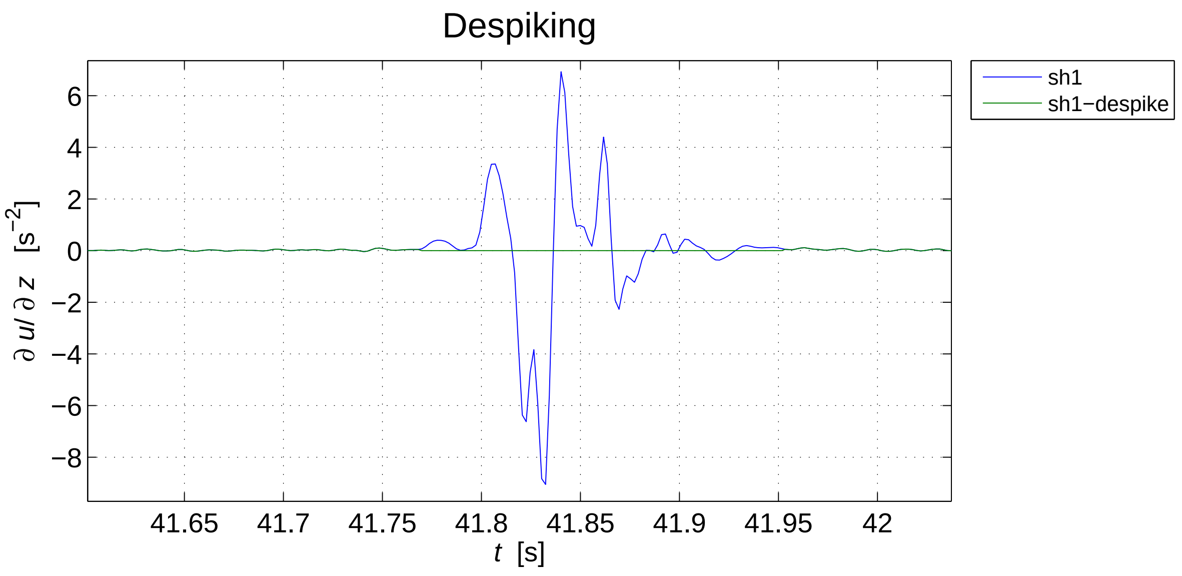

Figure 8: The same plot as the previous figure but with a close-up view.

extract_odas – Type A

Extract a range of records from a binary ODAS file

Syntax

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

Argument	Description
`fileName`	Name of the binary data file with or without the '.p' extension.
`startRecord`	Record number from which to start copying records.
`endRecord`	Record number of last record to be copied into the new file.
`newFile`	Name of the file created by this function. The first record is the first record of the original file because this record is special. Prior to version 6, it contains a copy of the address matrix. In versions >= v6, it contains the setup file string. The second record number is record 'startRecord' and the last is record 'endRecord'.
`successFlag`	Integer representing status. 0 if operation successful, otherwise <= -1. This flag can be used to issue warning messages, etc.

Description

Extract a range of records from a binary ODAS 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 collected with internally recording instruments can be very long and challenge the processing power of computers. This function provides a means to segment a file into shorter pieces to speed 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 an ODAS 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 for easier processing. The user previously determined that the data file contained relevant information 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.

extract_setupstr – Type A

Extract configuration string from v6 data file.

Syntax

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

Argument	Description
`dataFileName`	Name of a version >= 6 data file. (extension optional)
`configFileName`	Name of a configuration file that will be created and populated with the extracted configuration string.
`configStr`	Resulting configuration string.

Description

Extract the configuration string from a 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 constants used during data analysis. 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".

fig2pdf – Type B

Convert figures into PDF files with better quality then exporting from Matlab.

Syntax

```
fig2pdf( 'figure', 'fileName', [dimensions], legendPosition )
```

Argument	Description
`figure`	Either a string representing a .fig file or a figure object.
`fileName`	Output eps/pdf file names. Only used when 'figure' is a figure object.
`dimensions`	Width and height meadured in inches. Optional, omit or leave empty to use the default value of [8,6].
`legendPosition`	Position of the legend. An empty (") string uses the position defined by the .fig file. Default value = 'NorthEastOutside'

Description

Convert a figure object/file to an appropriately scaled PDF file. Avoids use of the Matlab PDF export function as it does not generate attractive plots. Instead, this function exports to an EPS file then converts the EPS file into the final PDF file. The PDF file maintains the same bounding box as the exported EPS file.

Axes set to font size 10 and title set to font size 12. To make the fonts larger/smaller - adjust the dimensions accordingly. Smaller dimensions make the fonts appear larger.

Legend is positioned in the 'OutsideNorthEast' corner by default.

The resulting graph is saved as a vector so you don't have to worry about the dimensions not being correct. Image will be attractive even when scaled. Plotting is performed with the "-painters" option.

Resulting PDF files have the same name as figName. Old files will be overwritten. Figures can have multiple sub-plots so long as they are arranged vertically.

Note: LaTeX should be installed on the computer. This function requires access to the "ps2pdf" program included with most LaTeX distributions.

Examples:

```
>> figure
>> x = 0:0.1:8*pi;
>> plot( [sin(x); cos(x)] )
>> fig2pdf( gcf, 'my_plot' )
```

Generating vector images (PDF) from the current figure. The final output will look the same regardless of the platform used to generate the plot.

```
>> fig2pdf( 'some_figure_file' )
>> fig2pdf( 'some_figure_file', [8,4], 'SouthEastInside' )
```

```
>> fig2pdf( 'some_figure_file', [], 'SouthEastInside' )
```

Generate PDF versions of the figure named 'some_figure_file.fig'. Use of '.fig' files allows this function to be used in scripts. Because the function produces the same images regardless of platform, the scripts simplify working with multiple platforms.

The second function call modifies the image size and legend position while the third fuction call modifies the legend position but uses the default [8,6] image size.

file_with_ext – Type A

Find file with default extension

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 file with a default extension. 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 – Type A

Fix bad records in ODAS data file

Syntax

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

Argument	Description
`fileName`	Full name of a binary odas 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 ODAS 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. If the file ends with a bad record(s) then those records are truncated. In addition, starting records are truncated if they have bad record(s). If a bad record has already been fixed it leaves that record unchanged.

(see also patch_odas.m)

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

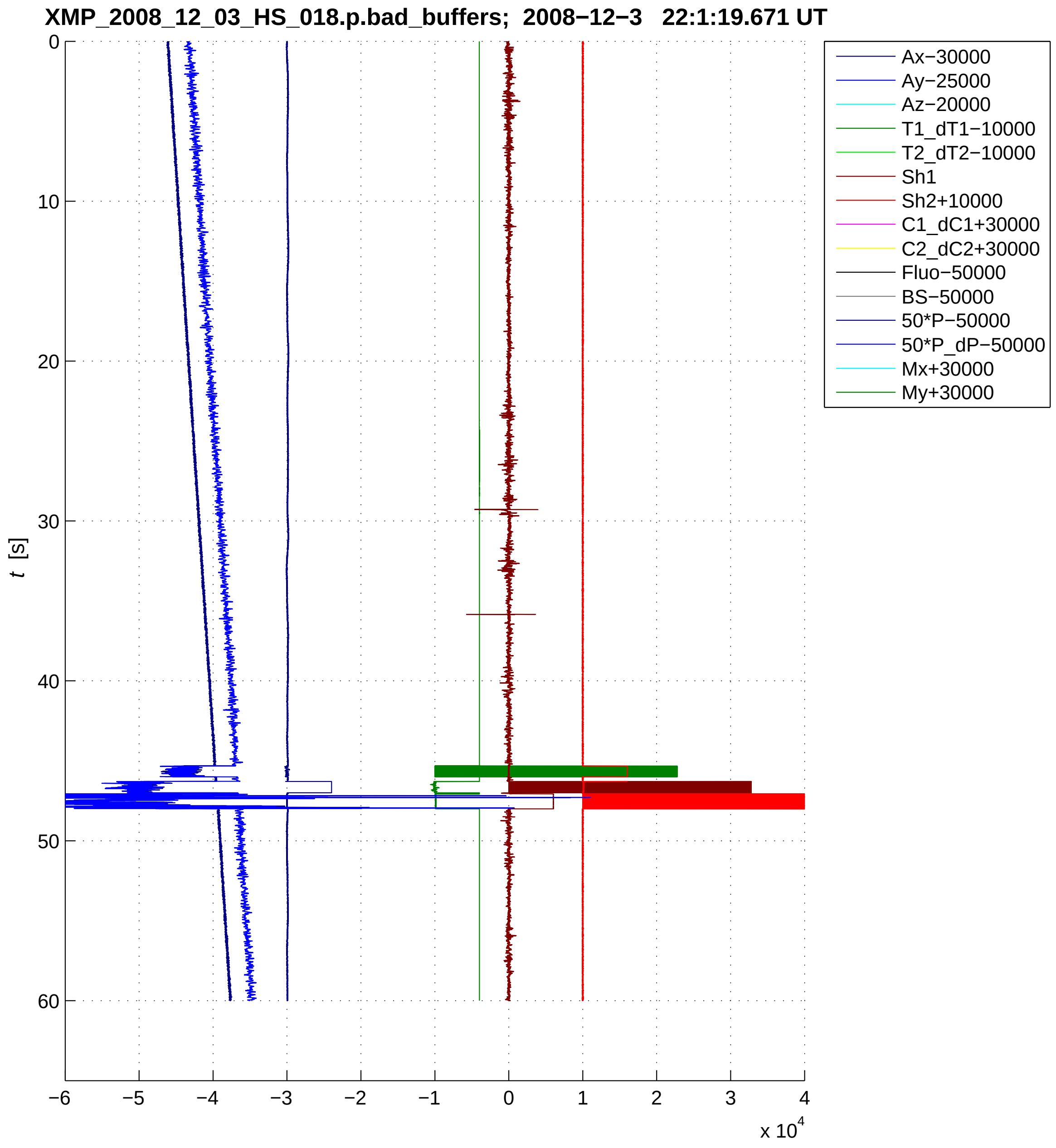

Figure 9: that has 'Bad Buffers' from about 45 to 48 seconds. This sample is from an Expendable Microstructure Profiler (XMP) which was notorious for communication failures in the early stages of its development. Records 45 through 48 are skewed and appear garbled as a result of communication failures.

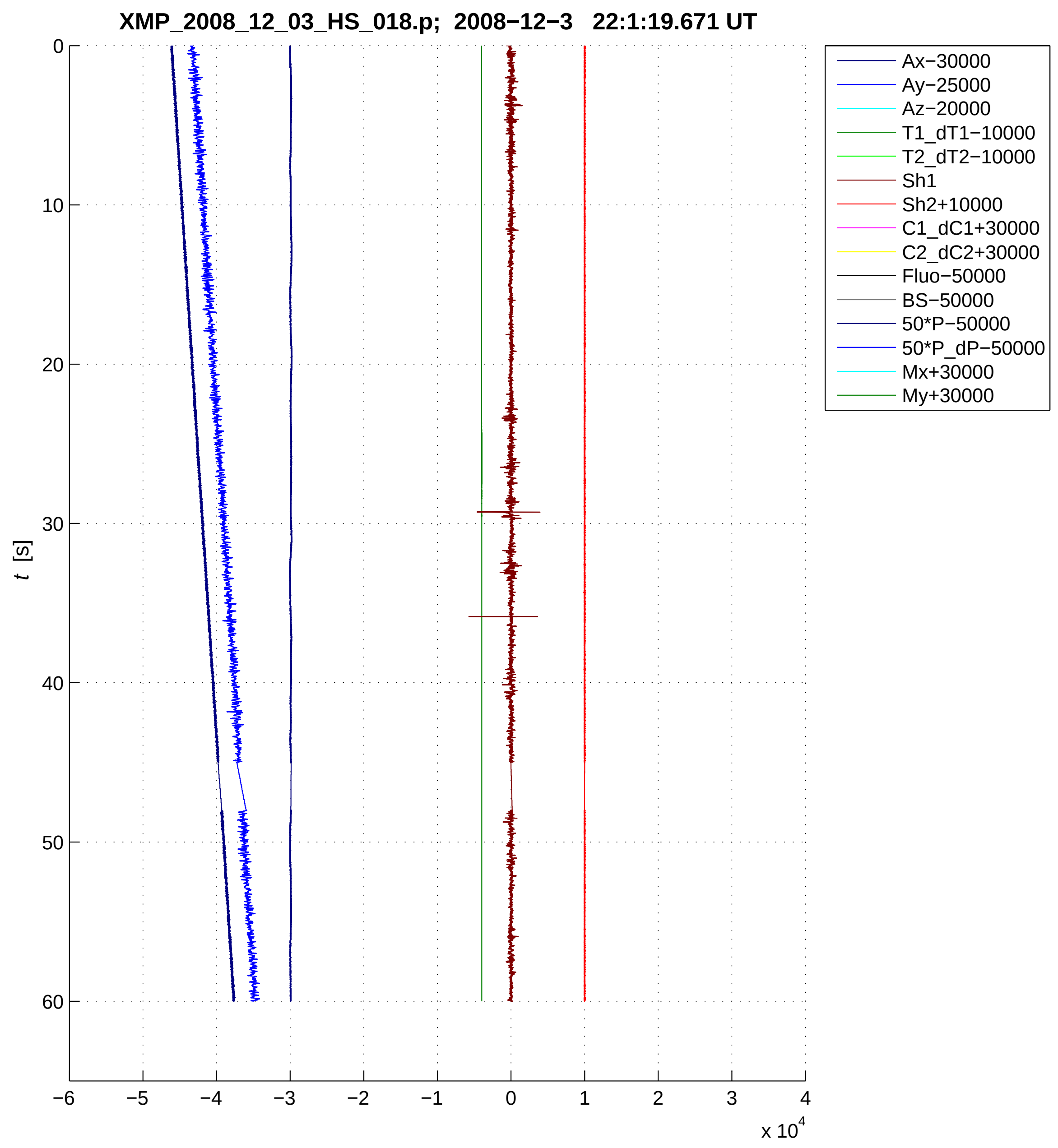

Figure 10: Same data as the previous figure after being processed by patch_odas.m and then by fix_bad_buffers.m. Three records have been replaced with interpolated data. Interpolated data should not be used for critical operations. However, the pressure records (black, blue lines) are now contiguous and can be processed to get an accurate record of depth.

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 – Type B

Escape underscore in supplied string for LaTeX

Syntax

```
stringOut = fix_underscore( stringIn )
```

Argument	Description
`stringIn`	Input string.
`stringOut`	Copy of the input string with all underscore characters prepended with a forward slash '\'.

Description

Escape the underscore character '_' into '_' within a string so that Matlab functions do not interpret it as a subscript command. This function allows file names to contain an underscore within plot text.

fopen_odas – Type A

Open a raw ODAS data file of unknown endian type

Syntax

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

Argument	Description
`fileName`	Name of the requested ODAS 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 raw ODAS 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

See the ODAS Data Acquisition Software Guide for more information on the endian flag.

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 – Type B

Calculate dissipation rates over an entire profile.

Syntax

```
diss = get_diss( SH, A, info )
```

Argument	Description
`SH`	Shear probe signals in physical units. It is assumed to be high-pass filtered to remove low-frequency wobble. Cut-off frequency should be roughly twice the fft-length, equivalent. Values are formatted as a matrix where each column is a shear probe.
`A`	Matrix of acceleration signals. Does not need to be in physical units.
`info`	Structure containing additional elements.
`diss`	Structure of results including the dissipation rate of TKE and related information. Results formatted as matrices with a row for each dissipation estimate.

Description

Calculate dissipation rates over an entire profile.

Input Structure

Argument	Description
`info.fft_length`	length of each fft segment in units of samples. For example info.fft_length = 1024
`info.diss_length`	the length of data used for each dissipation estimate. diss_length should be at least 2 times longer than fft_length. I recommend no less than a factor of 3.
`info.overlap`	the number of samples by which this function moves forward to make the next dissipation estimate. Should be larger than or equal to fft_length, and smaller than or equal to diss_length. I recommend diss_length/2. If you do not want any overlap, set overlap = diss_length.
`info.fs`	sampling rate in Hz.
`info.speed`	a scalar or vector of profiling speed to derive wavenumber spectra. If a vector, its length must equal the number of rows in SH.
`info.T`	a vector of temperature in degree C for calculating kinematic viscosity. Length must match SH. Salinity = 35 is assumed.
`info.P`	a vector of pressure for plotting the dissipation rates. Must match SH.
`info.Tolerance`	optional logarithmic tolerance condition on the average ratio of the observed spectrum divided by the Nasmyth spectrum. The ratio is calculated from the lowest wavenumber to K = 0.04*k_s (a position just past the peak of the spectrum), where K is the wavenumber in cpm (cyles per metre) and k_s is the Kolmogorov wavenumber, k_s = (epsilon/nu^3)^{1/4} in units of radians per metre. The default value is 0.3, which corresponds to a factor of 2 = (10^{0.3}).
`info.fit_order`	the order of the polynomial fit to the spectrum in log-log space used to estimate the spectral minimum wavenumber and this wavenumber is the inital estimate of the upper limit of integration for the estimation of the rate of dissipation of TKE. Optional - default = 6.
`info.f_AA`	the cut-off frequency of the anti-aliasing filter. Default = 98 Hz.
`info.fit_2_Nasmyth`	boolean value indicating a forced fit to the Nasmyth spectrum in the +1/3-range is desired. Default value is 0.

Output Structure

A structure of dissipation rate of TKE and related information. The number of rows equals the number of dissipation estimates. The elements are:

Argument	Description
`diss.e`	the rate of dissipation of TKE [W/kg]. One column for every shear probe.
`diss.K`	wavenumbers [cpm, or cycles per metre] with one column for every dissipation estimate.
`diss.sh_clean`	the wavenumber spectrum for each shear probe signal at each dissipation estimate. Every row is a 3-D matrix, where the highest dimension is the wavenumber index. The second and third dimensions are the cross-spectral matrix of the shear probes. The diagonal elements are the auto-spectra.
`diss.sh`	same as sh_clean but without cleaning by the Goodman coherent noise removal algorithm.
`diss.AA`	the cross-spectral matrix of acceleration signals. The diagonal elements are the auto-spectra.
`diss.UA`	the cross-spectral matrix of shear probe and acceleration signals.
`diss.F`	frequencies, with one column for every dissipation estimate. Currently every column is identical.
`diss.speed`	a column vector of the mean speed at each dissipation estimate.
`diss.nu`	column vector of kinematic viscosity [m^2/s] at each dissipation estimate.
`diss.P`	column vector of pressure at each dissipation estimate. Derived directly from the input.
`diss.T`	column vector of temperature at each dissipation estimate. Derived directly from the input.
`diss.Nasmyth`	the Nasmyth spectrum for the estimated dissipation rate, evaluated at the same wavenumbers as K

get_latest_file – Type A

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

Syntax

```
testString = get_latest_file()
```

Argument	Description
`testString`	Name of the latest ODAS binary data file in the current directory. Empty if no ODAS binary data files exist.

Description

Retrieve the name of the latest ODAS 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 – Type A

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`	Vector of pressure.
`W`	Vector of rate of change of pressure - make sure it has no flyers.
`pMin`	Minimum pressure for finding profiles.
`wMin`	Minimum rate of change of pressure for finding profiles.
`direction`	Direction of profile, either 'up' or 'down'.
`minDuration`	Minimum duration (s) for up/down to be considered a profile.
`fs`	Sampling rate of P and W in samples per second.
`profile`	2 X N matrix where each column is the start and end indexes for each profile. Empty if no profiles detected.

Description

Extract the slow sampling indices to the sections of a profile where the instrument is steadily moving up or down for at least min-duration seconds.

Call this function separately for ascents and descents.

Developed for finding sections in Slocum profiles but it can also be used for vertical profilers that have numerous profiles in a single file.

inifile_with_instring – Depreciated

Legacy function replaced by "setupstr"

Description

This is a legacy function. Please replace with the function 'setupstr'.

Complete documentation for this function can be found in the ODAS Library Manual v3.0. Alternatively, the documentation can be viewed by editing this function with the command:

```
>> edit inifile_with_instring
```

make_scientific – Type B

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 a matrix of values is inputed, each value is converted and a cell array of strings is returned.

Examples

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

Returns Pi with 5 significant digits: “3.1416 \times 10^{0}”

nasmyth – Type A

Generate a Nasmyth universal shear spectrum

Syntax

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

Argument	Description
`e`	Rate of dissipation in units of W/kg.
`nu`	Kinematic viscosity in units of m^2/s. Default value is 1e-6.
`N`	Number of spectral points. Default value is 1000.
`k`	Wavenumber in cpm.
`phi`	Nasmyth spectrum in units of s^-2 cpm^-1. 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 = 1e-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).

Form 3: Return the non-dimensional Nasmyth spectrum (G2 spectrum) of length N points. The wavenumber is $k = k'/ks$ [where k' is in cpm (see Oakey 1982)] and runs from 1e-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'/ks).

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 scaled Nasmyth spectra for the elements in e.

The form of the spectrum is computed from Lueck's (1995) fit to 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.
- Wolk, F., H. Yamazaki, L. Seuront and R. G. Lueck, 2002: A new free-fall profiler for measuring biophysical microstructure. J. Atmos. and Oceanic Techno., 19, 780-793.

odas_update – Type A

Perform online update and version check of ODAS Matlab Library

Syntax

```
status = odas_update( command )
```

Argument	Description
`command`	Function that should be initiated. Valid inputs are 'version', 'check', and 'update'.
`status`	Integer result of function call

Description

This function is used to update the ODAS Matlab Library. It checks the internet to see if the current version of the library is outdated and then downloads and installs the newer version if required.

The possible commands are;

- 'version',
- 'check',
- 'update';

where 'version' returns a numeric representation of the local version, 'check' goes onto the internet to see if a newer version is available returning the version number if an update is available, or 0 if there is no available update, and 'update' downloads and installs the latest version.

Examples

```
>> odas_update version
```

Return the current version of the local ODAS Matlab Library.

```
>> if odas_update( 'check' ),
>>     disp('the library should be updated');
>> end
```

Compare the current local version to the online version. Used to see if an update is required.

```
>> odas_update update;
```

Download and install the newest version of the ODAS Matlab Library. The program will interact with the user to determine the best location for the library.

patch_odas – Type A

Find bad buffers where the corruption can be fixed by a simple shift and interpolation

Syntax

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

Argument	Description
`file`	Name of 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 there is a communication failure with an instrument that 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 an ODAS binary data file. It is able to correct small errors in a record without damaging the surrounding data. It does this by locating the missing value and inserting a "best guess" approximation for that value.

This only works when damage to a data file is minor. Should the damage be too extensive, the algorithm will not work. In this scenario, the damaged record indices will be returned in the "fix_manually" vector.

Records that can not be fixed with this function can be patched with the "fix_bad_buffers" function. Use of this "fix_bad_buffers" should be minimized as it damages all channel data within the record.

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

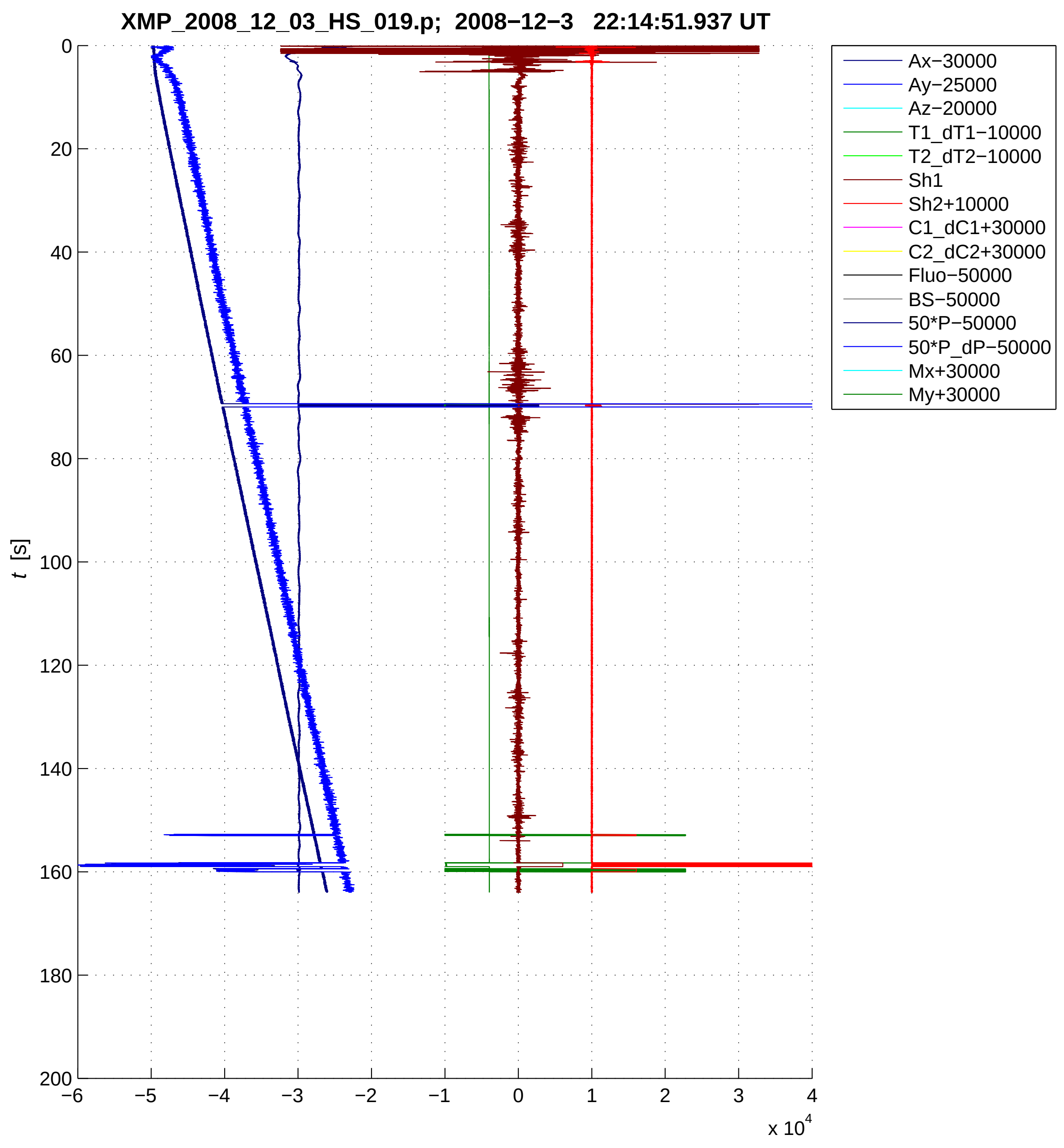

Figure 11: An example of a binary data file that has 4 bad records [70 153 159 160]. This data is from an Expendable Microstructure Profiler (XMP) that was notorious for bad buffers in the early stages of its development.

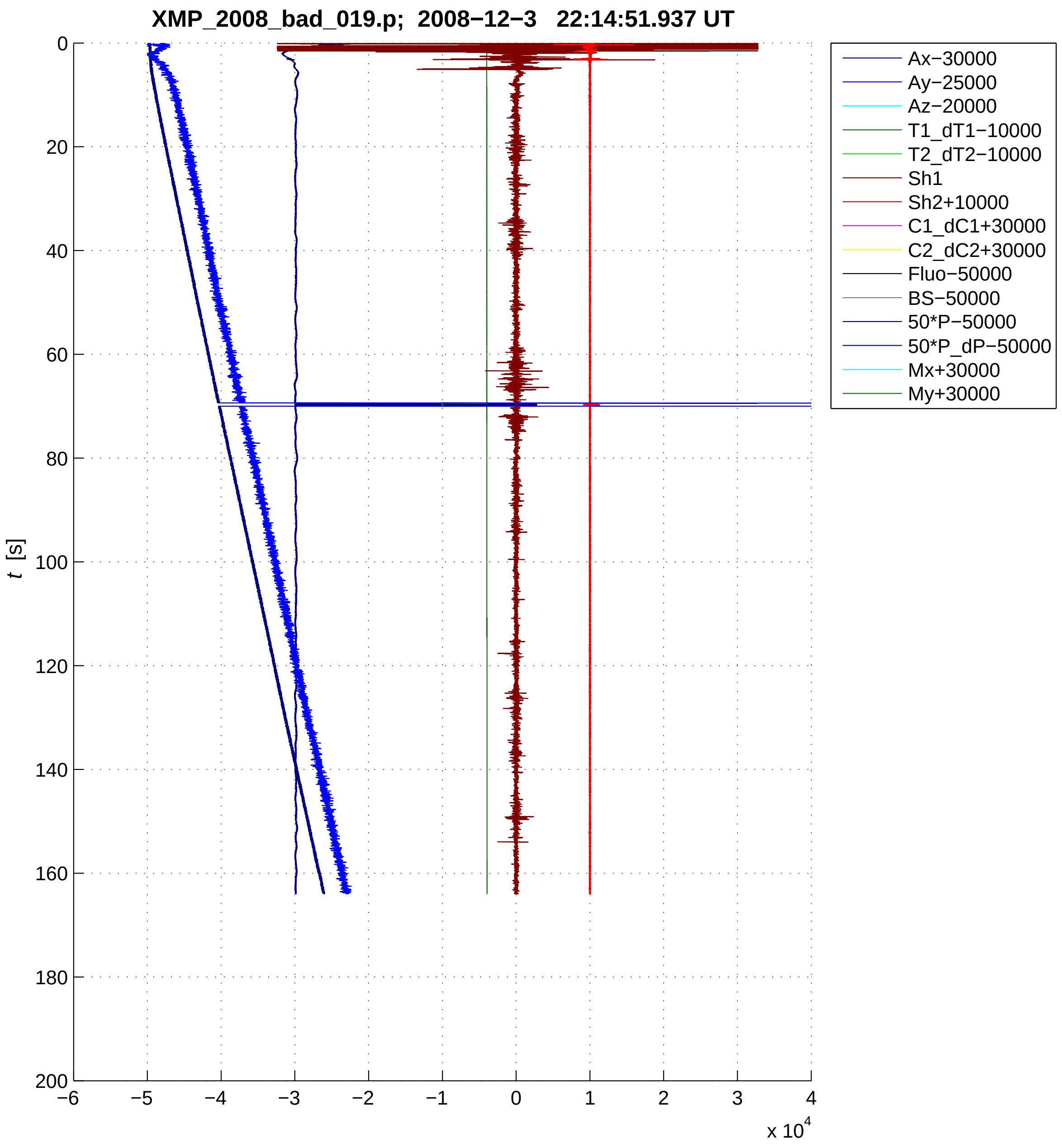

Figure 12: Plot of the previous figure after using patch_odas. The data in records [153 159 160] had skew corrected by inserting the missing datum. Corrected records can now be used for data processing. However, record 70 was not fixed because it had too many missing data points. The function fix_bad_buffers can patch this record with interpolated data but must be used carefully. The resulting record 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 files with patch_odas and then repair the remaining errors with fix_bad_buffers.

patch_setupstr – Type A

Patch a configuration string into an existing v6 ODAS data file

Syntax

`patch_setupstr( rawDataFile, configFile )`

Argument	Description
`rawDataFile`	Name of the ODAS data file into which 'configFile' should be embedded.
`configFile`	Name of the configuration file to embed.

Description

Patch an external configuration string into the first record of an ODAS data file.

This function is used to modify the header contained within an ODAS data file. One typically performs this task when calibration values contained within a data file need to be modified. It can also be used to convert older data files into v6 data files.

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.

Notes

- The original data file will be backed-up as 'fname_orig.p' only if this file does not already exist.
- The patched data file will be created as 'fname.p'.

- A comment will be added to the top of the new setup file string to emphasize that this data file has a patched setup file string.
- Certain parameters in the root section must be left unchanged in order to maintain the original structure of the data. Error messages will be displayed when attempting to change any of these values.
- Only the first header will be modified with respect to the new setup file string size at offset 12.

Examples

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

Embed the configuration string from 'setup.cfg' into the data file 'data_005.p'. If the backup file 'data_005_orig.p' does not exist, one is created with the contents of '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. Hidden in this example is how changes are made to the configuration file, 'data_005.cfg', by the edit command.

Notes

- Older v1 data files can be patched with a configuration file to update them to v6 data files. This greatly simplifies subsequent data processing and is highly recommended.
- Patching v1 data files with a v6 header only changes the first header.

patch_tomi – Type A

Legacy function, replace with patch_odas. This function call is redirected to patch_odas.

Syntax

```
[bad_record, fix_manually] = patch_tomi( file );
```

Description

Calls to patch_tomi can be directly replaced with patch_odas. The calling structure is the exact same so all that is required is for you to change the function name.

plot_HMP – Type A

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

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
```

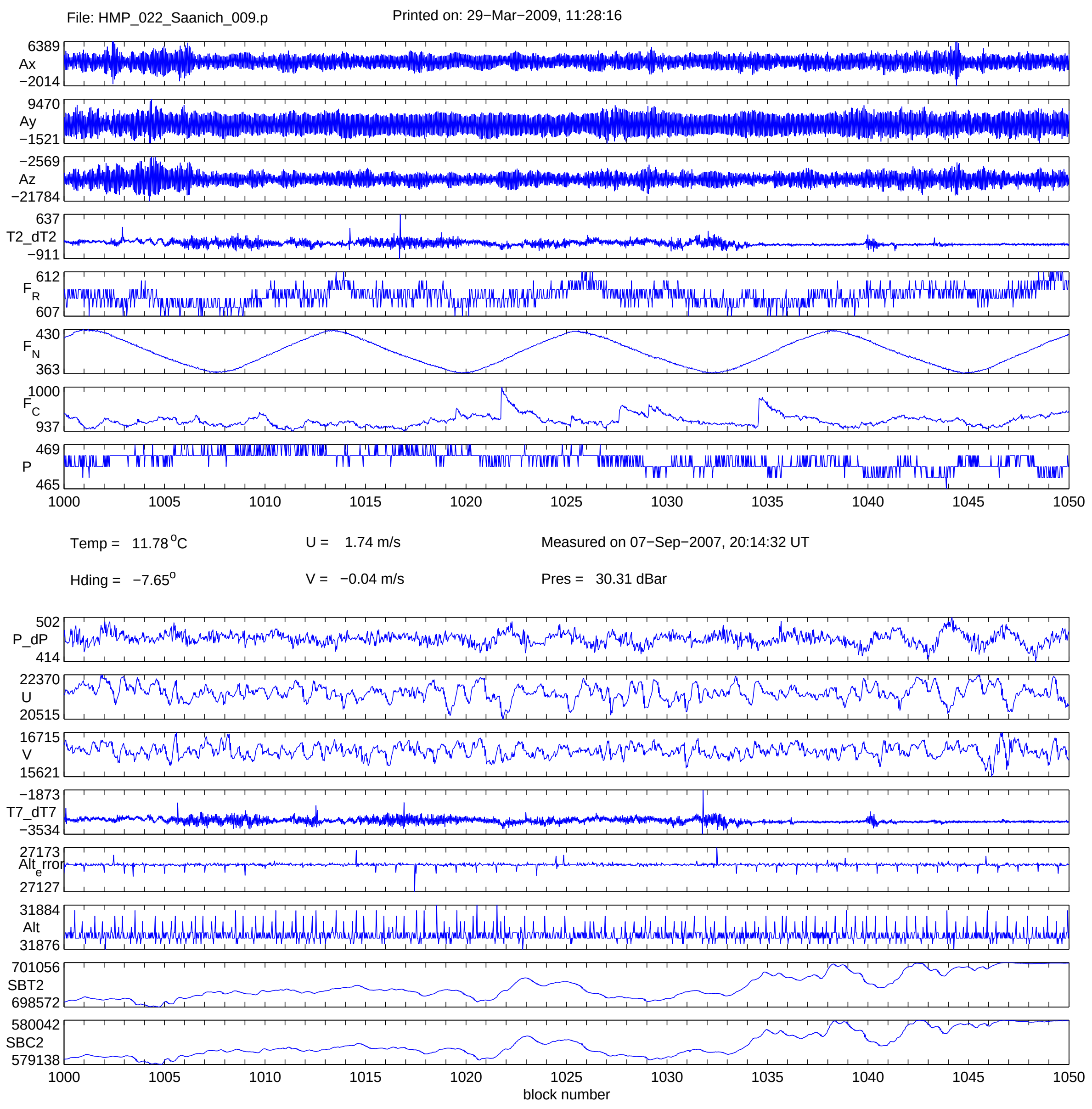

Figure 13: Data courtesy of Eric Kunze.

```
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_VMP – Type A

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

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....
```

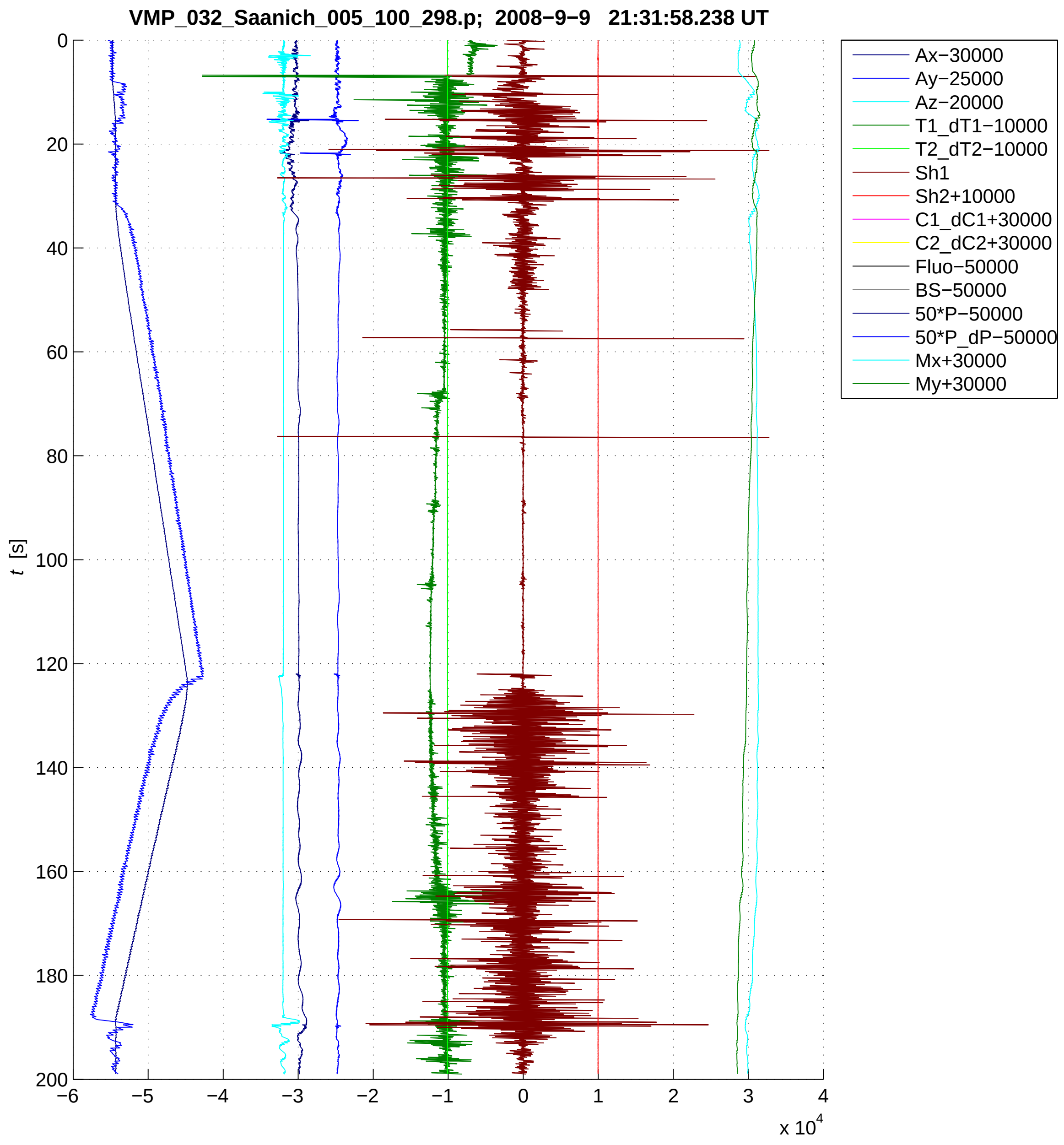

Figure 14: Example output from plot_VMP. A detailed explanation of the plot is found within the function description.

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.

plot_XMP – Type A

Simple real-time plots of data collected with XMP instrument.

Syntax

`plot_XMP( fileName )`

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

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....
```

psd_rolf – Depreciated

Calculate auto-spectrum. This is a legacy function, please use csd_odas.

Syntax

```
[Px, Fs] = psd_rolf( x, n_fft, rate )
```

Description

Calls to psd_rolf can be replaced with csd_odas. The calling structure for csd_odas differs from psd_rolf so some minor changes to your code will be required.

The three arguments to psd_rolf; x, n_fft, and rate; can all be passed into csd_odas but their location changes. Assuming the above three arguments were used to call psd_rolf, one would call csd_odas as follows;

```
[Px, Fs] = csd_odas( x, x, n_fft, rate, [], n_fft/2, 'linear');
```

query_odas – Type A

Extract slow/fast channels from address matrix in order.

Syntax

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

Argument	Description
`fileName`	ODAS 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 numbers stored in an ODAS binary (.p) file. The routine loads the address matrix from the .p file and returns a list of the channels in the matrix. If two output variables are provided, separate lists of slow and fast channels are returned in the same format as the address matrix. If only one output variable is used, the slow and fast channels are listed in order within a single vector.

Examples

```
>> query_odas
```

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

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

Channel numbers for the channels within the file 'myfile' are stored into the 'ch_list' variable.

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

Channel numbers for the channels within the file 'myfile' are saved in the same format as found in the address matrix. Slow and fast channels are separated into the two vectors 'ch_slow' and 'ch_fast'.

quick_bench – Type B

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

Syntax

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

Argument	Description
`dataFileName`	Name of the file to be processed (extension optional).
`serialNumber`	Serial number of the instrument as a string. Used in the resulting graphs.
`empty`	No return parameters but this function produces two figures.

Description

This function generates plots and figures from data collected from an RSI instrument that is being tested. While the instrument is on a bench, data is collected then processed using this function. Dummy probes should be installed when collecting data. 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 real profiles taken in the ocean, use quick_look.m to verify an instrument is working correctly.

Examples:

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

Perform a plot of the data file 'data_001.p' for an instrument with serial number 43. The serial number is not required but when provided, will be added to the resulting figures.

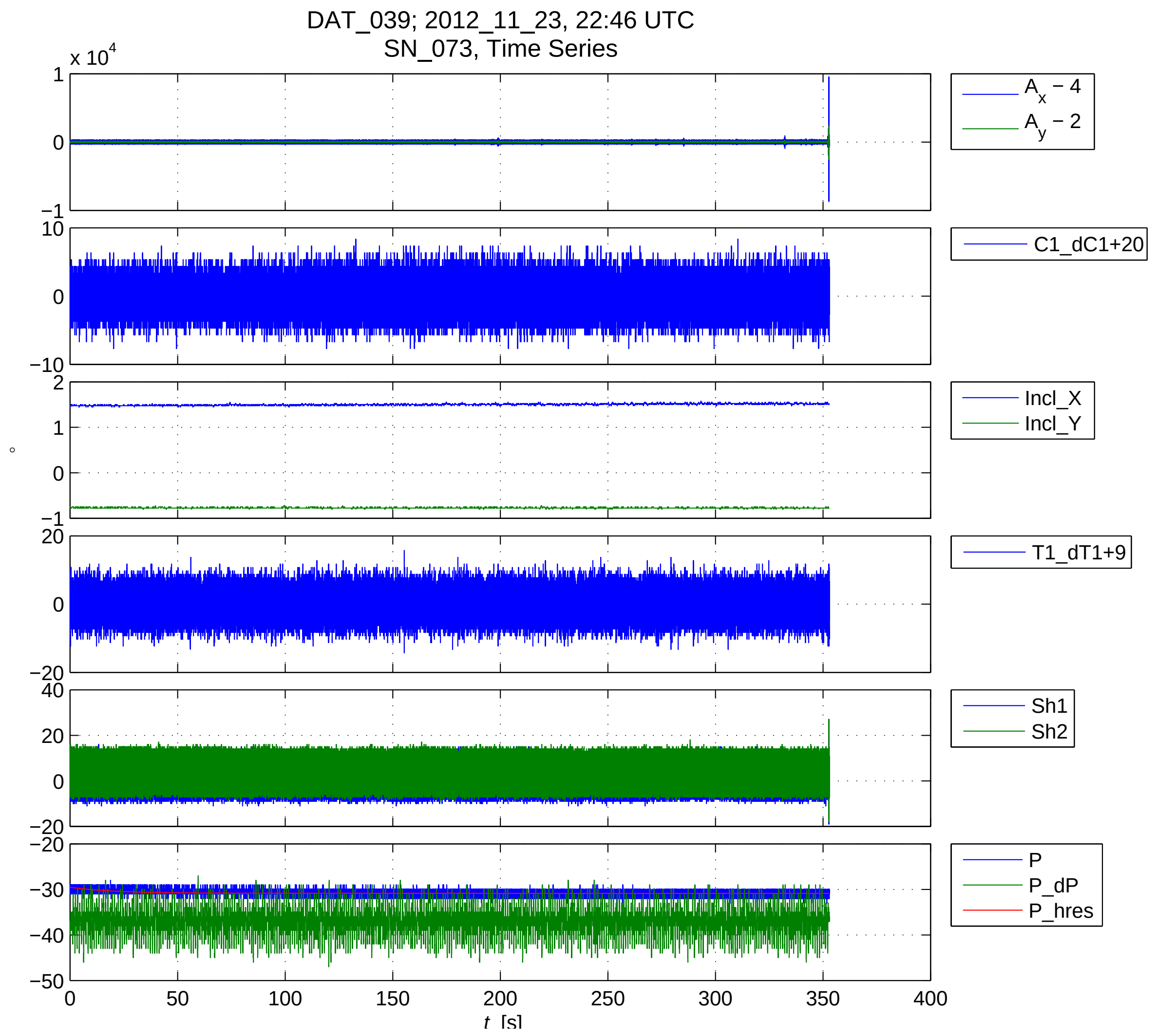

Figure 15: Quick_bench tries to plot most of the variables in your raw data file. These include accelerometers, pressure signals, shear probes, thermistors, magnetometers, inclinometers, voltage-output oxygen sensors, and micro-conductivity sensors. For some signals, the function subtracts the mean and 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.

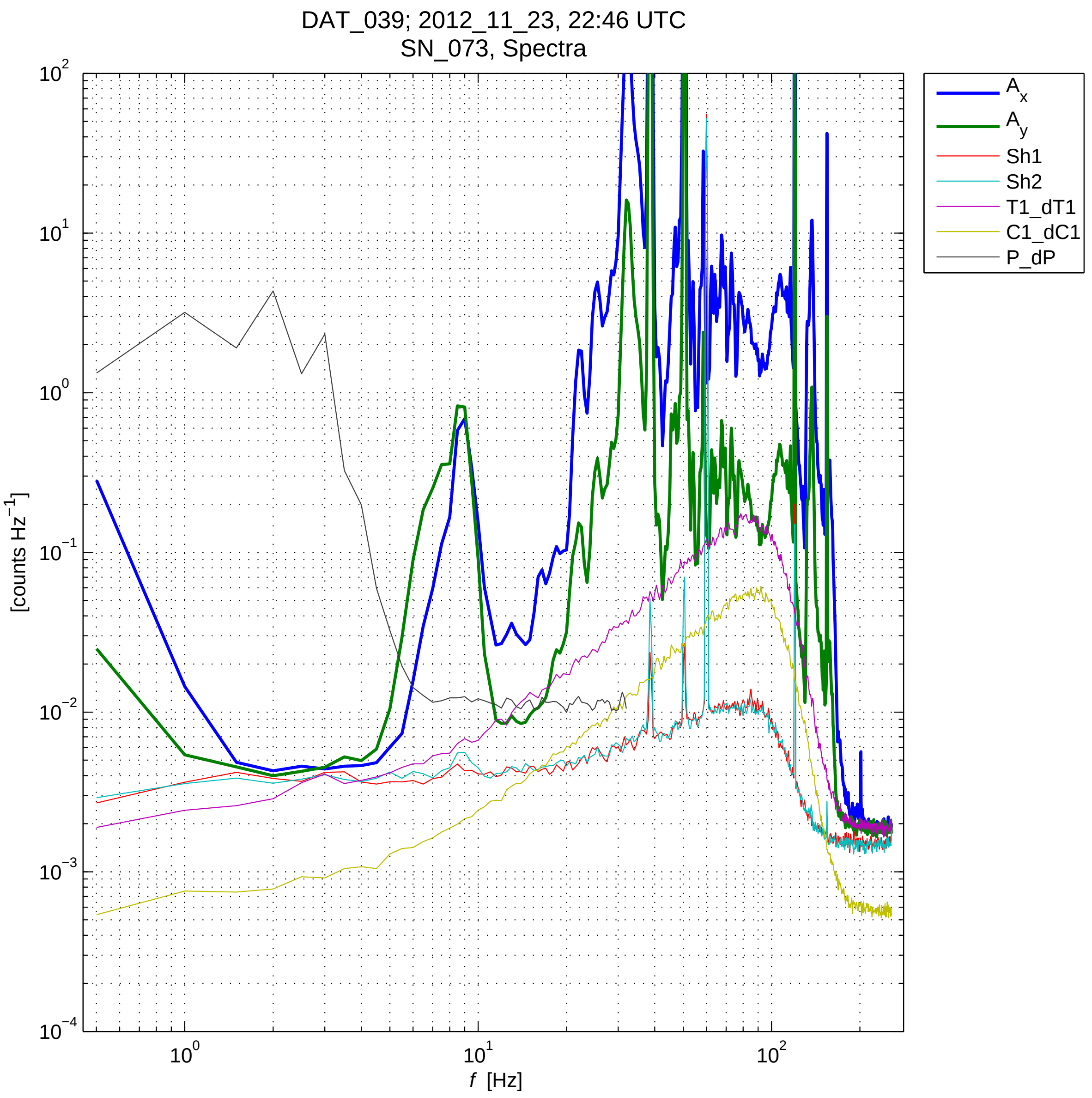

Figure 16: Spectra of some of the signals shown in the time-series figure. The instrument should be well cushioned to isolate it from vibrations. Even so, it is nearly impossible to suppress the output from the extremely sensitive accelerometers. Line frequency (50/60 Hz) contamination is also difficult 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 sensor. 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_bench_XMP – Type B

Generate plots and figures from a data file generated by an XMP instrument using ODAS-RT. Intended for 'bench' testing and not for real ocean profiles.

Syntax

```
quick_bench( fileName );
```

Argument	Description
`fileName`	Name of an ODAS data file to process. The data file is expected to be 1 to 2 minutes long.

Description

This is a function for the rapid generation of plots and figures from a data file generated by an XMP using ODAS-RT. It is intended for the quick evaluation of an instrument 'on the bench'.

quick_look – Type B

Generation of plots and figures from profiles recorded with a vertical profiler.

Syntax

```
quick_look( fname, P_start, P_end )
```

Argument	Description
`fname`	Name of the file to be processed. The file extension is optional.
`P_start`	Start point of the data section used for spectral estimates. This integer value is a depth value.
`P_end`	End point of the data section used for spectral estimates.

Description

This function is for the rapid generation of plots and figures from profiles recorded with a vertical profiler. Horizontal profilers should use quick_look_HMP. It is intended for the quick evaluation of an instrument during a sea trial. Clients are encouraged to examine this file to see an example of data processing.

If the depth range for calculating spectra is unknown, try some reasonable values for P_start and P_end and re-run the function after examing the resulting figures. Most figures do not depend on these values.

When using legacy data files, the local directory should contain a copy of the ODAS setup.txt file that contains calibration coefficients. This function will use those coefficients for the conversion to physical units. This is not required for newer v6 data files as the calibration coefficients are embedded within the data file.

The function performs a massive number of jobs. It starts by converting the binary data file into a Matlab mat-file and then loads the file. It systematically converts most channels into physical units. The following coefficients control the behavior of the quick_look.m function. Non-legacy ODAS data files (v6 or higher) only have to worry about the first section of variables.

```
% ********************************************************************
% Play with the following variables to optimize the outputs.
% ********************************************************************
P_min = 3.0;    % Minimum depth for profile start
W_min = 0.4;    % Minimum speed for profile start
Rise = 0;       % for rising profilers
fft_length = 1; % length in seconds
diss_length = 3*fft_length;
over_lap = diss_length/2;
HP_cut = 0.25;  % High-pass filter cut-off frequency (Hz) for shear probe
LP_CUT = 25;    % Low-pass filter cut-off frequency (Hz) for shear signals
%
% ********************************************************************
% Required for processing legacy data files.  The following default values
% should not be changed if processing version 6 data files.
% ********************************************************************
T1_offset =  0;         % Adjust T1 by this amount
T2_offset =  0;         % Adjust T2 by this amount
sbt_offset = 0.35;      % distance between FP07 and SBE3 in meters
                        %(usually 0.3 m or 2.5 m)
pump = 0;               % pump flag: 1 = SBT5 pump installed, 0 otherwise.
sh1_sens      = 0.0759; % SN725, 2010-11-25
sh2_sens      = 0.0719; % shear probe SN735, 2010-11-25
P_diff_gain   = 19.9;   % Gain of presure pre-emphasis ~20.5
T1_diff_gain  = 0.99;   % T1 differentiator gain ~1.0
T2_diff_gain  = 0.99;   % T2 differentiator gain ~1.0
C1_diff_gain = 0.4;     % micro-conductivity
Sh1_diff_gain = 1.01;   % Shear Channel 1 differentiator gain ~1.0
Sh2_diff_gain = 0.97;   % Shear Channel 2 differentiator gain ~1.0
ADC_FS = 4.096;         % Full-scale voltage of A-to-D converter
ADC_bits = 16;          % Number of bits in A-to-D converter
```

```
% ************************************************************************
```

Some variables are not completely converted to physical units. The thermistor and micro-conductivity signals are converted to physical units using the nominal sensitivity of these sensors by a linear transformation if you are using a data file prior to version 6. A linear transformation is reversible. The values will be close to the true values but not as close as they potentially could be. The reason is that these sensors are not calibrated. We expect that they will be calibrated using in situ measurements and a comparison against the Sea-Bird SBE3F thermometer and the SBE4C conductivity cell. The variables are then saved to the mat-file and then reloaded. If the mat-file already exists, then the reading of the binary file and its conversion are by-passed.

One of the figures shows the results of calibrating the thermistors but this result is not saved into the mat-file because the process is non-linear. However, it could be saved for future use by modifying the function. If the instrument carries micro-conductivity sensors, then calibration is also performed and shown in another window. Again, the results are not saved. One of the figures shows the frequency spectrum of acceleration, shear, temperature and micro-conductivity for the pressure range specified in the input arguments to this function. One can re-run this function for different pressure ranges to examine the shape of the spectra and to decipher the noise limits of the instrument. A wavenumber spectrum of the shear probe signals is shown in another figure. This section of the function does extensive computations and corrects the spectrum for the spatial resolution of the shear probes and for accelerometer-coherent vibrations of the profiler. It also, automatically finds the limits of spectral integration to compute the shear variance, which then is used to calculate the rate of dissipation of kinetic energy.

By default, the function saves plots as figures (.fig) and PDF files (.pdf). The figures allow the plots to be reformatted at any time in the future. The PDF files are high quality vector files which can be incorporated into most documents. Example plots generated by quick_look follow:

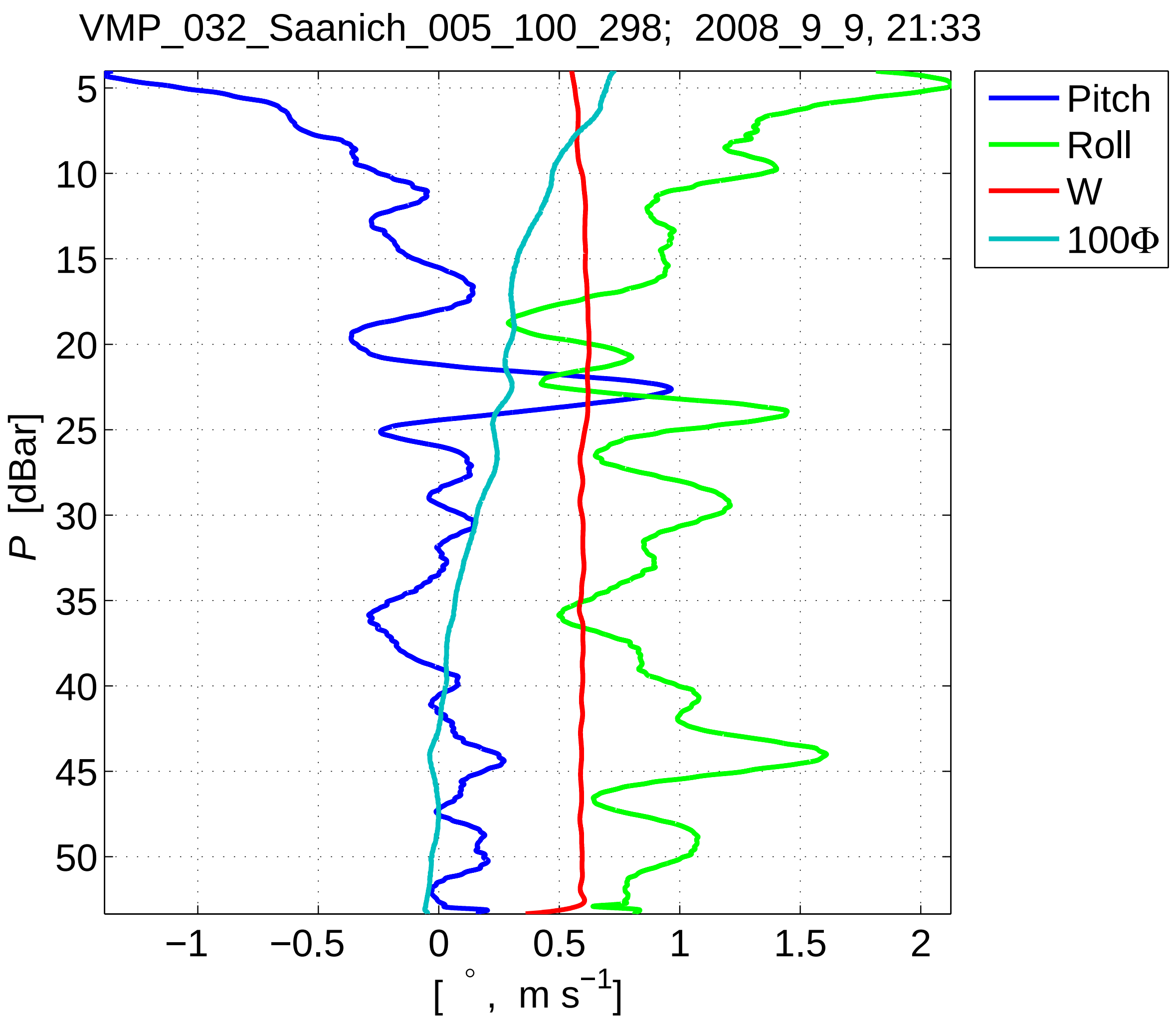

Figure 17: Quick_look example where pitch and roll are derived from the horizontal accelerometers and the fall-rate from the pressure transducer. If the instrument has a magnetometer then the magnetic orientation is shown in degrees divided by 100.

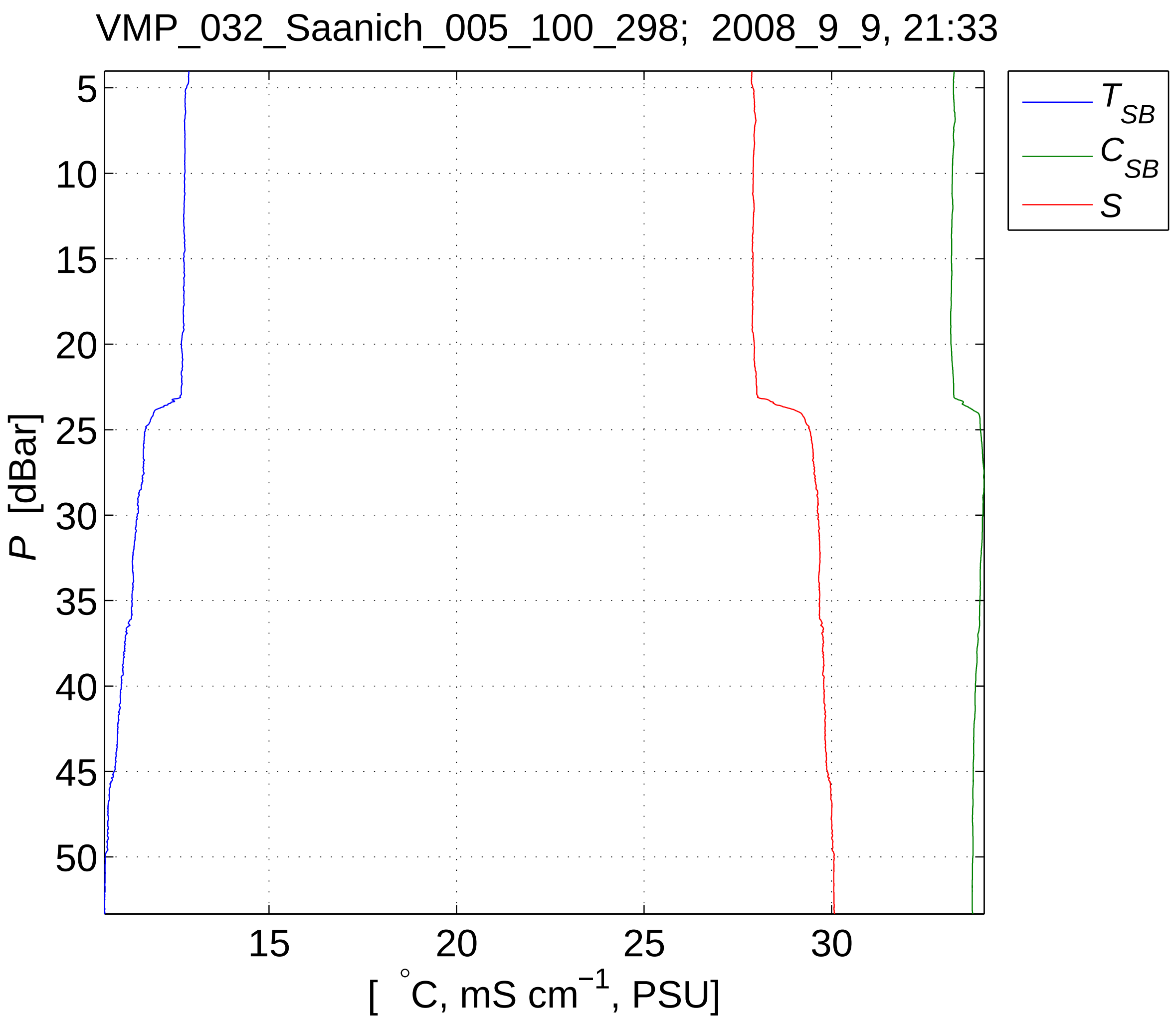

Figure 18: Quick_look example plots of data from Sea-Bird sensors. This figure is blank if the instrument does not carry Sea-Bird sensors.

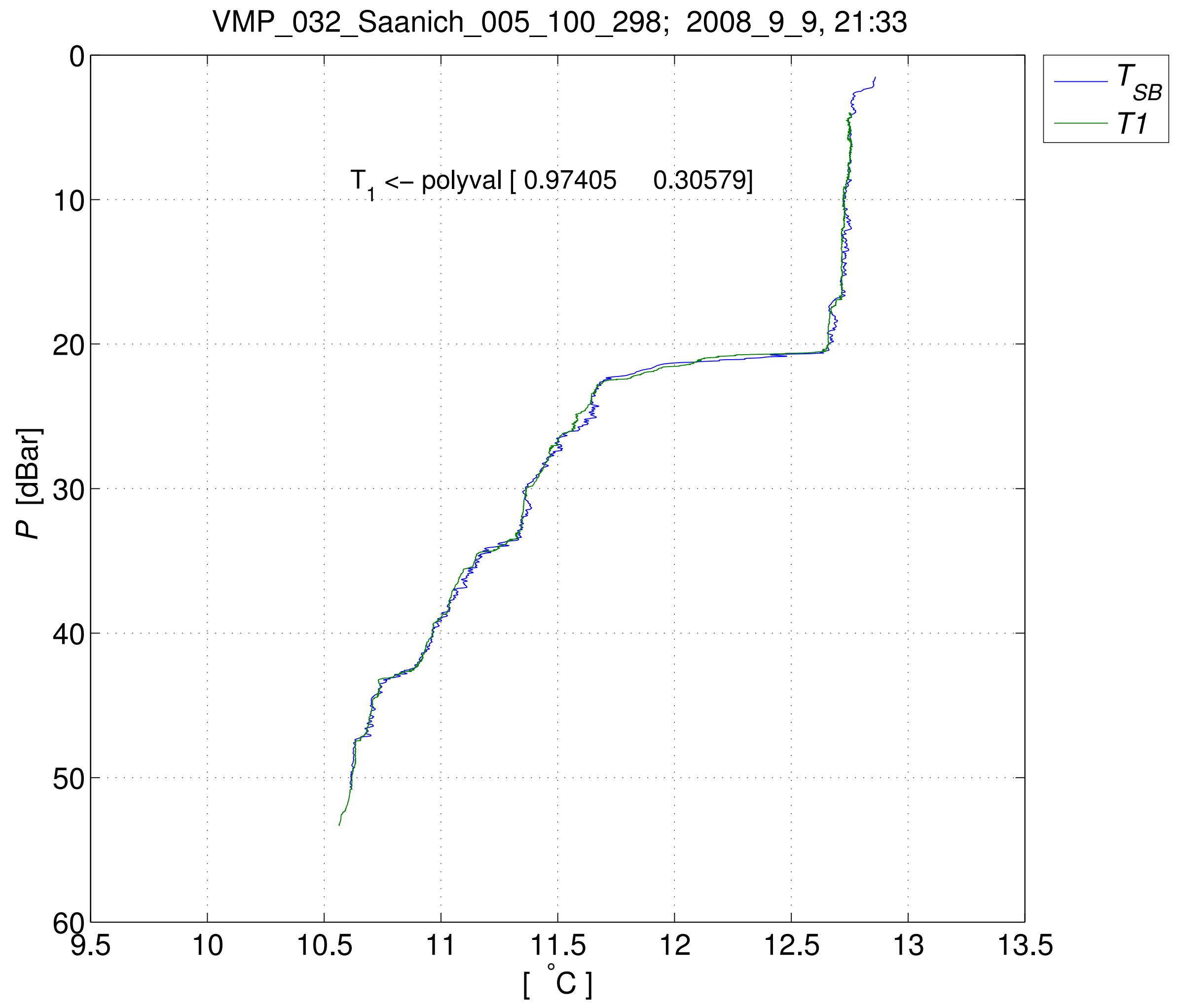

Figure 19: Quick_look example showing the results of a polynomial regression of the thermistors against the Sea-Bird thermometer. The user has to manually adjust the value of the separation of the Sea-Bird thermometer and the thermistors by editing the function. This instrument did not have a second thermistor installed. The polynomial coefficients indicate that the nominal thermistor conversion to physical units had to be shifted by 0.306 degrees Celsius and scaled by 0.97405 or its inverse, in order to agree with the Sea-Bird thermometer.

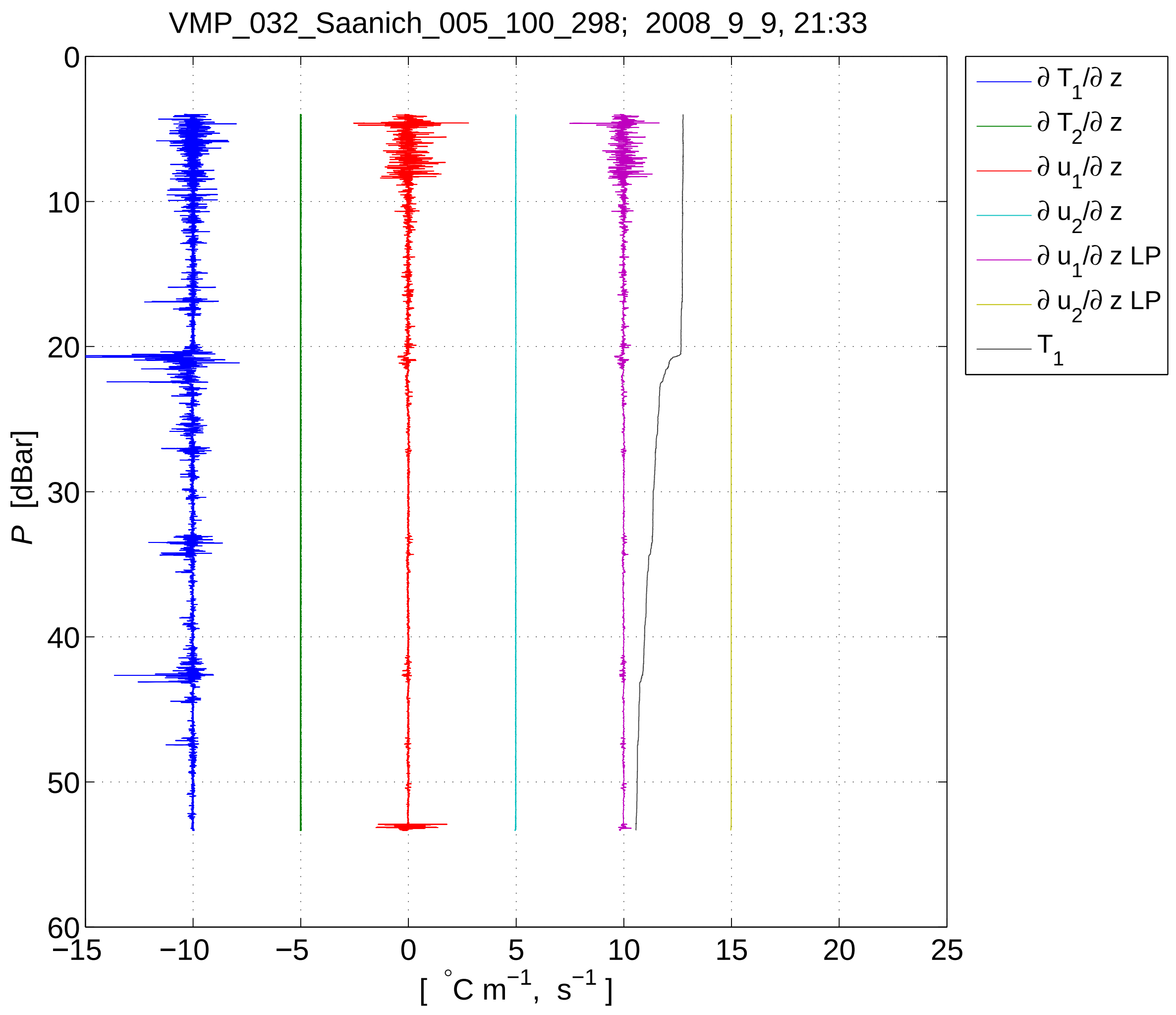

Figure 20: Quick_look example showing the vertical gradients of the micro-structure signals. LP indicates low-pass filtering at 30 Hz. This instrument had only a single thermometer and shear probe.

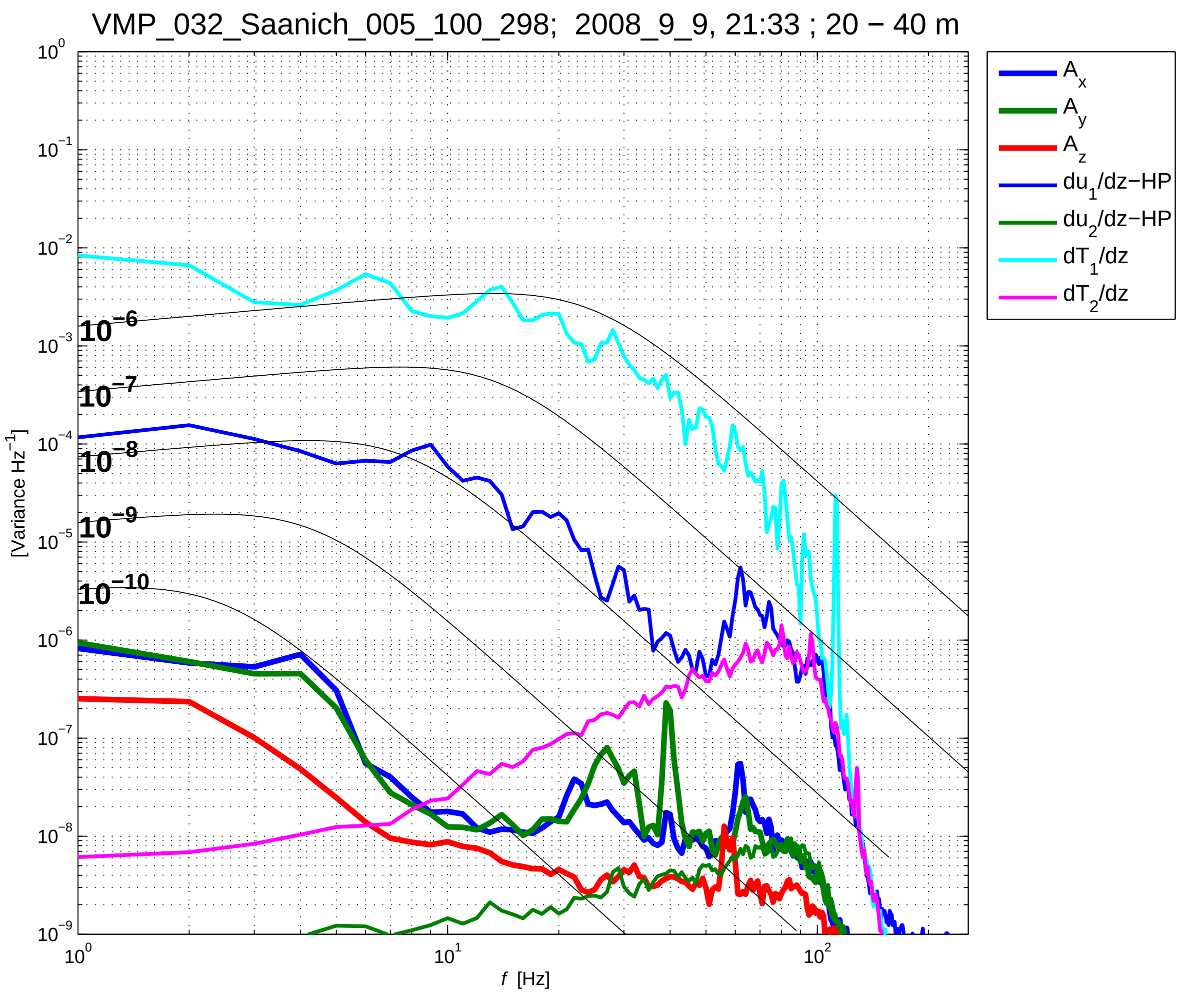

Figure 21: Quick_look plotted spectra for the pressure range selected by the input arguments to this function. All spectra are in physical units. Thermistor T2 and shear probe 2 were not installed, consequently, their spectra show the noise level of the electronics.

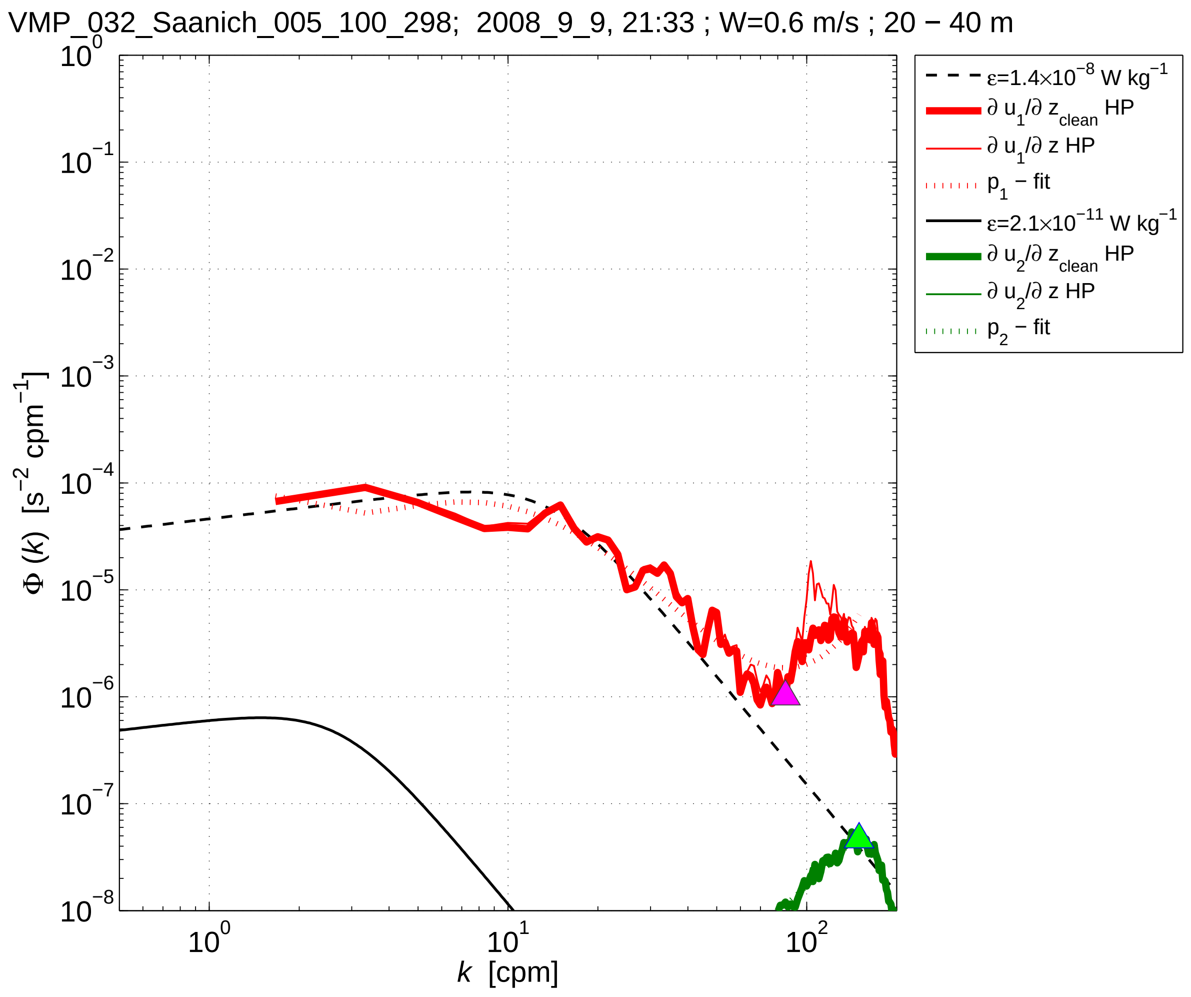

Figure 22: Final quick_look plot where the thin (red and green) lines are the wavenumber spectra for the selected pressure range. The thick lines are the spectra after correction for acceleration-coherent vibrations and the spatial averaging by the shear probes. The dotted lines are a polynomial fit used to estimate the upper limit of spectral integration for calculating the shear variance (triangle). The solid black lines are the Nasmyth spectra for the estimated rate of dissipation. The second shear probe was not installed, so its spectrum is mostly in nano-land below the limits of the figure.

quick_look_HMP – Type B

Generation of plots and figures from profiles recorded with a horizontal profiler.

Syntax

```
quick_look_HMP( fname, P_start, P_end )
```

Argument	Description
`fname`	Name of the file to be processed. The file extension is optional.
`P_start`	Start point of the data section used for spectral estimates. This integer value is a depth value.
`P_end`	End point of the data section used for spectral estimates.

Description

A function for the evaluation of a horizontal profiler using an actual profile in the ocean. It is usually used during sea trials of an instrument. It produces six figures of various channels including spectra of the shear probes, thermistors and micro-conductivity sensors. Clients are encouraged to examine this file to see one method of data processing. It is nearly equivalent to quick_look, so please look at quick_look for guidance on how to use this function. The data for all figures below are courtesy of Tim Boyd.

See quick_look.m for more information.

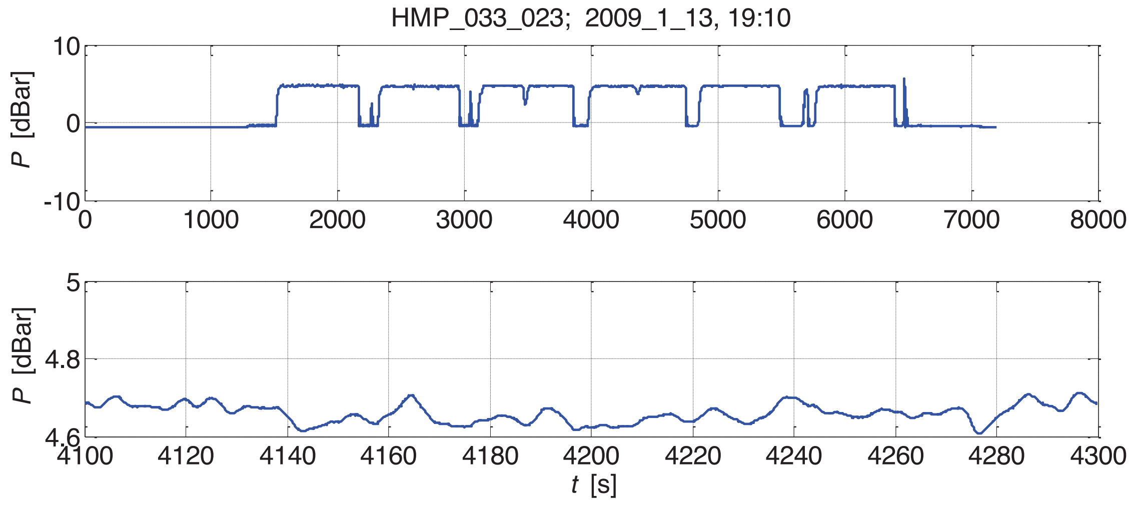

Figure 23: The first figure produced by quick_look_HMP. It shows the time series of pressure for the entire file (upper panel) and the pressure for the selected time range (lower panel).

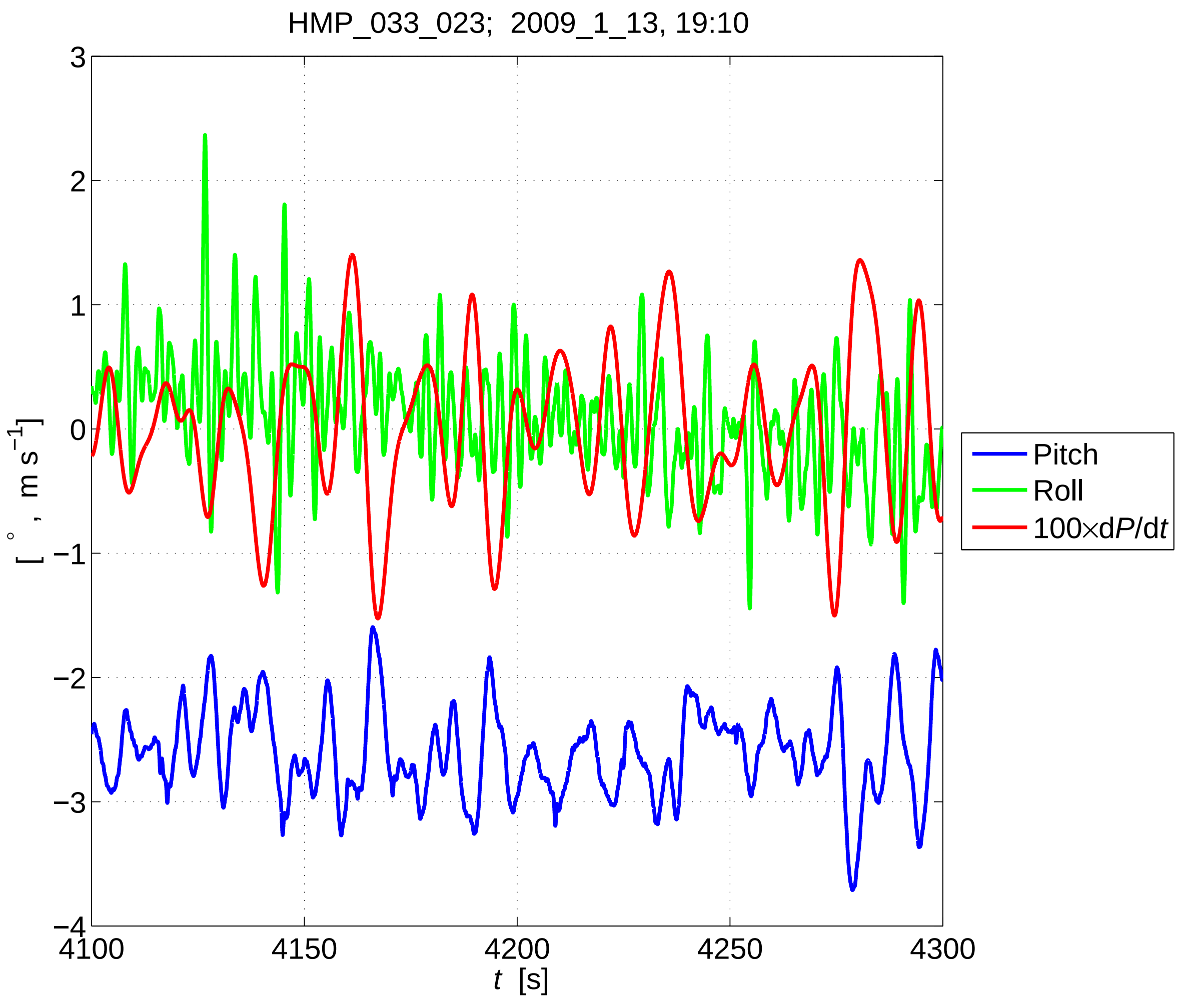

Figure 24: The second figure produced by quick_look_HMP. It shows the pitch, roll, and 100 times the rate of change of pressure (vertical velocity) for the selected time range.

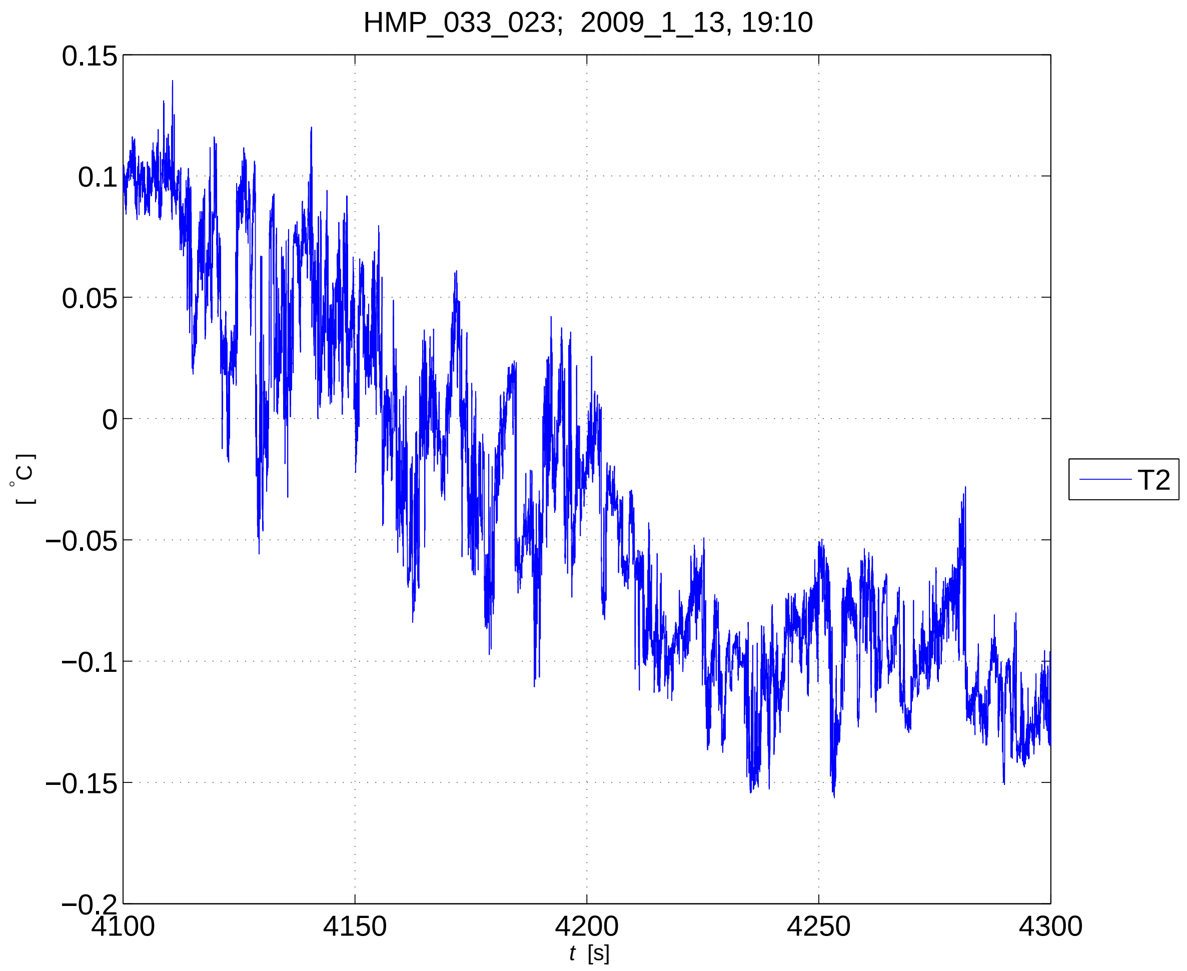

Figure 25: The third figure produced by quick_look_HMP. It shows the temperature for the selected time range using only the nominal linear conversion to physical units.

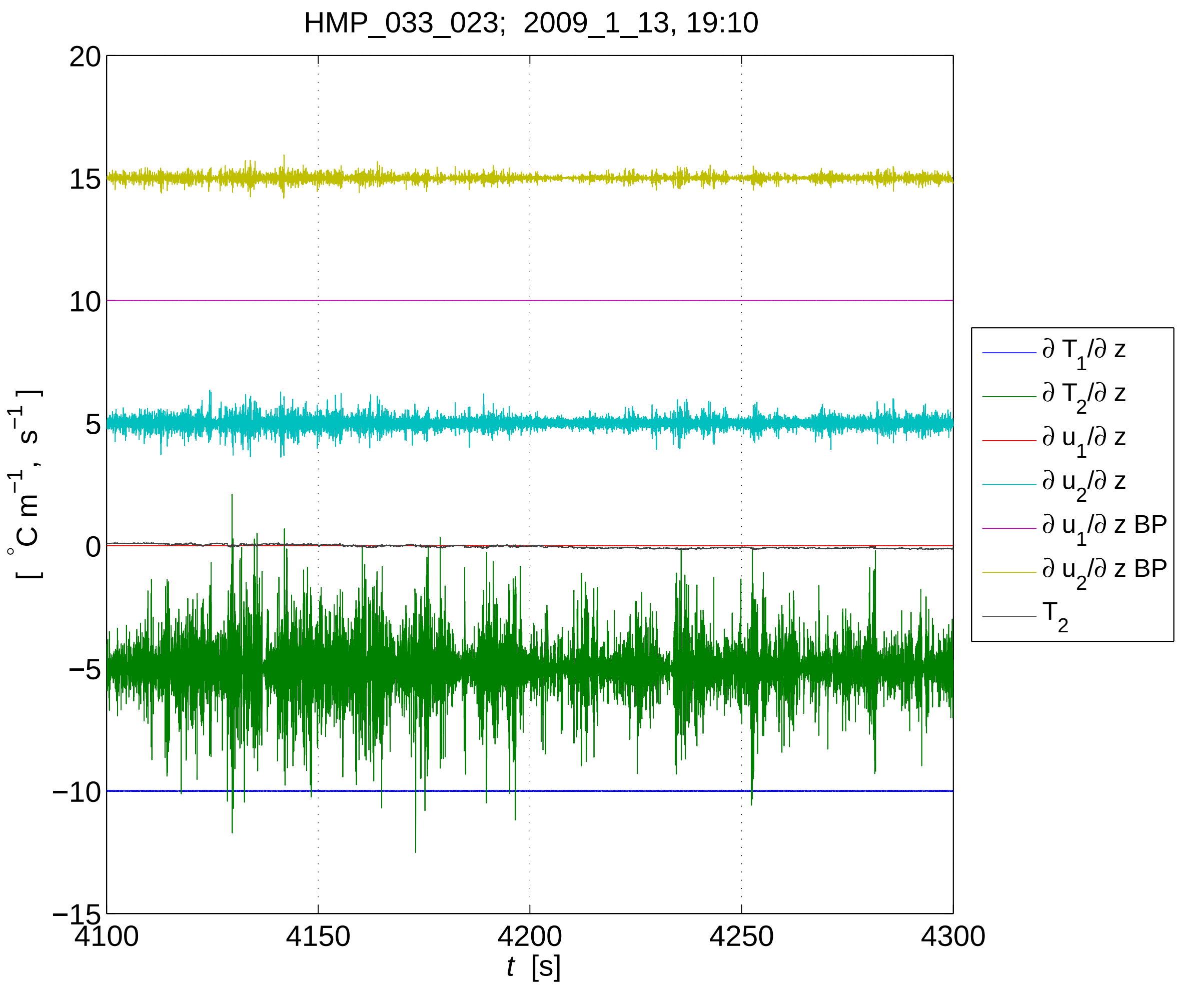

Figure 26: The fourth figure produced by quick_look_HMP. It shows the gradient of the microstructure signals for the selected time range. BP indicates low-pass filtered at 30 Hz.

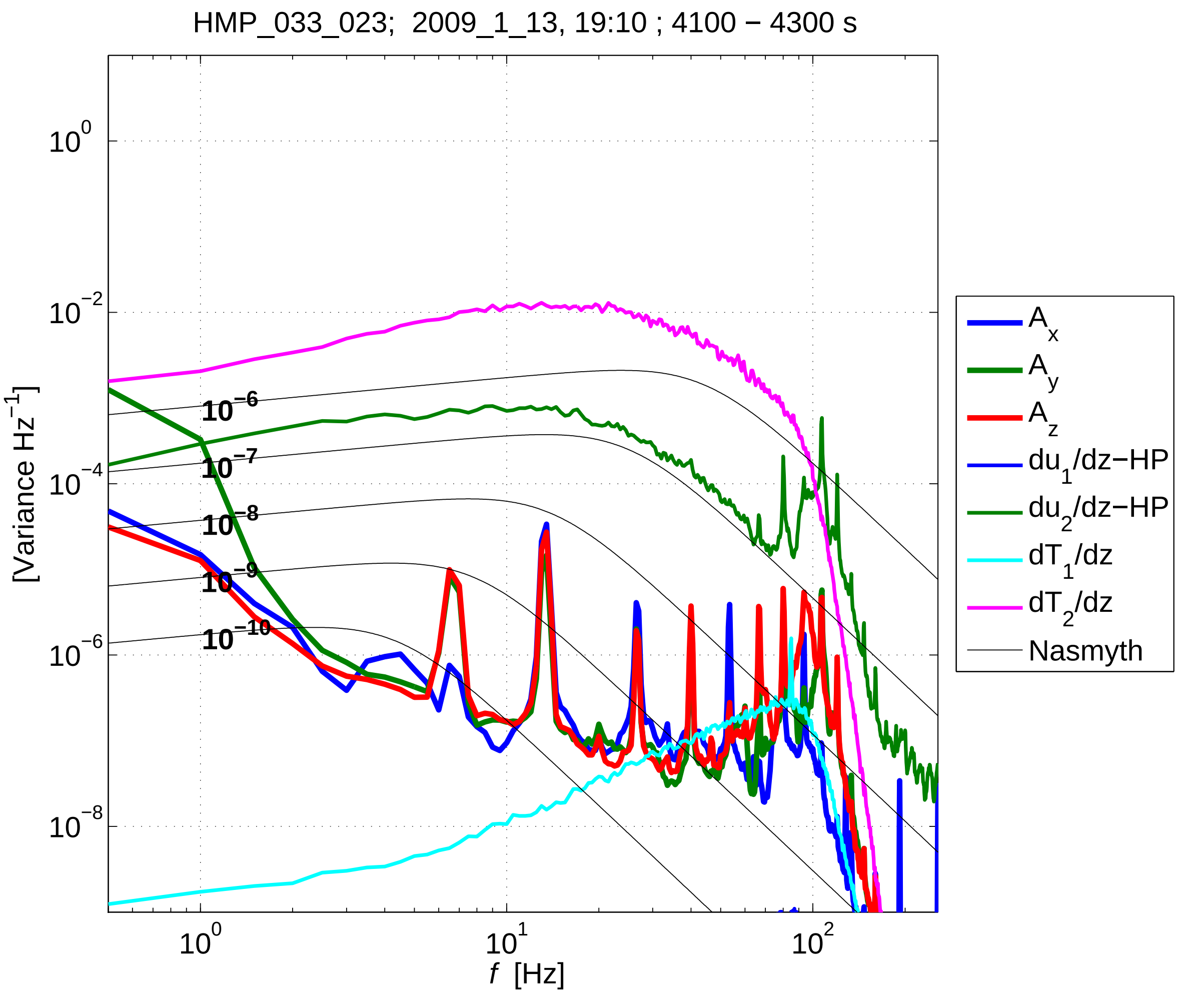

Figure 27: The fifth figure produced by quick_look_HMP. It shows the spectra of acceleration and all microstructure signals for the selected time range. Shear probe 1 and thermistor 1 were not installed. Note the propeller vibrations, evident in the acceleration spectra.

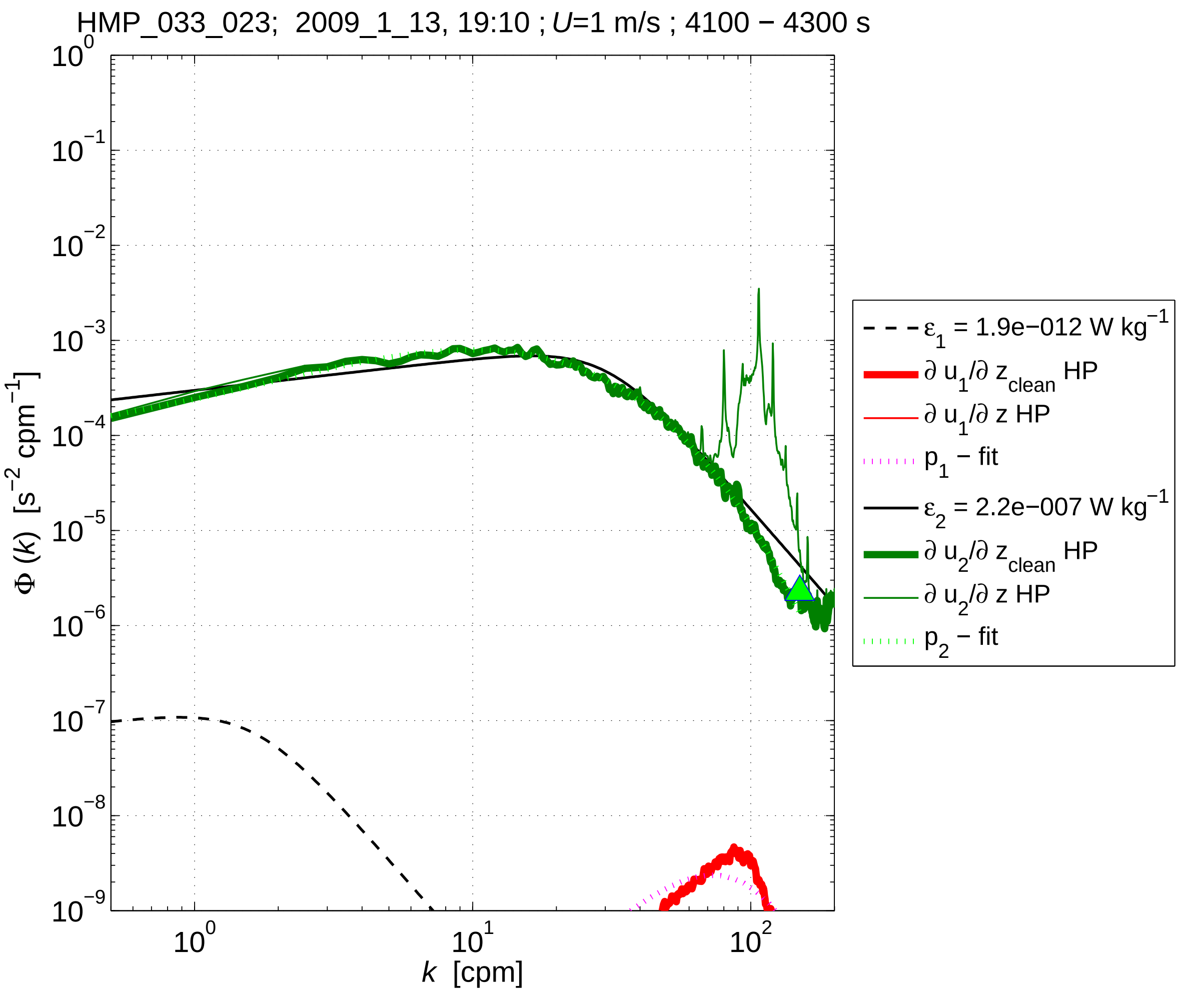

Figure 28: The sixth figure produced by quick_look_HMP. It shows the wavenumber spectra of the two shear probes. The thin green and red lines are the spectra of shear. The thick lines are the spectra after removal of acceleration-coherent vibrations and correction for the spatial response of the shear probes. The dotted curves are polynomial fits to determine the upper limit of spectral integration to obtain the variance of shear. The thin black lines are the Nasmyth spectrum for the rate of dissipation obtained from the shear variance. The triangles indicate the upper limit of spectral integration. Shear probe 1 and thermistor 1 were not installed. Note that the propeller vibrations have been removed from the shear probe spectrum (thin versus thick green curves).

read_odas – Type A

Convert an ODAS binary file to .mat format

Syntax

```
[var_list, outname] = read_odas( fname, ch_nums, ch_names )
```

Argument	Description
`fname`	Name of ODAS binary file. Optional, user prompted if omitted.
`ch_nums`	Vector of channel numbers to convert. Optional, all channels selected by default.
`ch_names`	Vector of names to assign the channels provided by 'ch_num'. Optional, default names used if not provided.
`var_list`	List of variables within the resulting .mat file.
`outname`	Name of the resulting .mat file.

Description

Convert an ODAS binary file (.p) into a MATLAB file (.mat). All requested channels are extracted from the input file and saved within a .mat file. The resulting .mat file can be loaded directly into Matlab.

If ch_nums is ommited, all channels found within the input file are extracted. Specific channels can be extracted by including ch_nums - a vector of numbers matching the requested channel numbers.

The extracted channels are given default names. Perferred names can be provided by including a ch_names argument. The ch_names argument should be a vector containing names to assign to those channels identified by ch_nums. If ch_names is included, ch_names and ch_nums should be vectors of the same length. ODAS v6 data files do not require this argument as the channel names are embedded inside the file.

If default channel names are used and the function can not guess what the correct name should be, a standard name is generated consisting of 'ch' and the channel number. For example, channel 239 would be named 'ch239'.

The default .mat file name is the input file name, fname, with the extension changed to '.mat'. If this file already exists, the user is queried for an alternate file name. When the output argument outname is used, a temporary name that differs from the default name is used for the .mat file. This is useful when functions need to access the raw data and will delete the generated .mat file when finished.

Examples

```
>> read_odas;
```

Extract all data from a .p file and store inside a newly generated .mat file. The input file name will be requested from the user.

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

Extract all data from 'my_file.p' and store inside 'my_file.mat'. Returns a list of the extracted channels inside 'my_vars'.

```
>> my_vars = read_odas('my_file.p', [1:4], {'gnd1','dw1','SBT1E','SBT1O'});
```

Function used on a legacy data file (pre-v6 header). Extract channels 1 to 4 from 'my_file.p' and store in 'my_file.mat'. The extracted channels are named 'gnd1', 'dw1', 'SBT1E', 'SBT1O' and are returned inside 'my_vars'.

Legacy ODAS Notes (pre v6)

- Default channel numbers and names are defined at the start of this function and can be modified for your instrument.
- Split channels are used when a sensor generates data as 32bit words. These words are addressed using two sequential 16bit channels. When read, these channels should be addressed individually with 'E' and 'O' suffixes appended to their names. When found, read_odas will internally combine the channels and return a single value. This behaviour can be turned off by modifying the 'combine_channels' flag inside the function.

read_tomi – Type B

Read a range of records from an ODAS data file.

Syntax

```
[blocks, header, data] = read_tomi( fid, start_block, end_block )
```

Argument	Description
`fid`	file descriptor for an open ODAS data file
`start_block`	block index of the first segment block
`end_block`	block index of the last segment block - inclusive
`blocks`	vector of the extracted record numbers
`header`	record header
`data`	requested record data

Description

Read a range of blocks from an ODAS data file. This function is primarily used by patch_odas but can also be used directly for the purpose of processing large data files into manageable portions.

Examples

```
>> [fid, error] = fopen_odas( 'raw_data_file.p', 'r' );
>> if isempty( error ),
>>     [blks, header, data] = read_tomi( fid, 1, 1 );
>>     fclose( fid );
>> end
```

Read the first block from the data file 'raw_data_file.p'.

save_odas – Type A

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`	the actual vector that is saved
`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 was written to spare this author recurring frustration with the Matlab 'save' command. The creators of Matlab have certainly done a marvelous service to the scientific and engineering community. However, it appears that they never envisioned that users might want to work with large data files and that they might want to update vectors in such a file. If you want to append a vector to an existing mat-file, then this operation proceeds quite quickly because the vector is simply placed into the back of the file. However, if you have loaded a vector, changed its value but not its length, and try to update your file by appending this modified vector into the mat-file, then Matlab ties itself into a multi-dimensional knot. It first copies the entire file into a temporary file, empties the original file, copies the vectors and variables one-by-one, replaces the target vector when its turn arrives, and then continues to copy vectors and variables one-by-one. Microstructure profiles tend to have vectors of about 10e6 in size, and re-saving a single vector can take up to 2 minutes!. Enter save_odas. This function examines the mat-file to see if it already contains a vector with the same name as the one that is to be saved. If it does not find a vector of the same name, then it simply issues the save append command. If it does find a vector with the same name, then it checks to see if it has the same length as the one that is to be saved. If the lengths are equal, then it uses the binary write command to over-write the vector in the mat-file directly. Updating a vector of length of about 10e6 takes approx. 1 second. Can you believe it? Can you imagine what this has done for my blood pressure?

This function only works for mat-files saved as version 6. You do this by using the v6 flag with the save command. This function, of course does this for you automatically. MathWorks has come out with 2 newer versions of its mat-files, but RSI has no knowledge of the internal structure of these newer-version files.

NOTE: This function works only with real vectors.

Examples

```
>> vector = 1:1000000;
>> save( 'my_big_vector.mat', 'vector' );
>> vector(1) = -1;
>> save_odas( 'my_big_vector.mat', 'vector', vector );
```

First creates a .mat file and saves a large vector. The vector is then modified and the .mat file updated.

save_rolf – Depreciated

Legacy function replaced by "save_odas"

Description

This is a legacy function, please change your functions/scripts to use the function 'save_odas' instead. This function calls 'save_odas' using:

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

setupstr – Type A

Read attributes from an ODAS configuration string.

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 an ODAS configuration string - an INI-like string used to store values used by the ODAS library. This string can be obtained by reading a configuration file (.cfg) or by extracting the relevant parts of a ODAS v6 or higher data file.

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 thereby reducing the time required to make those calls.

```
- obj = SETUPSTR('config_string')
   Parse the string 'config_string' and return 'obj'.  The returned value
   is a data structure containing values within the config_string.  The
   returned 'obj' is 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 parameter 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`	Either the configuration string or 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 file. Passed to 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

All search queries are interpreted as regular expressions. If empty, the query will match anything by using the regular expression '.*'. Each search query is prepended with '^' and appended with '$' to ensures exact matches.

Parameters that precede the first section are said to be in the 'root' section. To access these parameters, use 'root' as the section identifier.

Examples

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

From the variable 'setupfilestr', which contains a configuration file, query the value of '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 an 'id' value. Allow for 'id_even' and 'id_odd'.

Note: the above example will match any key starting with 'id'. For a more precise match, 'id.*' could be changed to 'id(_(even|odd))?'.

show_P – Type B

Extract and show the average pressure from an ODAS data file.

Syntax

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

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

Description

Calculate the record-average pressure from a binary ODAS data file. This function can be used to identify which portions of a data file contain data that is of interest. One can then decimate the data file into smaller files that contain those specific portions of data.

For this function to work, the pressure channel address must be present in the address matrix. The configuration file must also contain the appropriate coefficients for converting raw pressure values into physical units.

For legacy ODAS (prior to ODAS v6 data files), the user needs to supply the setup file that was used during data acquisition. Newer data files can be processed directly if they were obtained using a valid configuration file.

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. This plot allows one to visually identify the sections of a data file they are interested in. The decimate function can then be used to crop the data file to the portions that are of interest.

TI_correction – Type A

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

Syntax

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

Argument	Description
`T`	Temperature measured by the SBE3F thermometer representing the environmental temperature.
`alpha`	Amplitude coefficient for the thermal inertial of the SBE4C conductivity cell.
`beta`	Inverse relaxation time for the thermal inertia of the SBE4C conductivity cell, in units of 1/s.
`fs`	Sampling rate of the data, in samples per second.
`T_cell`	Water temperature in the cell adjusted for the thermal inertial of the cell. This value should be used for salinity calculations but not for density calculations.

Description

Adjusts the temperature measured with a Sea-Bird SBE3F thermometer to account for the temperature anomaly resulting from the thermal inertia generated inside a Sea-Bird SBE4C conductivity cell. This function is based on Lueck (1990), Lueck and Picklo (1990) and Morrison et al (1994). The user should be familiar with these papers to fully understand when this function should be used.

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 guesses of their values reduces significantly the errors in the calculated salinity. These errors are frequently called 'salinity spikes'. The user should also know that this function corrects only the thermal inertia. The short-term mismatch between the measured temperature and conductivity must also be corrected to reduce spurious salinity signals.

- 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 – Type A

Approximation for the kinematic viscosity of freshwater for $S = 0.0$

Syntax

```
v = visc00( t )
```

Argument	Description
`t`	temperature in degrees Celsius
`v`	viscosity in m^2/s

Description

Returns an approximation of the kinematic viscosity, based on temperature (in degrees C). The viscosity is derived from a 3-rd order polynomial fit of nu against T for salinity 0. The error of the approximation is less than 1% for (0 <= T <= 20) at atmospheric pressure.

(see also viscosity)

visc35 – Type A

Approximation for the kinematic viscosity of seawater for S = 35.0

Syntax

```
v = visc35( t )
```

Argument	Description
`t`	temperature in degrees Celsius
`v`	viscosity in m^2/s

Description

Return an approximation of the kinematic viscosity, based on temperature (in degrees C). The viscosity is derived from a 3-rd order polynomial fit of nu against T for salinity 35. The error of the approximation is less than 1% for (30 <= S <= 40) and (0 <= T <= 35) at atmospheric pressure.

(see also viscosity)

viscosity – Type A

Calculate kinematic and molecular viscosity of sea water

Syntax

```
[nu, mu] = viscosity( s, t, r )
```

Argument	Description
`t`	temperature in degrees Celsius
`s`	salinity in practical salinity units
`r`	density in kg/m^3
`nu`	the kinematic viscosity in units of m^2/s
`mu`	the molecular viscosity in units of kg/(m*s)

Description

Calculates the kinematic and molecular viscosity of sea water following Millero (1974). The range of validity is (5 <= t <= 25) and (0 <= S <= 40) 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.

Index

Depreciated
- csd_rolf, 56
- inifile_with_instring, 75
- psd_rolf, 90
- save_rolf, 113

Type A
- adis, 48
- check_bad_buffers, 49
- clean_shear_spec, 49
- conductivity_TPcorr, 51
- convert_odas, 52
- csd_odas, 54
- deconvolve, 56
- despike, 59
- extract_odas, 61
- extract_setupstr, 62
- file_with_ext, 65
- fix_bad_buffers, 66
- fopen_odas, 70
- get_latest_file, 74
- get_profile, 75
- nasmyth, 76
- odas_update, 78
- patch_odas, 79
- patch_setupstr, 83
- patch_tomi, 84
- plot_HMP, 85
- plot_VMP, 87
- plot_XMP, 89
- query_odas, 90
- read_odas, 109
- save_odas, 112
- setupstr, 113
- TI_correction, 117
- visc00, 118
- visc35, 119
- viscosity, 119

Type B
- fig2pdf, 63
- fix_underscore, 70
- get_diss, 71
- make_scientific, 76
- quick_bench, 91
- quick_bench_XMP, 95
- quick_look, 95
- quick_look_HMP, 103
- read_tomi, 111
- show_P, 116