
obsinfo

Release 0.110

Luis Arean and Wayne Crawford

Sep 14, 2023

TABLE OF CONTENTS:

1	Obsinfo Overview	1
1.1	Introduction	1
1.2	Object Model	2
1.3	Information Files	3
1.4	Python architecture	5
1.5	Resources	6
2	Classes	7
2.1	Information File Tree	7
2.2	Filter Types	8
3	Installation and Startup Guide	69
3.1	Prerequisites	69
3.2	Python installation	69
3.3	Python packages	70
3.4	Git installation	70
3.5	Obsinfo installation	71
3.6	Description of obsinfo file structure	71
3.7	Obsinfo setup	71
3.8	File discovery	72
4	Executing obsinfo	75
4.1	obsinfo-setup	75
4.2	obsinfo-makeStationXML	75
4.3	obsinfo-validate	77
4.4	obsinfo-print	77
5	Tutorial	79
5.1	Introduction	79
5.2	The general structure of information files	82
5.3	Building a simple network file with stations	84
5.4	Building a simple instrumentation file with channels	90
5.5	Building instrument component files with response stages: sensors and preamplifiers	96
5.6	Building a datalogger information file	101
5.7	Building a stage information file with different filters	103
5.8	Building a filter information file for different filters	106
5.9	Summary	108
6	Information files	111
6.1	Overview	111
6.2	Basic - two channel	120

6.3	Basic atomic - two channel	120
6.4	Basic atomic + configuration - two channel	120
6.5	Comparison with StationXML	121
6.6	Comparison with AROL/YASMINE	139
7	Datacite information files	151
7.1	Introduction	151
8	Advanced	155
8.1	Channel modifications	155
8.2	AROL compatibility	159
8.3	Best practices	160
8.4	Notes	163
8.5	Caveats	163
8.6	Troubleshooting	164
8.7	Addons	165
9	Training Course	167
9.1	Introducing obsinfo	167
9.2	Setting up	174
9.3	Creating a StationXML file using your own instruments and deployments	175
9.4	Advanced issues	175
9.5	Future plans	176
10	Developer's Corner	181
10.1	Introduction	181
10.2	Fundamentals	181
10.3	Delay correction	182
10.4	Channel Modifications	182
10.5	File discovery	190
10.6	ObsMetadata	190
10.7	Logging	191
10.8	Testing	191
11	Technical Documentation	193
11.1	Code Documentation: obsinfo package	193
12	CHANGELOG	229
13	Indices and tables	231
	Python Module Index	233
	Index	235

OBSINFO OVERVIEW

1.1 Introduction

obsinfo is a Python standalone application for creating FDSN-standard (StationXML) data and metadata for ocean bottom seismometers using standardized, easy-to-read information files in [YAML](#) or [JSON](#) formats. The advantages of **obsinfo** are the following:

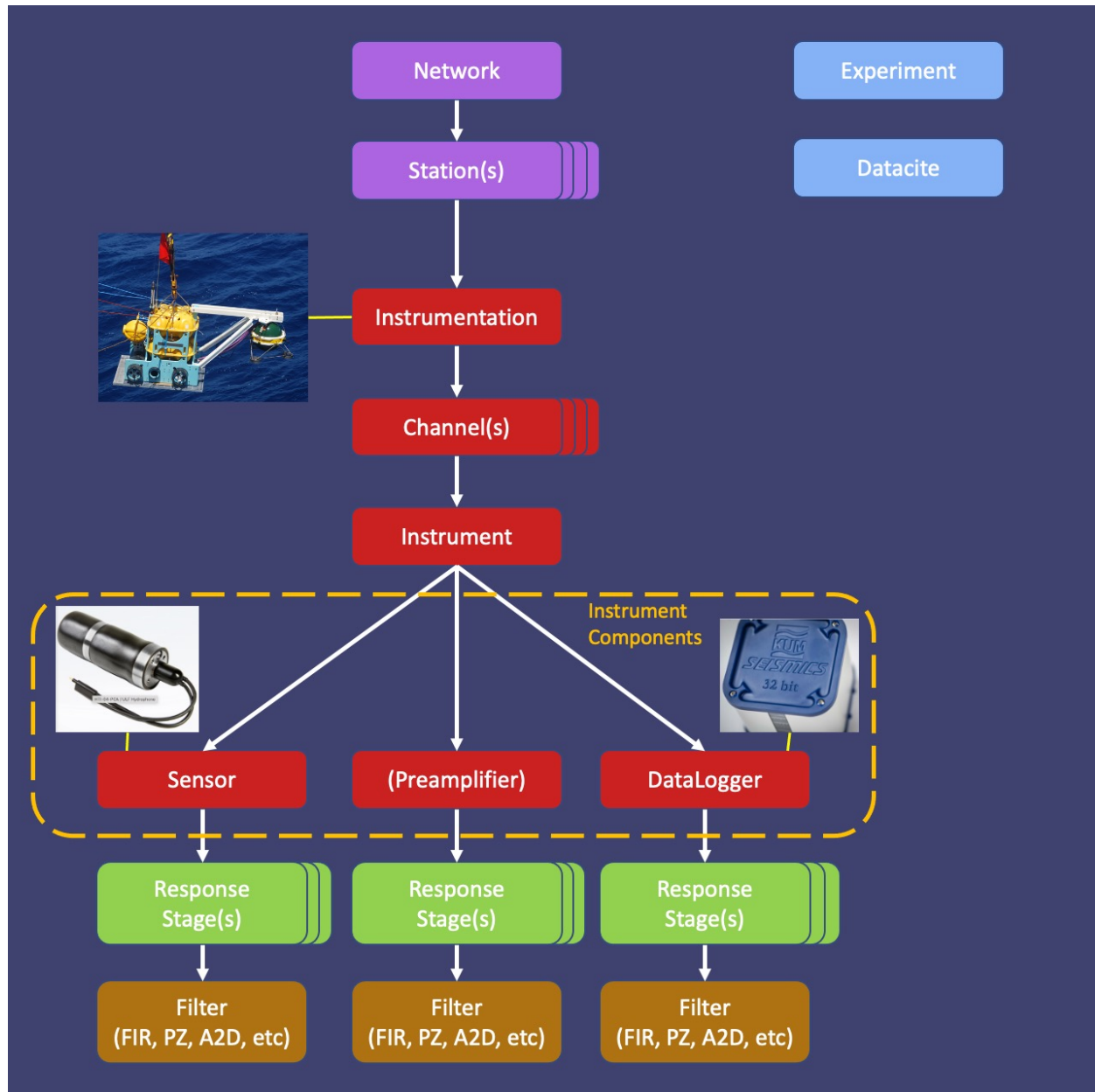
- Easier input user format, as YAML and JSON syntax is simpler than XML
- Flexibility to change configurations avoiding rewriting or modifying a new StationXML file each time (DRY, don't repeat yourself)
- Avoiding redundancy
- Extreme autonomy of operation from Internet connections, being text-based
- Adapting StationXML to OBS-specific functionality
- Simple interface to other systems, such as AROL-based Yasmine

Unlike general systems like Yasmine/AROL, **obsinfo** is conceived to reflect the vast variability of OBS equipment and the fact that changes on the field are common and cannot be planned in advance. Emergency procedures such as changing an equipment component or a stage in the high seas need to be taken into account. The fact that the editing of information needs to occur without necessarily a connection to a central database is a consideration too. Therefore, the choice has been made to avoid a GUI such as Yasmine and make **obsinfo** totally text-based using regular files. At the same time, every effort has been made to stay compatible with AROL, which is based on an earlier version of **obsinfo** itself, v0.106, so AROL yaml files can be imported into **obsinfo**. However, as of this publication, some differences exist which are documented [here](#).

Lack of redundancy and reusability is achieved by creating a repository of instrumentations which is referenced time and again by different campaigns. Flexibility is achieved by permitting these instrumentations to have several selectable configurations. as well as the ability to make punctual changes to *any* field via channel modifications (see [Advanced Topics](#)).

obsinfo also manages two “parallel” information file types: *experiment* and *datacite*. These files are not needed for creating StationXML but are useful for validating data and metadata and for creating DataCite files for DOI. The **datacite_** information file only contains fields that the lead scientist should provide to a data center.

1.2 Object Model



The reference manual is organized around the object model, in pages that describe each particular class. The object hierarchy starts in Class [Network](#). You can navigate from one class to the other by using the **Relationships** section.

1.3 Information Files

The system is based on “*information files*” in **JSON** or **YAML** format as input, filled in by appropriate actors and broken down into different categories to remove redundancy and simplify input as much as possible. Information files are taken as input and converted first to an obsinfo object, which in turn gets converted to an obs_py object, finally output as a StationXML file.

There are 6 main file types for StationXML and processing path creation:

Name	Description	Filled by	When filled
network	Deployed stations, their instruments and parameters	OBS facility	after a campaign
instrumentation	Instrument description	OBS facility	new/changed instruments
<i>instrument_components</i> sensors components dataloggers	Description of basic components	OBS facility -or- component manufacturer	when there are new are new components or calibrations
response_stage	Description of stages (digitizers, amplifiers/filters)	OBS facility -or- component manufacturer	when there are new components or calibrations
filter	Description of filters amplifiers and digitizers used in stages	OBS facility -or- component manufacturer	when there are new components or calibrations

Only the **network** files are OBS-specific.

The **instrumentation** files and their subfiles could be replaced by existing standards such as RESP files or the NRL (Nominal Response Library), but obsinfo provides a simpler and more standards-compliant way to specify the components, and it can automatically calculate response sensitivities based on gains and filter characteristics (using obsPy). obsinfo instrumentation files could also be used to make RESP-files and NRL directories, if so desired.

There are 2 other (optional) file types for quality control and Datacite creation:

Name	Description	Filled by	When filled
experiment	Lists of stations facilities and participants, plus desired verification. NOT NEEDED FOR PROCESSING	Chief scientist	after a data collection campaign
datacite	Scientist-specific information for DOI datacite files	Chief scientist	after a data collection campaign

1.3.1 File Hierarchy

It is recommended to arrange the information files in a file hierarchy such as this:

```
/authors (contains files used in headers of other information files)
/network
/instrumentation
/sensors
/preamplifiers
/dataloggers
```

(continues on next page)

(continued from previous page)

```
[/instrumentation_component]/stages  
[instrumentation_component]/stages/filters
```

where [instrumentation_component] = sensors, preamplifiers or dataloggers.

The hierarchy is completely up to the user, including the names of the folders/directories. In fact, it is perfectly possible to put all the information in a single file, although it is not recommended as reusability of filters, stages or components depends on independent files.

To reference a file from within another file, use the *JREF* syntax:

```
authors: - {$ref: "../authors/Wayne_Crawford.author.yaml#author"}
```

The effect of this directive is at the core of the philosophy of *obsinfo*, as it is this mechanism which allows reuse: it substitutes the content of the key `author` within the file `../authors/Wayne_Crawford.author.yaml` as the value of the key `authors`. If you want to include the complete file, remove the `#author` anchor.

1.3.2 File Naming Convention

While there is flexibility about the folder hierarchy, information files **must** follow the following naming convention:

<descriptive file name>.<obsinfo file type>.<file format>

where

<descriptive file name> usually includes vendor and configuration shorthand to make the file easily identifiable by users,

<obsinfo file type> is one of `campaign`, `network`, `instrumentation`, `sensor`, `preamplifier`, `datalogger`, `stage`, `filter`

<file format> is one of `yaml`, `yaml` or `json`.

Examples:

- `TI_ADS1281_FIR1.stage.yaml` is a stage with a Texas Instruments FIR filter nested in it, in YAML format.
- `BBOBS.INSU-IPGP.network.json` is a network of broad-band stations deployed by INSU IPGP, in JSON format.

There are three resources to look up the exact syntax for each information file. One is the *Introduction* which takes you step by step building a typical hierarchy of information files. The different *Classes* pages have a complete explanation of all the attributes in the class. Unless otherwise noted, attributes in the file have the same syntax as attributes in the class. The third resource is the formal syntax of the file, which is a JSON schema, which is always referenced in the Class page.

1.3.3 File Metadata

All information files contain common metadata

- **format_version:** - This is a required field. It reflects the template version against which this file must be validated
- **revision:** - Revision information (date in particular) to keep change control of file.
 - **date:** - date of revision
 - **authors:** - authors of revision, usually a reference to an author file

- **notes:** - Optional extra information which will not be put in final metadata.
- **yaml_anchors:** - YAML anchors to avoid redundancy, for advanced YAML users. [Here](#) is a guide on how to use YAML anchors.

1.4 Python architecture

1.4.1 Executables

The following command-line executables perform the main tasks:

- **makeSTATIONXML:** generates StationXML files from a network + instrumentation information files

```
$ python3 makeStationXML -h
```

displays all the options of makeStationXML.

To create a StationXML file from a file called <filename>, type:

```
$ python3 makeStationXML.py [options] filename
```

- **obsinfo-validate:** validates an information file against its schema
- **obsinfo-print:** prints a summary of an information file

The following command-line executables make scripts to run specific data conversion software:

- **obsinfo-make_LCHEAPO_scripts:** Makes scripts to convert LCHEAPO data to miniSEED
- **obsinfo-make_SDPCHAIN_scripts:** Makes scripts to drift correct miniSEED data and package them for FDSN-compatible data centers

1.4.2 Package and Modules

The package name is **obsinfo**

obsinfo.main contains code to initialize the main obsinfo routine, to read and potentially validate main (network) information file and to write StationXML file

obsinfo.network and **obsinfo.instrumentation** contain the main code to process the corresponding information files.

obsinfo.OBSMetadata contains code to read and validate information files in either YAML or JSON formats.

obsinfo.misc contains miscellaneous code, currently deprecated and unsupported, which is not used anymore in the application

obspsy.addons contains modules specific to proprietary systems:

- **obspsy.addons.LCHEAPO** creates scripts to convert LCHEAPO OBS data to miniSEED using the 1c2ms software
- **obspsy.addons.SDPCHAIN** creates scripts to convert basic miniSEED data to OBS-aware miniSEED using the SDPCHAIN software suite
- **obspsy.addons.OCA** creates JSON metadata in a format used by the Observatoire de la Cote d’Azur to create StationXML

1.4.3 Auxiliary subdirectories

obsinfo/data/schema

data/schema contains JSON Schema for each file type.

obsinfo/_examples/

Contains example information files and scripts:

- `_examples/Information_Files` contains a complete set of information files
 - `_examples/Information_Files/network` contains **network** files
 - `_examples/Information_Files/instrumentation` contains **instrumentation**, **instrument_components**, **response** and **filter** files.
- `_examples/scripts` contains bash scripts to look at and manipulate these files using the executables. Running these scripts is a good way to make sure your installation works, looking at the files they work on is a good way to start making your own information files.

obsinfo/tests/

Contains test cases and code using `unittest.py`. The tests are performed either on the information files under `test/data` or on `_examples`.

1.4.4 Comments on versioning

We use standard MAJOR.MINOR.MAINTENANCE version numbering but, while the system is in prerelease:

- MAJOR==0
- MINOR increments every time the information file structure changes in a **non-backwards-compatible** way
- MAINTENANCE increments when the code changes or the file structure changes in a **backwards-compatible** way

Current version: v0.110

1.5 Resources

Installation and Startup Guide <InstallStartup>

Tutorial <Tutorial1>

Class Reference Pages <class11>

Information File Templates <FILE>

For the YAML specification, see [YAML](#) . For a tutorial, see [YAML Tutorial](#)

For the JSON specification, see [JSON](#) . For a tutorial, see [JSON Tutorial](#)

For the JREF notation, see [JREF Notation](#)

For the StationXML reference, see [FDSN StationXML Reference](#)

2.1 Information File Tree

network | *Network*

- network_info | *FDSNNetwork*
- operator | *Operator*
- station | *Station*
 - processing | *Processing*
 - * clock_correct_leap_second | *LeapSecond*
 - * clock_correct_linear_drift | *LinearDrift*
 - location | *Location*
 - * location_base | *LocationBase*
 - instrumentation(s) | *Instrumentation*
 - * equipment | *Equipment*
 - * channel | *Channel*
 - No label | *Instrument*
 - No label | *InstrumentComponent*
 - sensor | *Sensor*
 - seed_codes | *SeedCodes*
 - preamplifier | *Preamplifier*
 - datalogger | *Datalogger*
 - response_stages | *ResponseStages*
 - stage | *Stage*
 - filter | *Filter*
 - ADConversion | *ADConversion*
 - Analog | *Analog*
 - Coefficients | *Coefficients*
 - Digital | *Digital*

FIR | *FIR*PolesZeros | *PolesZeros*ResponseList | *ResponseList*

Names left of the | symbol are as they appear in information files as labels/keys. Names right of the | are the corresponding classes in the object model and the Python implementation of that model. An empty label means the label does not exist in information files but exists as an object model / Python class.

2.2 Filter Types

- type="ADConversion" | *ADConversion*
- type="Analog" | *Analog*
- type="Coefficients" | *Coefficients*
- type="Digital" | *Digital*
- type="FIR" | *FIR*
- type="PolesZeros" | *PolesZeros*
- type="ResponseList" | *ResponseList*

2.2.1 Network

Description

An OBS network is a seismological network of stations as part of a campaign *stations* in a given campaign.

Python class:

Network

YAML / JSON label:

network Contained in a network file.

Corresponding StationXML structure

Network

Object Hierarchy

Superclass

None

Subclasses

None

Relationships

- Gathers one or more *Stations*
- Is part of a Campaign (not implemented in *obsinfo* as a class).

Attributes

Name	Type	Re- quired	De- fault	Equivalent StationXML	Remarks
net- work_info	<i>FDSNNetwork</i>	Y	<i>None</i>	<i>None</i>	
operator	<i>Operator</i>	Y	<i>None</i>	OperatorFDSN	Not required in Sta- tionXML
stations	Array of <i>Station</i>	Y	<i>None</i>	StationFDSN	
re- stricted_state	List of values: “open”, “closed”, “par- tial”, “unknown”	N	<i>None</i>	<i>None</i>	

JSON schema

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/network.schema.json>

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/definitions.schema.json>

Example

- Part of the network information file https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/_examples/Information_Files/network/SPOBS.INSU-IPGP.network.yaml with `station` content elided:

```
---
format_version: "0.107"

...

revision:
  authors:
    - $ref: 'authors/Wayne_Crawford.author.yaml#author'
  date: "2017-10-04"
network:
```

(continues on next page)

(continued from previous page)

```
operator:
  reference_name: "INSU-IPGP"
  full_name: "INSU-IPGP OBS Park"
campaign_ref_name: "SPOBS"
network_info:
  code: "4G"
  name: "Short period OBSs"
  start_date: "2007-07-01"
  end_date: "2025-12-31"
  description: "Short period OBS network example"
  comments: ["Lucky Strike Volcano, North Mid-Atlantic Ridge"]
stations:
  "LSVW":
    ...

  "LSVE":
    ...
```

- Another example: https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/_examples/Information_Files/network/BBOBS.INSU-IPGP.network.yaml

Class Navigation

==> *Station*

==> *FDSNNetwork*

==> *Operator*

2.2.2 FDSNNetwork

Description

FDSN network contains specifications compatible with the FDSN standards to describe the *Network*. This class is not actually implemented as such in the Python code, its attributes are assigned in the `__init__()` method of the *Network* class.

Python class:

None These attributes are treated in *Network*.

YAML / JSON label:

network_info

Corresponding StationXML structure*None* Individual attributes in this class belong to the Network attribute.**Object Hierarchy****Superclass***None***Subclasses***None***Relationships**

- Is part of the specification of a *Network*

Attributes

Name	Type	Required	Default	Equivalent StationXML	Remarks
code	string	Y	<i>None</i>	<i>code</i>	Codes are assigned by FDSN according to this process
name	string	Y	<i>None</i>	<i>None</i>	Will be added as a Comment
description	string	Y	<i>None</i>	Description	Not required in StationXML
comment	string	N	<i>None</i>	Comment	
start_date	date	Y	<i>None</i>	startDate	Not required in StationXML
end_date	date	Y	<i>None</i>	endDate	Not required in StationXML

JSON schema
<https://www.gitlab.com/obsinfo/obsinfo/data/schemas/network.schema.json>
<https://www.gitlab.com/obsinfo/obsinfo/obsinfo/data/schemas/definitions.schema.json>

Example

FDSN Network_info section referred in network information file.

```
network_info:
  code: "4G"
  name: "Short period OBSs"
  start_date: "2007-07-01"
  end_date: "2025-12-31"
  description: "Short period OBS network example"
  comments: ["Lucky Strike Volcano, North Mid-Atlantic Ridge"]
```

Class Navigation

Network <==

2.2.3 Operator

Description

Operator specifies the operator of the network, corresponding to the operator field in StationXML. It is also copied to the operator field in station in StationXML.

Python class:

Operator

YAML / JSON label:

operator

Corresponding StationXML structure

Operator Both in Network and Station.

Object Hierarchy

Superclass

None

Subclasses

None

Relationships

- Is part of a *Network*

Attributes

Name	Type	Re- quired	De- fault	Equivalent StationXML	Remarks
reference_name	string	Y	<i>None</i>	Operator. Agency	Operating agency abbreviation
full_name	number	Y	<i>None</i>	Operator.Contact. Name	Operating agency full name, used in contact name
contact_name	number	Y	<i>None</i>	Operator.Contact. Name	
email	email	Y	<i>None</i>	Operator.Contact. Email	Not required in StationXML
phone_number	string	N	<i>None</i>	Operator.Contact. Phone	
website	URL	N	<i>None</i>	Operator. WebSite	

JSON schema

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/network.schema.json>

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/definitions.schema.json>

Example

Operator section in network information file https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/_examples/Information_Files/network/SPOBS.INSU-IPGP.network.yaml

```
operator:
  reference_name: "INSU-IPGP"
  full_name: "INSU-IPGP OBS Park"
```

Class Navigation

Network <==

Instrumentation <==

2.2.4 Station

classes/

Description

An OBS station is an actual implementation of one or several *instrumentation* in a given campaign and *network*.

Python class:

Station

YAML / JSON label:

stations Pertaining to Network Information File

Corresponding StationXML structure

Station

Object Hierarchy

Superclass

None

Subclasses

None

Relationships

- Implements one or several *Instrumentations*
- Is part of a *Network*
- Is in one or several *Locations*

Attributes

Name	Type	Re- quired	De- fault	Equiva- lent Sta- tionXML	Remarks
<i>code</i>	string	Y	<i>None</i>	Station. <i>code</i>	Code does not appear as a YAML/JSON attribute. It's simply the key of the station.
site	string	Y	<i>None</i>	Site	
location_code	string	Y	<i>None</i>	alternate- Code	See class <i>Location</i> for details
locations	Array of <i>Location</i>	Y	<i>None</i>	<i>Operator</i>	The use of locations is not simple. See class <i>Location</i> for details
instrumentation	<i>Instrumentation</i>	Y	<i>None</i>	<i>None</i>	This is mostly contained in attribute Channel in StationXML
processing	<i>Processing</i>	N	<i>None</i>	Description	Will appear appended to Description in StationXML
restricted_state	List of values: "open", "closed", "partial", "unknown"	N	"unknown"	<i>None</i>	
comments	string	N	<i>None</i>	Comment	
start_date	date	N	<i>None</i>	startDate	All three instrument components have these dates. They're the same for all the station.
end_date	date	N	<i>None</i>	endDate	All three instrument components have these dates. They're the same for all the station.

JSON schema

<https://gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/station.schema.json>

Example

- Part of network information file https://gitlab.com/resif/obsinfo/-/tree/master/obsinfo/_examples/Information_Files/network/SPOBS.INSU-IPGP.network.yaml corresponding to the stations, with an example of use of the anchor &LINEAR_CLOCK_DEFAULT. The instrumentations parts are explained under *Instrumentation*.

```

yaml_anchors:
  obs_clock_correction_linear_defaults: &LINEAR_CLOCK_DEFAULTS #Definition of the_
↪ anchor as the next three attributes.
    time_base: "Seascan MCX0, ~1e-8 nominal drift"
    reference: "GPS"
    start_sync_instrument: 0

...

stations:
  "LSVW":
    site: "Lucky Strike Volcano West"

```

(continues on next page)

(continued from previous page)

```

start_date: "2015-04-22T12:00:00Z"
end_date: "2016-05-28T21:01:00Z"
location_code: "00"
instrumentations:
  -
    base:
      $ref: "instrumentation/SPOBS2.instrumentation.yaml#instrumentation"
      datalogger_config: "125sps"
locations:
  "00":
    base: {$ref: 'location_bases/SURFACE_DROP.location_base.yaml#location_
↪base'}
    position: {lon: -32.32504, lat: 37.29744, elev: -2030}
processing:
  - clock_correction_linear_drift:
    <<: *LINEAR_CLOCK_DEFAULTS
    start_sync_reference: "2015-04-21T21:06:00Z"
    end_sync_reference: "2016-05-28T20:59:00.32Z"
    end_sync_instrument: "2016-05-28T20:59:03Z"

"LSVE":
  site: "Lucky Strike Volcano East"
  start_date: "2015-04-22T12:00:00Z"
  end_date: "2016-05-28T21:01:00Z"
  location_code: "00"
  instruments:
    -
      base:
        $ref: "instrumentation/SPOBS2.instrumentation.yaml#instrumentation"
        datalogger_config: "125sps"
  locations:
    "00":
      base: {$ref: 'location_bases/ACOUSTIC_SURVEY.location_base.yaml
↪#location_base'}
      position: {lon: -32.02504, lat: 37.25744, elev: -2130}
  processing:
    - clock_correct_linear_drift:
      <<: *LINEAR_CLOCK_DEFAULTS
      start_sync_reference: "2015-04-21T21:06:00Z"
      end_sync_reference: "2016-05-28T20:59:00.32Z"
      end_sync_instrument: "2016-05-28T20:59:01Z"

```

Class Navigation

Network <==> Instrumentation

==> Location

==> Processing

2.2.5 Instrumentation

Description

An OBS instrumentation is an ensemble of **instruments** associated with specific **channels** constitute a physical unity that will be launched and recovered as a unit. While *obsinfo* is concerned only with the signal processing aspects of the instrumentation, an OBS instrumentation also includes the physical parts of the OBS frame, ballast elements, recovery devices, communication and power supply.

Channels in the instrumentation all have string labels, which are usually channel numbers. They must specify an orientation. Default chann properties can be specified with a label `default`. These properties are common to all channels *unless* overridden by attributes present in specific channels. For example, if a sensor **X** appears under the label `default` but a sensor **Y** appears under the label “2” then for channel 2 the selected sensor will be **Y**. If a preamplifier **Z** is specified under the label `default` and no preamplifier is specified under the label “2”, then channel 2 will have preamplifier **Z**. All attributes can be specified under the default label.

Python class:

Instrumentation

YAML / JSON label:

instrumentation Contained in an instrumentation file

Corresponding StationXML structure

None At the Station level StationXML documents the total number of channels and the selected number of channels. Both are equal in OBS and are calculated implicitly.

Object Hierarchy

Superclass

None

Subclasses

None

Relationships

- Is used in a *Station*
- Has one or several *Channels*
- Has one or specifications defined in *Equipment*

Attributes

Name	Type	Re- quired	De- fault	Equivalent XML	Sta-	Remarks
equipment	<i>Equipment</i>	Y	<i>None</i>	<i>None</i>		
channels	Array of <i>Channel</i>	Y	<i>None</i>	Channel		
channel_modifications	Array of <i>Channel</i>	Y	<i>None</i>	Channel		See <i>AdvancedTopics</i> for details

The attribute `channel_modifications` is used to modify the attributes of a channel. In particular, instruments are supposed to be a rather static database of components and their configurations, but occasionally it is necessary to change some of the attributes for particular campaigns.

Under this keyword the user can specify a complete hierarchy down to the filter level. Only the value(s) specified will be modified. So if a “leaf” value is changed, such as gain value, only the gain value for a particular stage will be changed. But if a complete sensor is specified, the whole component along with its stages and filters will be modified. For more details, see *AdvancedTopics*.

JSON schema

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/instrumentation.schema.json>

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/definitions.schema.json>

Example

YAML code for instrumentation information file https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/_examples/Information_Files/instrumentation/BBOBS1_2012%2Binstrumentation.yaml with the `channel_template` and `das_channels` parts elided.

```
---
format_version: "0.110"
revision:
  date: "2019-12-19"
  authors:
    - {$ref: "authors/Wayne_Crawford.author.yaml#author"}
    - {$ref: "authors/Romuald_Daniel.author.yaml#author"}
instrumentation:
```

(continues on next page)

(continued from previous page)

```

operator: {$ref: "operators/INSU-IPGP.operator_info.yaml#operator_info"}
equipment:
  model: "BBOBS1"
  type: "Broadband Ocean Bottom Seismometer"
  description: "LCHEAPO 2000 BBOBS 2012-present"
  manufacturer: "Scripps Inst. Oceanography - INSU"
  vendor: "Scripps Inst. Oceanography - UNSU"

channels:
  default:
    ...

  "1":
    ...

  "2":
    ...

  "3":
    ...

  "4":
    ...

```

Class Navigation

Station <==> Channel

2.2.6 Equipment

Description

Equipment class describes the characteristics of a particular instrumentation or instrument component, such as the vendor, model, serial number and calibration dates.

Python class:

Equipment

YAML / JSON label:

equipment

Corresponding StationXML structure

- Equipment
- Datalogger
- Sensor
- Preamplifier

All of these classes have the same attributes as the *obsinfo* class, which are populated from the `Equipment` subclass of `Instrumentation`, `Sensor`, `Preamplifier` and `Datalogger` in *obsinfo*

Object Hierarchy

Superclass

None

Subclasses

None

Relationships

Belongs to:

- Instrumentation
- Datalogger
- Sensor
- Preamplifier

Attributes

Name	Type	Required	Default	Equivalent StationXML	Remarks
type	string	Y	<i>None</i>	type	Not required in StationXML
description	string	Y	<i>None</i>	Description	Not required in StationXML
manufacturer	string	Y	<i>None</i>	manufacturer	Not required in StationXML
model	string	Y	<i>None</i>	model	Not required in StationXML
vendor	string	N	<i>None</i>	vendor	Not required in StationXML
serial_number	string	N	<i>None</i>	serial_number	Not required in StationXML
installation_date	date	N	<i>None</i>	startDate	Not required in StationXML
remove_date	date	N	<i>None</i>	endDate	Not required in StationXML
calibration_date	date	N	<i>None</i>	calibDate	Not required in StationXML

- (str):
 - vendor (str):
 - serial_number (str):
 - resource_id (str):
 - obspy_equipment (object of class Equipment from *obsinfo.core.inventory.equipment*)

JSON schema

<https://www.gitlab.com/obsinfo/obsinfo/data/schemas/network.schema.json>

<https://www.gitlab.com/obsinfo/obsinfo/obsinfo/data/schemas/definitions.schema.json>

Example

FDSN Network_info section referred in network information file.

```
network_info:
  code: "4G"
  name: "Short period OBSs"
  start_date: "2007-07-01"
  end_date: "2025-12-31"
  description: "Short period OBS network example"
  comments: ["Lucky Strike Volcano, North Mid-Atlantic Ridge"]
```

Class Navigation

Network <==

2.2.7 Channel

Description

An OBS channel is an *Instrument* plus an orientation. An *Instrumentation* complex consists of one or several channels, each one implementing the signal processing of an instrument.

Actual channels all have string labels, which are usually channel numbers. They must specify an orientation. Default channel properties can be specified with a label `default`. This is not an actual channel. These properties are common to all channels *unless* overridden by attributes present in specific channels. For example, if a sensor **X** appears under the label `default` but a sensor **Y** appears under the label “2” then for channel 2 the selected sensor will be **Y**. If a preamplifier **Z** is specified under the label `default` and no preamplifier is specified under the label “2”, then channel 2 will have preamplifier **Z**. All attributes can be specified under the `default` label.

Configurations are defined at the instrument component level, but are selected at the channel level. A configuration selection attribute specifies a configuration for each of the three instrument components in a channel: sensor, preamplifier and datalogger. They are the attributes `sensor_configuration`, `preamplifier_configuration` and `datalogger_configuration`, respectively.

Python class:

Channel

YAML / JSON label:

- channels
- default
- Particular string labels for each channel

Channels are part of the instrumentation information file.

Corresponding StationXML structure

Channel

Object Hierarchy

Superclass

None

Subclasses

None

Relationships

- Implements the signal of an *Instrument*
- Is part of an *Instrumentation*

Attributes

Name	Type	Required	Default	Equivalent StationXML	Remarks
instrument	<i>Instrument</i>	Y	<i>None</i>	<i>None</i>	Attributes of instrument in Channel in StationXML
orientation:	Orientation	Y	<i>None</i>	<i>None</i>	
• orientation_code	List of values: X,Y,Z,1,2,3,H	Y	<i>None</i>	<i>None</i>	If orientation code is 1,2,3 or H it must include as a dictionary azimuth and dip. See example.
• azimuth	number	Y	<i>None</i>	Channel. <i>Azimuth</i>	0.0 Azimuth < 360.0 in degrees
• dip	number	Y	<i>None</i>	Channel. <i>Dip</i>	-90.0 Dip 90.0 in degrees
location_code	string	Y	<i>None</i>	<i>locationCode</i>	See <i>Location-Base</i> for details of how location codes are used.
sensor_configuration	string	N	<i>None</i>	<i>None</i>	For <i>obsinfo</i> use only. This selects one of the configurations defined in the instrument components.
preamplifier_configuration	string	N	<i>None</i>	<i>None</i>	For <i>obsinfo</i> use only. This selects one of the configurations defined in the instrument components.
datalogger_configuration	string	N	<i>None</i>	<i>None</i>	For <i>obsinfo</i> use only. This selects one of the configurations defined in the instrument components.

Orientation codes are a FDSN standard. By convention, if the orientation code is **X**, **Y** or **Z**, these represent the regular coordinates in space following the right-hand rule, within five degrees of the actual directions. So **X** corresponds to an azimuth of 0° and a dip of 0°, **Y** corresponds to an azimuth of 90° and a dip of 0°, and **Z** corresponds to an azimuth of 0° and a dip of -90° (the positive **Z** direction is towards the bottom). However, if **1**, **2** or **3** are specified, these represent three linearly independent directions but not necessarily coincidental with the regular coordinates, so an azimuth and a dip _must_ be specified. The same is true of the **H** (hydrophone) code.

JSON schema

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/instrumentation.schema.json>

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/definitions.schema.json>

Example

Channels part of instrumentation information file https://gitlab.com/resif/obsinfo/-/tree/master/obsinfo/_examples/Information_Files/instrumentation/BBOBS1_2012%2B.instrumentation.yaml

```
channels:
  default:
    datalogger: {$ref: "dataloggers/LC2000.datalogger.yaml#datalogger"}
    preamplifier:
      {$ref: "preamplifiers/LCHEAPO_BBOBS.preamplifier.yaml#preamplifier"}
    sensor: {$ref: "sensors/NANOMETRICS_T240_SINGLESIDED.sensor.yaml#sensor"}

    preamplifier_configuration: "0.225x"

  "1": {orientation_code: {"2": {azimuth.deg: [90, 0]}}}
  "2": {orientation_code: {"1": {azimuth.deg: [0, 0]}}}
  "3":
    orientation_code: "Z"
    preamplifier_configuration: "1x"
  "4":
    orientation_code : {"H": {azimuth.deg: [0,0], dip.deg: [90,0]}}
    preamplifier: {$ref: "preamplifiers/LCHEAPO_DPG.preamplifier.yaml#preamplifier"}
    sensor: {$ref: "sensors/SIO_DPG.sensor.yaml#sensor"}
```

Class Navigation

Instrumentation <==> *Instrument*

2.2.8 Instrument

Description

An OBS instrument (measurement instrument) records one physical parameter. It is composed of a *Sensor*, an optional *Preamplifier* and a *Datalogger*.

Python class:

Instrument

YAML / JSON label:

None Conceptually, the three instrument components are gathered under an instrument, which has a class in Python. However, as shorthand, we **omit** the instrument label in information files and list the `sensor`, `preamplifier` and `datalogger` components directly under `channel`.

Corresponding StationXML structure

None Attributes of an instrument are assigned to a Channel.

Object Hierarchy

Superclass

None

Subclasses

None

Relationships

- Is assigned to a *Channel*
- Composed of a *Sensor*, an optional *Preamplifier* and a *Datalogger*

Attributes

Name	Type	Required	Default	Equivalent StationXML	Remarks
sensor	<i>Sensor</i>	Y	<i>None</i>	Sensor	
preamplifier	<i>Preamplifier</i>	N	<i>None</i>	Preamplifier	
datalogger	<i>Datalogger</i>	Y	<i>None</i>	Datalogger	

JSON schema

None

Example

None

Class Navigation

Channel <==> *InstrumentComponent*

2.2.9 InstrumentComponent

Description

An *Instrument* in *obsinfo* is broken down into three components: *Sensor*, an optional *Preamplifier* and a *Datalogger*. All of them are subclasses of this class, with some specialized attributes.

What characterizes all components is that they have an *ordered* list of response stages, along with different configuration definitions. The idea is to specify all regularly used configurations (you can always add more later). These different configuration override selected default attributes at the stage and filter level, or add new attributes to them. In turn, the information files at the instrumentation level select one particular configuration definition, and thus, one set of overrides and additions. Configurations are usually labeled with a code which specifies the main characteristic that changes in a particular set of configurations, such as sample rate or gain; in general, We present examples for three different ways to characterize configuration definitions.

On the other hand, it is very important to realize that stages must be specified in order. The top level order is sensor - preamplifier - datalogger, but within these three components it is up to the user to make sure the stages are in the correct order, starting with the one closer to the sensor.

The class *InstrumentComponents* does not appear explicitly in YAML or JASON files but it's part of the object model.

Python class:

InstrumentComponents

YAML / JSON label:

None

Corresponding StationXML structure

None

There are structures in StationXML for *Sensor*, *Preamplifier* and a *Datalogger*.

Object Hierarchy

Superclass

None

Subclasses

- *Sensor*
- *Preamplifier*
- *Datalogger*

Relationships

- Contains *ResponseStages*
- Specs are described in *Equipment*
- Is part of an *Instrument*

Attributes

Name	Type	Re- quired	De- fault	Equiva- lent Sta- tionXML	Remarks
equip- ment	<i>Equip- ment</i>	Y	<i>None</i>	<i>None</i>	In StationXML this is at the channel level: there is only one equipment.
re- sponse_stages	<i>Respons- Stages</i>	N	<i>None</i>	<i>Response</i>	
con- figura- tion_default	string	N	<i>None</i>	<i>None</i>	This is for <i>obsinfo</i> functionality only, if configuration is not specified at the instrumentation level
con- figura- tion_definition_attributes	Stage and filter attributes	N	<i>None</i>	<i>None</i>	A dictionary of stage and filter attributes to override the default configuration. A particular configuration definition may be selected at the instrumentation level

JSON schema

None

Example

None

Navigation

Instrument <==

==> *Sensor*

==> *Preamplifier*

==> *Datalogger*

2.2.10 Sensor

Description

A sensor is an *InstrumentComponent* belonging to an *Instrument*. It models an OBS sensor and so is the generator of the signal being processed. Inheriting from *InstrumentComponent*, it has all its attributes plus the ones below.

Python class:

Sensor

YAML / JSON label:

sensor Sensor usually has its own information file (best practice)

Corresponding StationXML structure

Channel.Sensor

Object Hierarchy

Superclass

InstrumentComponent

Subclasses

None

Relationships

- Contains *ResponseStages*
- Is part of an *Instrument*

Attributes

Name	Type	Required	Default	Equivalent StationXML	Remarks
seed_codes	<i>SeedCodes</i>	Y	<i>None</i>	Channel. code	See explanation in class <i>SeedCodes</i> Only first two codes set here. Orientation set at channel level.

For the rest of attributes, see superclass :ref: `InstrumentComponent <InstrumentComponent>`

JSON schema

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/sensor.schema.json>

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/definitions.schema.json>

Example

From sensor information file https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/_examples/Information_Files/sensors/NANOMETRICS_T240_SINGLESIDED.sensor.yaml (complete file).

```
---
format_version: "0.110"
revision:
  date: "2017-11-30"
  authors:
    - {$ref: "authors/Wayne_Crawford.author.yaml#author"}
sensor:
  equipment:
    model: "Trillium T240"
    type: "Broadband seismometer"
    description: "Trillium T240 seismometer, single-sided connection"
    manufacturer: "Nanometrics, Inc"
    vendor: "Nanometrics, Inc"

  seed_codes:
    band_base: "B"
    instrument: "H"

  configuration_default: "SINGLE-SIDED_SN1-399"
```

(continues on next page)

(continued from previous page)

```

configuration_definitions:
  "SINGLE-SIDED_SN1-399" :
    equipment:
      description: "negative shorted to ground, serial numbers 1-399"
    response_stages:
      - $ref: "responses/Trillium_T240_SN1-399-singlesided_theoretical.stage.yaml
↪ #stage"
  "SINGLE-SIDED_SN400plus" :
    equipment:
      description: "negative shorted to ground, serial numbers 400+"
    response_stages:
      - $ref: "responses/Trillium_T240_SN400-singlesided_theoretical.stage.yaml
↪ #stage"

notes:
  - "INSU-IPGP OBS park sphere sensor pairs are: Sphere01-133, Sphere02-132,"
  - "Sphere03-134, Sphere04-138, Sphere05-137, Sphere06-830, Sphere07-136,"
  - "Sphere08-829, Sphere09-826"

```

Class Navigation

InstrumentComponent <==> *ResponseStages*

2.2.11 SeedCodes

Description

Seed Codes are defined by the FDSN to characterize *channels* according to their data sources and signal treatment characteristics.

Python class:

SeedCodes

YAML / JSON label:

seed_codes

Corresponding StationXML structure

Channel.code

Object Hierarchy

Superclass

None

Subclasses

None

Relationships

- Belongs to an *Sensor*

Attributes

Name	Type	Re- quired	De- fault	Equiv- alent Sta- tionXML	Remarks
band	string with restrictions (1 char)	Y	<i>None</i>	<i>None</i>	For a complete explanation of codes, click on band-code . Called band code in the new FDSN nomenclature.
instrument_code	string with restrictions (1 char)	Y	<i>None</i>	<i>None</i>	For a complete explanation of codes, click on instrument-code . Called source code in the new FDSN nomenclature.
orientation_code	string with restrictions (1 char)	N	<i>None</i>	<i>None</i>	For a complete explanation of codes, click orientation-code . See subsection <i>Geographic orientation subsource codes</i> . This code is assigned at the channel level. Called subsurface orientation code in the new FDSN nomenclature. While part of the seed code, it is assigned at the channel level.

JSON schema

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/sensor.schema.json>

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/definitions.schema.json>

Example

Class Navigation

Sensor <==

2.2.12 Preamplifier

Description

An optional preamplifier may be part of an OBS instrument. It is an *InstrumentComponent* with response stages and no particular attributes of its own.

Python class:

Preamplifier

YAML / JSON label:

preamplifier

Corresponding StationXML structure

Preamplifier

Object Hierarchy

Superclass

InstrumentComponent

Subclasses

- *Filter*

Relationships

- Is element of *Equipment*
- Contains *Response Stages*

Attributes

None

For the rest of attributes, see superclass :ref: `InstrumentComponent <InstrumentComponent>`

JSON schema

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/preamplifier.schema.json>

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/definitions.schema.json>

Example

Preamplifier information file https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/_examples/Information_Files/preamplifiers/LCHEAPO_BBOBS.preamplifier.yaml (complete file)

```
---
format_version: "0.110"
revision:
  date: "2017-11-30"
  authors:
    - $ref: "authors/Wayne_Crawford.author.yaml#author"

preamplifier:
  equipment:
    model: "BBOBS-GAIN"
    type: "Analog gain card"
    description: "INSU BBOBS gain card"
    manufacturer: "SIO or IPGP"
    vendor: ~

  configuration_default: "1x"

  configuration_definitions:
    "0.225x":
      config_description: "0.225x gain"
      response_stages:
        - $ref: "responses/INSU_BBOBS_gain0.225_theoretical.stage.yaml#stage"
    "1x":
      config_description: "1x gain"
      response_stages:
        - $ref: "responses/INSU_BBOBS_gain1.0_theoretical.stage.yaml#stage"
```

Class Navigation

InstrumentComponent <==> *ResponseStages*

2.2.13 Datalogger

Description

A datalogger is the part of an OBS instrument which records the signal after processing. It is an *InstrumentComponent* with response stages and attributes such as the global delay correction and the overall sample rate of the instrument.

Python class:

Datalogger

YAML / JSON label:

datalogger

Corresponding StationXML structure

Datalogger

Object Hierarchy

Superclass

InstrumentComponent

Subclasses

None

Relationships

- Is element of *Equipment*
- Contains *Response Stages*

Attributes

Name	Type	Required	Default	Equivalent StationXML	Remarks
sample_rate	number	Y	None	None	
delay_correction	number	N	None	None	This is used by <i>obsinfo</i> to calculate StationXML correction for each stage. If None, correction = delay; if not None, correction for each stage is 0 except the last, which is the value of delay_correction.

For the rest of attributes, see superclass :ref: `InstrumentComponent` <InstrumentComponent>`

JSON schema

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/datalogger.schema.json>

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/definitions.schema.json>

Example

Datalogger information file https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/_examples/Information_Files/dataloggers/LC2000.datalogger.yaml :

```
--
format_version: "0.110"
revision:
  date: "2019-12-20"
  authors:
    - $ref: 'authors/Wayne_Crawford.author.yaml#author'
notes:
  - "Delay correction is hard-coded to 29 samples in LCHEAPO software"

datalogger:
  equipment:
    model: "CS5321/22"
    type: "delta-sigma A/D converter + digital filter"
    description: "CS5321/22 delta-sigma A/D converter + FIR digital filter"
    manufacturer: "Cirrus Logic"
    vendor: "various"
  configuration_default: "125 sps"

  configuration_definitions:
    "62.5sps":
      config_description: "62.5 sps"
      sample_rate: 62.5
      delay_correction: 0.464
      response_stages:
        - $ref: "responses/CS5321_FIR1.stage.yaml#stage"
        - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
        - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
```

(continues on next page)

(continued from previous page)

```

- $ref: "responses/CS5322_FIR2.stage.yaml#stage"
- $ref: "responses/CS5322_FIR2.stage.yaml#stage"
- $ref: "responses/CS5322_FIR2.stage.yaml#stage"
- $ref: "responses/CS5322_FIR2.stage.yaml#stage"
- $ref: "responses/CS5322_FIR2.stage.yaml#stage"
- $ref: "responses/CS5322_FIR2.stage.yaml#stage"
- $ref: "responses/CS5322_FIR3.stage.yaml#stage"
"125sps":
  config_description: "125 sps"
  sample_rate: 125
  delay_correction: 0.232
  response_stages:
    - $ref: "responses/CS5321_FIR1.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR3.stage.yaml#stage"
"250sps":
  config_description: "250 sps"
  sample_rate: 250
  delay_correction: 0.116
  response_stages:
    - $ref: "responses/CS5321_FIR1.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR3.stage.yaml#stage"
"500sps":
  config_description: "500 sps"
  sample_rate: 500
  delay_correction: 0.058
  response_stages:
    - $ref: "responses/CS5321_FIR1.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR3.stage.yaml#stage"
"1000sps":
  config_description: "1000 sps"
  sample_rate: 1000
  delay_correction: 0.029
  response_stages:
    - $ref: "responses/CS5321_FIR1.stage.yaml#stage"

```

(continues on next page)

(continued from previous page)

```
- $ref: "responses/CS5322_FIR2.stage.yaml#stage"
- $ref: "responses/CS5322_FIR2.stage.yaml#stage"
- $ref: "responses/CS5322_FIR2.stage.yaml#stage"
- $ref: "responses/CS5322_FIR2.stage.yaml#stage"
- $ref: "responses/CS5322_FIR3.stage.yaml#stage"
```

Class Navigation

InstrumentComponent <==> *ResponseStages*

2.2.14 ResponseStages

Description

Stages are discrete units in the block diagram of an electronic circuit which perform a specific function and is usually physically circumscribed to a printed board. An instrument component in *obsinfo* is usually composed of several chained stages which connect the output of one stage to the input of the next one. This class implements the change of individual stages.

It is important that contiguous stages are consistent in two ways:

1. Output units of a stage must be equal to input units of the next stage
2. Output sample rate of a stage must match the input sample rate of the next stage

Python class:

ResponseStages

YAML / JSON label:

response_stages

Corresponding StationXML structure

Response

Object Hierarchy

Superclass

None

Subclasses

None

Relationships

- Contains one or several *Stages*
- Belongs to an *Instrument Component*

Attributes

Name	Type	Required	Default	Equivalent StationXML	Remarks
Stage	Array of <i>Stage</i>	N	None	StageFDSN	

Calculated Attributes

These attributes do not exist in the YAML/JSON file. They are or may be calculated programmatically to feed corresponding values in the StationXML file or for other purposes.

Name	Type	Default	Equivalent StationXML	Remarks
number	number	None	Stage. Number	Calculated depending on position
sensitivity	number	None	InstrumentSensitivity	Calculated with obspy.obspy_Sensitivity using gain.frequency of first stage as reference frequency and then recalculated with frequency out of sensitivity calculation.
total_sample_rate	number	None	None	Calculated as sum of stage sample rates in order to validate against declared sample_rate.

JSON schema

https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/response_stages.schema.json

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/definitions.schema.json>

Example

Response stages part of a datalogger information file https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/_examples/Information_Files/dataloggers/LC2000.datalogger.yaml

```
response_stages:
  - $ref: "responses/CS5321_FIR1.stage.yaml#stage"
  - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
  - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
  - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
  - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
```

(continues on next page)

(continued from previous page)

- **\$ref**: "responses/CS5322_FIR2.stage.yaml#stage"
- **\$ref**: "responses/CS5322_FIR3.stage.yaml#stage"

Class Navigation

InstrumentComponent <==> *Stage*

2.2.15 ResponseList

Description

A *filter* can be characterised by the list of impulse responses it yields, instead of its *transfer function*. These responses are triples of [frequency (in Hz), amplitude, phase (in degrees)], expressed in a list.

Python class:

ResponseList

YAML / JSON label:

ResponseList

Corresponding StationXML structure

ResponseList

Object Hierarchy

Superclass

Filter

Subclasses

None

Relationships

- Is nested in *Stage*

Attributes

Name	Type	Required	Default	Equivalent StationXML	Remarks
elements	Array of Values: [number, number, number] where first element = frequency (in Hz) second element = amplitude third element =: phase (in degrees)	Y	<i>None</i>	ResponseListElement: Frequency Amplitude Phase	

JSON schema

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/filter.schema.json>

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/definitions.schema.json>

Example

No available example.

Class Navigation

Filter <==

2.2.16 Stage

Description

Stages are discrete units in the block diagram of an electronic circuit which perform a specific function and is usually physically circumscribed to a printed board. An instrument component in *obsinfo* is usually composed of several chained stages which connect the output of one stage to the input of the next one.

Python class:

Stage

YAML / JSON label:

Unnamed element of `response_stages` array. The array itself has a label `response_stages`

Corresponding StationXML structure

Stage

Object Hierarchy

Superclass

None

Subclasses

None

Relationships

- Is element of *ResponseStages*
- Nests one *Filter*

Attributes

Name	Type	Required	Default	Equivalent StationXML	Remarks
name	string	N	<i>None</i>	e.g. FIR. name	In StationXML this attribute is at the filter (PZ, Coeff, FIR, etc.) level.
description	string	N	<i>None</i>	e.g. FIR. Description	In StationXML this attribute is in the filter (PolesZeros, Coefficients, FIR, etc.)
input_units	IRISUnits	Y	<i>None</i>	e.g. FIR. InputUnits	In StationXML this attribute is at the filter (PZ, Coeff, FIR, etc.) level.
output_units	IRISUnits	Y	<i>None</i>	e.g. FIR. OutputUnits	In StationXML this attribute is at the filter (PZ, Coeff, FIR, etc.) level.
gain:		Y	<i>None</i>	StageGain	
• frequency	number	Y	<i>None</i>	Frequency	In Hertz
• value	number	Y	<i>None</i>	Value	
filter	Filter	Y	<i>None</i>	<i>None</i>	No filter attribute in StationXML. Individual filters are subsumed in Stage.
calibration_date	date	N	<i>None</i>	<i>None</i>	In StationXML this attribute is only found at the equipment level.
decimation_factor	number	N	1.0	DecimationFactor	
input_sample_rate	number	Y	<i>None</i>	DecimationInputSampleRate	
delay	number	N	0.0	DecimationDelay	If not set, will be calculated as filter.offset / input_sample_rate
polarity	string with values “+” and “-”	Y	<i>None</i>	<i>None</i>	“+” = counts increase when the input voltage increase, “-” otherwise.

Calculated Attributes

These attributes do not exist in the YAML/JSON file. They are or may be calculated programmatically to feed corresponding values in the StationXML file or for other purposes.

Name	Type	De- fault	Equiva- lent Sta- tionXML	Remarks
stage_sequence_number	integer	0	number	
correction	number	0.0	Decimation.CorrectedDelay	This value is calculated as a function of delay_correction in class Datalogger. If delay_correction exists correction=0 for all stages but the last, which has value = delay_correction. If it does not exist, correction = delay.
output_sample_rate	number	0.0	None	

JSON schema

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/stage.schema.json>

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/definitions.schema.json>

Example

Stage information file https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/_examples/Information_Files/instrumentation/dataloggers/responses/CS5321_FIR3.stage.yaml.

```
---
format_version: "0.110"
revision:
  date: "2017-11-30"
  authors:
    - $ref: "authors/Wayne_Crawford.author.yaml#author"
notes: ["From CS5322_Filter.pdf"]
stage:
  decimation_factor : 2
  gain : {value: 1, frequency: 0}
  input_units : { name : "counts", description: "Digital Counts"}

  description : "DECIMATION - CS5322 FIR3 (linear phase)"
  filter:
    $ref: "FIR/CirrusLogic_CS5322_FIR3.filter.yaml#filter"
  extras:
    DBIRD_response_type : "THEORETICAL"
```

Class Navigation

ResponseStages <==> *Filter*

2.2.17 Filter

Description

The class *Filter* describes the different types of filters that process the signal in the stages of the instrument components of an OBS.

Superclass

None

Subclasses

- *PolesZeros*
- *FIR*
 - *Analog*
- *Coefficients*
 - *ADConversion*
 - *Digital*
- *Response List*

Relationships

Is nested in a *Stage*

Attributes

Name	Type	Re- quired	De- fault	Equivalent Sta- tionXML	Remarks
type	string	Y	<i>None</i>	N/A	Possible values: “PolesZeros”, “FIR”, “Coefficients”, “Ana- log”, “Digital”, “ADConversion”
off- set	num- ber	Y	<i>None</i>	Decima- tion.Offset	If delay is not present in the corresponding Stage, it will be set to offset/input_sample_rate

Depending on the *type*, other attributes will be required

JSON schema

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/filter.schema.json>

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/definitions.schema.json>

Example

This is the filter information file https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/_examples/Information_Files/instrumentation/dataloggers/responses/FIR/CirrusLogic_CS5322_FIR3.filter.yaml , which specifies a “FIR”-type filter.

```
---
format_version: "0.110"
revision:
  date: "2017-11-30"
  authors:
    - $ref: "authors/Wayne_Crawford.author.yaml#author"

notes: ["101 coefficients, linear phase filter"]

filter:
  type: "FIR"
  symmetry: "NONE"
  offset: 50
  coefficients:
    - -3.09982E-6
    - -2.94483E-5
    - -9.8002E-5
    - -1.62383E-4
    - -1.00029E-4
    - 1.20655E-4
    - 2.61935E-4
    - 2.52755E-5
    - -4.10488E-4
    - -3.66852E-4
    - 3.7627E-4
    - 8.54597E-4
    - -3.05213E-5
    - -0.00127677
    ...
    - 1.20655E-4
    - -1.00029E-4
    - -1.62383E-4
    - -9.8002E-5
    - -2.94483E-5
    - -3.09982E-6
```

Class Navigation

Stage <==

==> *PolesZeros*

==> *FIR*

==> *Coefficients*

==> *ResponseList*

==> *ADConversion*

==> *Digital*

==> *Analog*

2.2.18 ADConversion

Description

StationXML does not specify analog to digital stages. We implement them as a Coefficients filter with one numerator coefficient, which is equal to one.

Python class:

ADConversion

YAML / JSON label:

ADConversion

Corresponding StationXML structure

Coefficients (with no coefficients except one numerator equal to one)

Object Hierarchy

Superclass

Coefficients

Subclasses

None

Relationships

- Is nested in *Stage*

Attributes

Name	Type	Required	Default	Equivalent StationXML	Remarks
input_full_scale	number	N	<i>None</i>	<i>None</i>	
output_full_scale	number	N	<i>None</i>	<i>None</i>	

JSON schema

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/filter.schema.json>

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/definitions.schema.json>

Example

Filter section in stage information file https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/_examples/Information_Files/instrumentation/dataloggers/responses/CS5321_FIR1.stage.yaml

```
filter:
  type : "AD_CONVERSION"
  input_full_scale : 9 # 9 V pp
  output_full_scale : 10485760 # 4FFFFFF@Vref and B000000@-Vref
```

Class Navigation

Filter <==

2.2.19 Analog

Description

StationXML does not specify Analog stages which do not have filters. They are implemented here as a PZ filter without poles or zeroes.

Python class:

Analog

YAML / JSON label:

Analog

Corresponding StationXML structure

PolesZeros (with no poles or zeros))

Object Hierarchy

Superclass

PolesZeros

Subclasses

None

Relationships

- Is nested in *Stage*

Attributes

Name	Type	Required	Default	Equivalent StationXML	Remarks

JSON schema

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/filter.schema.json>

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/definitions.schema.json>

Example

In stage information file https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/_examples/Information_Files/instrumentation/preamplifiers/responses/INSU_BBOBS_gain0.225_theoretical.stage.yaml

```
filter :
  type : "Analog"
```

Class Navigation

Filter <==

2.2.20 Coefficients

Description

The Coefficients class is a representation of a finite impulse response (FIR) filter, which is a *filter* whose impulse response (or response to any finite length input) is of finite duration, because it settles to zero in finite time. It is used mainly for FIR filters which are not symmetric. A symmetric FIR filter should use the FIR class.

The impulse response (that is, the output in response to a Kronecker delta input) of an Nth-order discrete-time FIR filter lasts exactly $N + 1$ samples (from first nonzero element through last nonzero element) before it then settles to zero. FIR filters can be discrete-time or continuous-time, and digital or analog.

For a more detailed discussion, [click here](#).

Python class:

Coefficients

YAML / JSON label:

Coefficients

Corresponding StationXML structure

Coefficients

Object Hierarchy

Superclass

Filter

Subclasses

- *Digital*
- *ADConversion*

Relationships

- Is nested in *Stage*

Attributes

Name	Type	Required	Default	Equivalent StationXML	Remarks
transfer_function_type	List of values: LAPLACE (RADIANS/SECOND), LAPLACE (HERTZ), DIGITAL (Z-TRANSFORM)	N	LAPLACE (RADIANS/SECOND)	PzTransferFunctionType	More info...
numerator_coefficients	List of numbers	Y	<i>None</i>	Numerator	
denominator_coefficients	List of numbers	Y	<i>None</i>	Denominator	

JSON schema

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/filter.schema.json>

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/definitions.schema.json>

Example

```
---
format_version: "0.110"
filter:
  type: "Coefficients"
  numerator_coefficients:
    [1, 0.1, -0.3, 0.6]
  denominator_coefficients:
    [-0.2, 0.8, 0.4, -0.3]
```


Class Navigation

Filter <==

2.2.21 Digital

Description

StationXML does not have a class for digital stages which are not filters. They are therefore implemented as a Coefficients filter with one numerator coefficient, equal to 1.

Python class:

Digital

YAML / JSON label:

DIGITAL

Corresponding StationXML structure

Coefficients (with no coefficients)

Object Hierarchy

Superclass

Coefficients

Subclasses

None

Relationships

- Is nested in *Stage*

Attributes

Name	Type	Required	Default	Equivalent StationXML	Remarks
<i>None</i>					

JSON schema

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/filter.schema.json>

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/definitions.schema.json>

Example

No existing file for this example.

```
filter:  
  type: "Digital"
```

Class Navigation

Filter <==

2.2.22 FIR

Description

A finite impulse response (FIR) filter is a *filter* whose impulse response (or response to any finite length input) is of finite duration, because it settles to zero in finite time.

The impulse response (that is, the output in response to a Kronecker delta input) of an Nth-order discrete-time FIR filter lasts exactly $N + 1$ samples (from first nonzero element through last nonzero element) before it then settles to zero. FIR filters can be discrete-time or continuous-time, and digital or analog.

Alternatively, FIR filters in *obsinfo* are also commonly documented using the *Coefficients* class, though FIR has the advantage of allowing representation of symmetric FIR coefficients without repeating them.

For a more detailed discussion, [click here](#).

Python class:

FIR

YAML / JSON label:

FIR

Corresponding StationXML structure

FIR

Object Hierarchy**Superclass***Filter***Subclasses***None***Relationships**

- Is nested in *Stage*

Attributes

Name	Type	Required	Default	Equivalent StationXML	Remarks
symmetry	List of values: ODD, EVEN, NONE	Y	<i>None</i>	<i>Symmetry</i>	
coefficients	List of numbers	N	<i>None</i>	<i>NumeratorCoefficient</i>	
coefficient_divisor	number	N	1.0	<i>NOT USED</i>	

JSON schema

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/filter.schema.json>

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/definitions.schema.json>

Example

In filter information file https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/_examples/Information_Files/instrumentation/dataloggers/responses/FIR/TexasInstruments_ADS1281_FIR1.filter.yaml

```
--
format_version: "0.107"
filter:
  type: "FIR"
  symmetry: "NONE"
  delay.samples: 5
  coefficient_divisor: 512
  coefficients:
    - 3
    - 0
    - -25
    - 0
    - 150
    - 256
    - 150
    - 0
    - -25
    - 0
    - 3
```

Class Navigation

Filter <==

2.2.23 PolesZeros

Description

A Pole-Zero *filter*. Every digital filter can be specified by its poles and zeros (together with a gain factor). Poles and zeros give useful insights into a filter's response. For a more detailed discussion, [click here](#).

Python class:

- PolesZeros

YAML / JSON label:

- PolesZeros

Corresponding StationXML structure

No direct correspondence. Mapped into subattribute [PolesZeros](#) of attribute [Stage](#).

Object Hierarchy

Superclass

Filter

Subclasses

- *Analog*

Relationships

- Is nested in *Stage*

Attributes

Name	Type	Required	Default	Equivalent StationXML	Remarks
transfer_function_type	List of Values: LAPLACE (RADIANS/SECOND), LAPLACE (HERTZ), DIGITAL (Z-TRANSFORM)	N	LAPLACE (RADIANS/SECOND)	PzTransferFunctionType	More info...
zeros	List of numbers	Y	<i>None</i>	Zero	
poles	List of numbers	Y	<i>None</i>	Pole	
normalization_frequency	number	N	<i>None</i>	Normalization-Frequency	
normalization_factor	number	N	<i>None</i>	Normalization-Factor	Frequency at which the Normalization-Factor is valid. This should be the same for all stages and within the passband, if any.

JSON schema

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/filter.schema.json>

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/definitions.schema.json>

Example

In the filter information file https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/_examples/Information_Files/instrumentation/sensors/responses/PolesZeros/Trillium_T240_SN400-_generic.filter.yaml .

```
---
format_version: "0.110"
revision:
  date: "2018-06-01"
  authors:
    - {$ref: "authors/Wayne_Crawford.author.yaml#author"}

filter:
  type: "PolesZeros"
  transfer_function_type: "LAPLACE (RADIANS/SECOND)"
  zeros :
    - [0.0, 0.0]
    - [0.0, 0.0]
    - [-72.5, 0.0]
    - [-159.3, 0.0]
    - [-251, 0.0]
    - [-3270.0, 0.0]
  poles :
    - [-0.017699, 0.017604]
    - [-0.017699, -0.017604]
    - [-85.3, 0.0]
    - [-155.4, 210.8]
    - [-155.4, -210.8]
    - [-713, 0]
    - [-1140, -0]
    - [-4300, -0]
    - [-5800, -0]
    - [-4300, 4400]
    - [-4300, -4400]
  offset: 0

notes:
  - poles et zeros d'un Trillium T240 no de serie 400+
  - d'apres le fichier Trillium240_UserGuide_15672R7.pdf de Nanometrics.

extras: None
```

Class Navigation

Filter <==

2.2.24 Processing

Description

This class has no correlate in StationXML, but it is used for important documentation purposes in *obsinfo*, and thus is included as a comment in StationXML. It documents the addition or subtraction of leap seconds to the signal data, and any kind of clock drift. As is well known, OBS equipment does not have a GPS connection and its clock must be manually synchronized.

Python class:

Processing

YAML / JSON label:

processing

Corresponding StationXML structure

None

Object Hierarchy

Superclass

None

Subclasses

None

Relationships

- Is element of *Station*
- Is composed of *LeapSecond*
- Is composed of *LinearDrift*

Attributes

Name	Type	Required	Default	Equivalent StationXML	Remarks
linear_drift	<i>LinearDrift</i>	N	<i>None</i>	<i>None</i>	
leap_second	<i>LeapSecond</i>	N	<i>None</i>	<i>None</i>	

JSON schema

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/network.schema.json>

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/definitions.schema.json>

Example

Processing section of network information file https://gitlab.com/resif/obsinfo/-/tree/master/obsinfo/_examples/Information_Files/network/BBOBS.INSU-IPGP.network.yaml .

```
processing:
  - clock_corrections:
      linear_drift:
        time_base: "Seascan MCX0, ~1e-8 nominal drift"
        reference: "GPS"
        start_sync_instrument: 0
        start_sync_reference: "2015-04-23T11:20:00"
        end_sync_reference: "2016-05-27T14:00:00.2450"
        end_sync_instrument: "2016-05-27T14:00:00"
```

Station <==

==> LinearDrift

==> LeapSecond

2.2.25 LinearDrift

Description

This class has no correlate in StationXML, but it is used for important documentation purposes in *obsinfo*, and thus is included as a comment in StationXML. It documents the clock drift. As is well known, OBS equipment does not have a GPS connection and its clock must be manually synchronized.

Python class:

Processing

The subclass LinearDrift is not implemented in Python but directly as attributes of class Processing

YAML / JSON label:

processing: clock_correct_linear_drift

Corresponding StationXML structure

None

Object Hierarchy

None

Superclass

None

Subclasses

None

Relationships

- Is element of *ProcessingClockCorrections*

Attributes

Name	Type	Re- quired	De- fault	Equivalent StationXML	Remarks
time_base	string	Y	<i>None</i>	<i>None</i>	Time base of OBS
reference	string	Y	<i>None</i>	<i>None</i>	Reference used
start_sync_instrument	times- tamp	Y	<i>None</i>	<i>None</i>	If set to zero or absent, see below in Calculated Attributes.
start_sync_reference	times- tamp	Y	<i>None</i>	<i>None</i>	
end_sync_instrument	times- tamp	Y	<i>None</i>	<i>None</i>	
end_sync_reference	times- tamp	Y	<i>None</i>	<i>None</i>	

Calculated Attributes

Name	Type	De-fault	Sta-tionXML	Remarks
start_sync_instrument	times-tamp	<i>None</i>	<i>None</i>	If set to 0 or absent in the file, it is set equal to the start_sync_reference

JSON schema

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/network.schema.json>

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/obsinfo/-/tree/master/obsinfo/data/schemas/definitions.schema.json>

Example

Processing section of network information file https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/_examples/Information_Files/network/BBOBS.INSU-IPGP.network.yaml .

```
clock_correct_linear_drift:
  time_base: "Seascan MCX0, ~1e-8 nominal drift"
  reference: "GPS"
  start_sync_reference: "2015-04-23T11:20:00"
  end_sync_reference: "2016-05-27T14:00:00.2450"
  end_sync_instrument: "22016-05-27T14:00:00"
```

Processing <==

2.2.26 LeapSecond

Description

This class has no correlate in StationXML, but it is used for important documentation purposes in *obsinfo*, and thus is included as a comment in StationXML. It documents the addition or subtraction of leap seconds to the signal data. As is well known, OBS equipment does not have a GPS connection and its clock must be manually synchronized.

Python class:

Processing

The subclass LeapSecond is not implemented in Python but directly as attributes of class Processing

YAML / JSON label:

clock_correct_leap_second

Corresponding StationXML structure*None***Object Hierarchy****Superclass***None***Subclasses***None***Relationships**

- Is element of *ProcessingClockCorrections*

Attributes

Name	Type	Re- quired	De- fault	Equivalent StationXML	Remarks
time	timestamp	Y	<i>None</i>	<i>None</i>	
type	string (1 char, + or -)	Y	<i>None</i>		A positive leapsecond is a 61 second minute, a negative one, a 59 sec.
cor- rected_in_end_sync	boolean	Y	<i>None</i>	<i>None</i>	
cor- rected_in_end_data	boolean	Y	<i>None</i>		

JSON schema

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/stage.schema.json>

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/definitions.schema.json>

Example

Section in network information file https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/_examples/Information_Files/campaign/LEAP_SECOND.INSU-IPGP.network.yaml .

```
- clock_correct_leapsecond:
    time: "2016-12-31T23:59:60"
    type: "+"
    corrected_in_end_sync: True
```

Class Navigation

Processing <==

2.2.27 Location

Description

This class serves two purposes. If a single location is specified or if the location code “00” is specified, this will be geographic location of the corresponding *Station* in StationXML. The rest of locations serve to group channels that treat the signal of a single sensor. They can be physically in other geographic locations or not.

Python class:

Location

YAML / JSON label:

location:

Corresponding StationXML structure

None

Location codes appear in channels of a given instrumentation. All locations corresponding to these codes are specified as individual attributes in the channel section of StationXML.

Object Hierarchy

Superclass

None

Subclasses

None

Relationships

- Is assigned, as a code, to a *Channel*
- Is assigned to a *Station*

Attributes

Name	Type	Required	Default	Equivalent StationXML	Remarks
<i>code</i>	string	Y	None	location_code	
base	LocationBase	N	None	None	Individual fields of base in Station attribute. See LocationBase for details.
position:	Position	Y	None	None	
• lat	number	Y	None	Latitude	Expressed in degrees/min/sec/fractions of sec.
• lon	number	Y	None	Longitude	Expressed in degrees/min/sec/fractions of sec.
• elev	number	Y	None	Elevation	Expressed in meters

JSON schema

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/network.schema.json>

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/definitions.schema.json>

Example

Facility section in network information file https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/_examples/Information_Files/network/SPOBS.INSU-IPGP.network.yaml

```
locations:
  "00":
    base: {$ref: 'location_bases/SURFACE_DROP.location_base.yaml#location_
↪base'}
    position: {lon: -32.32504, lat: 37.29744, elev: -2030}
```

Class Navigation

Station <==> *Instrumentation*

==> *LocationBase*

2.2.28 LocationBase

Description

LocationBase specifies parameters specific to a type of location.

Python class:

LocationBase

YAML / JSON label:

location_base

Corresponding StationXML structure

None Individual attributes in this class belong to the Station attributes.

Object Hierarchy

Superclass

None

Subclasses

None

Relationships

- Is part of a *Location*

Attributes

Name	Type	Re- quired	De- fault	Equivalent StationXML	Remarks
uncer- tainties	dictionary of {lat: num- ber, lon:number, elev: number}	Y	<i>None</i>	Included in latitude, longitude and elevation (see Class Loca- tion)	In meters. As uncer- tainties.m in YANL / JSON
depth	number	Y	<i>None</i>	<i>None</i>	In meters. As depth.m in YANL / JSON
geology	string	Y	<i>None</i>	Geology	
vault	string	Y	<i>None</i>	Vault	
localisa- tion_method	string	Y	<i>None</i>	<i>None</i>	Added in Comment in StationXML

JSON schema

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/network.schema.json>

<https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/data/schemas/definitions.schema.json>

Example

LocationBase information file https://www.gitlab.com/resif/obsinfo/-/tree/master/obsinfo/_examples/Information_Files/location_b

```

---
format_version: "0.107"
revision:
  date: "2018-06-01"
  authors:
    - $ref: "Wayne_Crawford.author.yaml#author"
location_base:
  depth.m: 0
  geology: "unknown"
  vault: "Sea floor"
  uncertainties.m: {lon: 200, lat: 200, elev: 20}
  localisation_method: "Sea surface release point"

```

Class Navigation

Location <==

INSTALLATION AND STARTUP GUIDE

3.1 Prerequisites

obsinfo is written in Python 3.6.9. While it is expected that it will run in any version of Python 3, this has not been tested.

You must have Python installed in your computer with at least version 3.6.9.

3.1.1 Linux

Most Linux installations have Python preinstalled. However, it might not be the latest version. Check the version and update to 3.6.9 at least.

To check your version, use any of the following commands:

```
$ python --version
$ python2 --version
$ python3 --version
```

3.1.2 Windows

Must have Windows 10 installed.

3.2 Python installation

The following link provides complete information on how to install Python in several platforms:

<https://realpython.com/installing-python/#how-to-check-your-python-version-on-linux>

The official installation guide for **UNIX** is here:

<https://docs.python.org/3/using/unix.html>

Windows:

<https://docs.python.org/3/using/windows.html>

MacOS:

<https://docs.python.org/3/using/mac.html>

3.3 Python packages

The following packages must be installed, preferably with a package manager, either using `pip` or `conda`.

Instructions to install a package using `pip`. If you don't have `pip` installed, this same link instructs you who to install it.

<https://packaging.python.org/tutorials/installing-packages/>

Instructions to install a package using `conda`

<https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-pkgs.html>

Packages

Standard library (no need to install)

```
difflib
glob
inspect
json
math
pathlib
pprint
os
re
sys
unittest
warnings
```

External libraries

```
jsonschema
jsonref
gitlab
numpy
obspy
pyyaml
```

In particular *obsinfo* relies on *obspy* to create the objects needed to output a StationXML file.

3.4 Git installation

If you wish to install *obsinfo* using `git`, you have to install it first.

Linux:

```
$ sudo apt-get install git
```

In **Windows**, download from this site:

```
https://git-scm.com/download/win
```

3.5 Obsinfo installation

obsinfo is a published package in PyPI. Currently, version v0.110 is available as a beta version.

Make sure you have the latest pip version: .. code-block:: console

```
$ python3 -m pip install --upgrade pip
```

or

```
$ python -m pip install --upgrade pip
```

depending on your Python configuration.

Next, use pip to install *obsinfo*:

```
$ python3 -m pip install obsinfo
```

or

```
$ python -m pip install obsinfo
```

depending on your Python configuration.

3.6 Description of obsinfo file structure

data	YAML/JSON schemas
_examples	Example information files
instrumentation	Python code
main	Python code
misc	Python code
network	Python code
obsMetadata	Python code

3.7 Obsinfo setup

Linux:

The *obsinfo* executables are:

- `obsinfo-makeStationXML` to create StationXML files
- `obsinfo-validate` to validate the syntax of information files
- `obsinfo-print` to simply print information files without creating the StationXML file. This may be useful as a test.

All executables are installed in `~/.local/bin`, which is usually in the PATH variable of the operating system. If this is not the case, add that directory:

```
$ PATH=$PATH:$HOME/.local/bin
$ export PATH
```

To avoid having to set up the variable in each session, store these instructions in your `.bashrc` file in your home directory.

Remember in **Windows** and **MacOS** you can setup the variable in the configuration panel. Follow the instructions to set environment variables.

After having installed *obsinfo* with pip or conda, you need to run a setup. This gives you an opportunity of creating a local directory where the existing examples of information files will be copied, instead of the arcane and standard `~/.local/lib/python3.x/site-packages/obsinfo/_examples/Information_Files`.

In its simplest form, simply run:

```
$ obsinfo-setup -d <directory name>
```

The program will copy the examples to the directory mentioned and will perform other administrative tasks.

IMPORTANT: EVEN if you don't want to create a new directory, you **MUST** run `obsinfo-setup`, as it performs several housekeeping tasks, notably creating the configuration file `.obsinfo` and the directory `.obsinfo` which is used for logging purposes.

`obsinfo-setup` is very flexible. Through it you can reconfigure several variables that will tell *obsinfo* where to find your files. More on this in the next section.

3.8 File discovery

`obsinfo-setup` sets several variables specific to *obsinfo* and stores them in a configuration file called `.obsinfo` in your home directory called `.obsinfo`. Here they are with their default values:

```
gitlab_repository: www.gitlab.com
gitlab_project: resif/obsinfo
gitlab_path: _examples/Information_Files
datapath: [https://www.gitlab.com/resif/obsinfo/obsinfo/_examples/Information_Files]
obsinfo_branch: master
```

The first, second and third variables are used to tell *obsinfo* where the remote repository for information files is. The fourth one needs more explanation. It works like the `PATH` variable in Linux: whenever a file in a `$ref` is found in an information file, *obsinfo* will **sequentially** look for the file in all the directories specified in `obsinfo_datapath`. In the default case, the only place where *obsinfo* will look is the remote directory. Observe this is the concatenation of the first three variables, the repository, the project and the path.

Finally, since the remote repository is assumed to be a gitlab repository, we must specify a gitlab branch, which by default is `master`.

All these defaults are achieved by simply running:

```
$ obsinfo-setup
```

with no parameters. But with options you can specify all the variables. `gitlab_repository` is specified with `-g`, `gitlab_project` with `-p`, `gitlab_path` with `-P` and `obsinfo_branch` with `-b`.

Other options allow you to not copy the examples (`-x`) or modify the `datapath` variable. This is done so you don't have to manually edit `.obsinfo` **which is discouraged**. Three local directories are considered in `datapath`: a working directory, a local repository and an example directory. We have seen that the `-d` option creates (if non existing) the example directory. When specified, this directory takes precedence over the remote directory (unless `-x` is specified). The `datapath` variable will look like:

```
datapath: [<local example directory>, https://www.gitlab.com/resif/obsinfo/obsinfo/_
↳ examples/Information_Files]
```

where <local example directory> is the directory specified with -d. If you add the working directory (with -w) and the local repository (with -l), they will take precedence in this order:

```
datapath: [<working directory>, <local repository>, <local example directory>, https://
↳www.gitlab.com/resif/obsinfo/obsinfo/_examples/Information_Files]
```

You can change the order putting the remote repository first by using the -v option. Of course, you change the order of the local directories by simply specifying one or the other as working, local repository or example.

All this is summarized here:

```
-x, --no_examples      Don't import examples, only templates, and remove
                        examples directory from the datapath
-c, --no_copy          Don't import anything at all, don't create dest
                        directory, which will be removed from datapath
-n, --no_remote        Install obsinfo without access to a gitlab repository.
                        May be needed in some operating systems for
                        compatibility issues
-v, --invert_datapath  Put remote gitlab repository first. All local
                        directories will keep their order
-b, --branch           Specifies the git branch to use, if not master
-d DEST, --dest DEST  Destination directory for templates and examples.
-g GITLAB, --gitlab GITLAB
                        Gitlab repository)
-p PROJECT, --project PROJECT
                        path to project and the directory where information
                        files lie within the Gitlab repository)
-l LOCAL_REPOSITORY, --local_repository LOCAL_REPOSITORY
                        Specify local repository for information files and
                        include it as first or second option in datapath
-w WORKING_DIRECTORY, --working_directory WORKING_DIRECTORY
                        Specify working directory for obsinfo and include it
                        as first option in datapath
-P REMOTE_PATH, --remote_path REMOTE_PATH
                        Specify remote directory under project
```

IMPORTANT: In datapath, the remote directory is always the concatenation of gitlab_repository, gitlab_project and gitlab_path when created by obsinfo-setup. Therefore, if you manually change the datapath in the .obsinfo file you will not get the expected result. Always use obsinfo-setup to change that file.

Of course, you will create your own information files in a directory selected by you.

EXECUTING OBSINFO

There are four main executables:

- `obsinfo-setup`
- `obsinfo-makeStationXML`
- `obsinfo-validate`
- `obsinfo-print`

4.1 `obsinfo-setup`

creates an *.obsinfo* file that indicates where `obsinfo` should look for the reference files. More details in the Installation and Startup Guide

4.2 `obsinfo-makeStationXML`

Now you're all set to run `obsinfo`. Type

```
$ obsinfo-makeStationXML -h
```

to display all the options of `makeStationXML`. Most are self-explanatory.

To create a StationXML file from a file called `<filename>`, type:

```
$ obsinfo-makeStationXML [options] filename
```

The output file, by default, will have the stem part of the network filename followed by “station.xml”. That is, if the filename to be processed is `BBOBS.INSU-IPGP.network.yaml`, the resulting file will be called by default `BBOBS.INSU-IPGP.station.xml`. This can be changed with the `-o` option.

A single file is processed at a time. This is basically to simplify operation and avoid confusion with file discovery. However, a `for` statement in a shell script can be used to process several files at a time, as is customary.

The most important thing about the way *obsinfo-makeStationXML* operates is where it finds its information files. As a rule, the argument you pass to the application must have a non-ambiguous path to a network information file, such as:

..`code-block`:

```
BBOBS.INSU-IPGP.network.yaml

./BBOBS.INSU-IPGP.network.yaml

/home/arean/examples/BBOBS.INSU-IPGP.network.yaml

../../my_examples/BBOBS.INSU-IPGP.network.yaml
```

Standard POSIX notation is used. The first and second examples will look for the file in the current working directory. The third one is called an absolute path and will try to locate the file in the `/home/arean/examples/` directory. The fourth one is a path relative to the working directory which will go up to levels and then down to `/my_examples` to find the file.

All other files (i.e. the files in `$ref` attributes, will operate in a different fashion. Examples two, three and four will work in the same way, but example one will be considered a file that needs discovery. This discovery is performed through the variable `obsinfo_datapath`, which is a list of directories separated by commas, in typical Python/YAML syntax. Every director is visited in sequence. When the file is found in one of the, the discovery stops. If not file is found in any of the directories, an error message is generated.

We can also make the filename passed as argument to `obsinfo-makeStationXML` behave in this way with the option `-r`. If we use this option, even the network file passed as an argument will be discovered in one of the directories in `OBSINFO_DATAPATH`.

The rest of the options are self-explanatory, and are explained by a message generated with:

```
$ obsinfo-makeStationXML -h

usage: obsinfo-makeStationXML [-h] [-r] [-l] [-v] [-q] [-d] [-t] [-V]
                               [-o OUTPUT]
                               input_filename

positional arguments:
  input_filename          is required and must be a single value

optional arguments:
  -h, --help              show this help message and exit
  -r, --remote             Assumes input filename is discovered through obsinfo_datapath.
                           Does not affect treatment of $ref in info files
  -l, --validate          Performs complete validation, equivalent to obsinfo-validate,
↳ before processing
  -v, --verbose            Prints processing progression
  -q, --quiet             Silences a human-readable summary of processed information file
  -d, --debug             Turns on exception traceback
  -t, --test              Produces no output
  -V, --version            Print the version and exit
  -S, --station           Create a StationXML file with no instrumentation
  -o OUTPUT, --output OUTPUT
                           Names the output file. Default is <input-filename-stem>.station.
↳ xml
```


4.3 obsinfo-validate

This executable will validate the specified file:

```
$ obsinfo-validate [options] filename
```

obsinfo-validate will identify the type of file from the filename and run the relevant validation.

Option *-r* works as in *obsinfo-makeStationXML*. The rest of the options are self-explanatory:

```
$ obsinfo-validate -h

usage: obsinfo-validate [-h] [-q] [-r] [-d] input_filename

positional arguments:
  input_filename  Information file to be validated.

optional arguments:
  -h, --help      show this help message and exit
  -q, --quiet      Quiet operation. Don't print informative messages
  -r, --remote     Search input_filename in the DATAPATH repositories
  -d, --debug      Print traceback for exceptions
```

As mentioned in Best Practices, it is always a good idea to validate files before trying to create a StationXML file. Use a bottom-up approach to avoid getting difficult-to-read error messages: start with filters, then stages, then components, then instrumentations, then networks.

All files in official central repositories are assumed to have been validated.

4.4 obsinfo-print

This executable will print the *obsinfo* objects generated out of the specified file:

```
$ obsinfo-print [options] filename
```

obsinfo-print will identify the type of file from the filename and run the relevant printing routine. When used with the *-l* option, it will print up to a certain level specified after the option according to the keywords below. For example, a sensor file with a stage level will not print the filter information, and a network file with a station level will not print the instrumentation information and down.

```
usage: obsinfo-print [-d] [-h] [-l LEVEL] input_filename

positional arguments:
  input_filename  Information file to be validated.

optional arguments:
  -l or --level: prints up to LEVEL, where LEVEL is:
                  all
                  stage
                  component
                  instrumentation
                  channel
                  station
```

(continues on next page)

(continued from previous page)

```
network
-d or --debug: do not catch exceptions to show error trace.
-h or --help: prints this message
```

TUTORIAL

5.1 Introduction

Information files are written in YAML or JSON, mostly YAML. YAML is a markup language (despite some claim its acronym means “yaml ain’t a markup language”) that permits users to encode data in a structured format which can be shared, written and read by many applications using text files (rather than binary ones), a process known as *serialization* of data. It is one of the standard tools for this purpose, others being JSON and XML, which tend to be more verbose and harder to read and write. This tutorial will center on YAML, but it is readily translatable to JSON for users fluent in the use of that markup language, which can, at any rate, be learned [here](#). Keep in mind YAML is a superset of JSON, so some functionality is not readily implemented in the latter.

This is **not** a YAML tutorial. We freely mix required YAML syntax with best practices advocated for *obscure* and leave out many aspects of the language. For people wanting to get acquainted with YAML, a number of resources are available, such as [this](#).

5.1.1 Basic YAML syntax

YAML files, as in the other markup languages mentioned, are structured hierarchically. The basic structure is the key-value pair, which permits to assign a value retrievable by key (and easily readable by a human user), such as:

```
last_name : "Presley"  
first_name : "Elvis"
```

Being hierarchical, these key-value pairs can be nested:

```
artist_name:  
  last_name : "Presley"  
  first_name : "Elvis"
```

Space indentation is used in YAML to nest key-value pairs. NEVER use tabs. As a convention, two spaces are used and all key-value pairs at the same level must be equally indented.

YAML uses three dashes “—” to separate different streams of data. Always put as the first line in your file the three dashes as a best practice. All YAML files such have a `.yaml` extension.

5.1.2 YAML data types

There are other observations for the little piece of code above. First, the data types. Scalars can be either a **number**, a **boolean** (True or False as values) or a **string**, enclosed in double quotes. Numbers can be integers (without decimal point) or floating point (with decimal point).

Other data structures include lists and dictionaries. A list is simply an enumeration of elements which can be any data type, enclosed in brackets or listed in separate indented lines which start by dashes. The following are equivalent:

```
die_toss: [1,2,3,4,5,6]
```

```
die_toss:
```

- 1
- 2
- 3
- 4
- 5
- 6

Dictionaries are a collection of key-value pairs. They can either be indented, as above (`last_name: "Presley"` and `first_name: "Elvis"` are actually elements of a dictionary value associated with the key `artist_name`) or as enumerations enclosed in curly brackets, or, again, as dashes. As a side note, the curly brackets syntax makes YAML compatible with JSON. The following are equivalent:

```
artist_name:
```

```
  last_name : "Presley"  
  first_name : "Elvis"
```

```
artist_name: { last_name : "Presley", first_name : "Elvis" }
```

```
artist_name:
```

- last_name : "Presley"
- first_name : "Elvis"

5.1.3 YAML variables

YAML is case sensitive. In *obsinfo* we only use keys in lower case with words separated by underscores.

5.1.4 Code reuse

The `$ref` special variable, a JSON feature, is used in *obsinfo* to allow the inclusion of the content of another file in the current YAML file:

```
revision:
```

```
  date: "2018-06-01"
```

```
  authors:
```

- \$ref: "Wayne_Crawford.author.yaml#author"

In this example, only the part corresponding to the key `author` will be included. Note that a file can be totally included by omitting the `#author` anchor. `$ref` references will totally override all other keys at their level. For example, if we had another field at the `authors` level:

```
revision:
  date: "2018-06-01"
  authors:
    - $ref: "Alfred_Wegener.author.yaml#author"
    email: Alfred_Wegenerd@yahoo.de
```

the email field will disappear in the final result. Contrast this with YAML anchors, to be discussed next.

5.1.5 YAML anchors

YAML anchors are used to avoid repetition, according to the DRY (“don’t repeat yourself”) principle. In this real *obsinfo* example, an anchor is defined which has the value of a dictionary:

```
yaml_anchors:
  obs_clock_correction_linear_defaults: &LINEAR_CLOCK_DEFAULTS
    time_base: "Seascan MCX0, ~1e-8 nominal drift"
    reference: "GPS"
    start_sync_instrument: 0
```

Further down the information file the following appears in several places with different values for the `start_sync_reference`, `end_sync_reference` and `end_sync_instrument` keys:

```
processing:
  - clock_correction_linear_drift:
    <<: *LINEAR_CLOCK_DEFAULTS
    start_sync_reference: "2015-04-21T21:06:00Z"
    end_sync_reference: "2016-05-28T20:59:00.32Z"
    end_sync_instrument: "2016-05-28T20:59:03Z"
```

When an anchor is referenced with a star (*) it’s called an alias and has the effect of replacing the alias by the anchor definition. The effect will be:

```
processing:
  - clock_correction_linear_drift:
    time_base: "Seascan MCX0, ~1e-8 nominal drift"
    reference: "GPS"
    start_sync_instrument: 0
    start_sync_reference: "2015-04-21T21:06:00Z"
    end_sync_reference: "2016-05-28T20:59:00.32Z"
    end_sync_instrument: "2016-05-28T20:59:03Z"
```

Furthermore, the `<<` label above indicates that this is a mapping. If no fields with the similar name appear under the alias, its effect is to simply replace the alias by the anchor, as mentioned earlier. But if there are fields such as `time_base` under the alias, like this:

```
- clock_correction_linear_drift:
  <<: *LINEAR_CLOCK_DEFAULTS
  time_base: "unavailable"
```

those fields will be overridden, so the effect of the latter piece of code is the following:

```
- clock_correction_linear_drift:
  time_base: "Seascan MCX0, ~1e-8 nominal drift"
```

(continues on next page)

(continued from previous page)

```
reference: "GPS"
start_sync_instrument: 0
```

overriding the value of `time_base`.

All this allows for code reuse without needing an external file as in `$ref`.

Next page, Information File Structure, discusses how to start creating *obsinfo* information files.

5.2 The general structure of information files

Information files are arranged in a hierarchy, with upper files referencing lower files. There are 5 main file types in *obsinfo*:

Name	Description	Filled by	When filled
network	Deployed stations, their instruments and parameters	OBS facility	after a campaign
instrumentation	Instrument description	OBS facility	new/changed instruments
<i>instrument_components</i> sensors components dataloggers	Description of basic components	OBS facility -or- component manufacturer	when there are new are new components or calibrations
stage	Description of stages (digitizers, amplifiers/filters)	OBS facility -or- component manufacturer	when there are new components or calibrations
filter	Description of filters (amplifiers and digitizers used in stages)	OBS facility -or- component manufacturer	when there are new components or calibrations

The hierarchy is a best practice. In principle, all information contained in lower level files can be included in the network file. However, it is highly recommended that the “`$ref`” functionality to refer to other files is used.

5.2.1 File Hierarchy

It is recommended to arrange the information files in a file hierarchy such as this:

..code-block:

```
/authors (contains files used in headers of other information files)
/network
/instrumentations
/sensors
/preamplifiers
/dataloggers
[instrumentation_component]/stages/
[instrumentation_component]/stages/filters (filters can be substituted by a specific
↪ kind of filter: FIR, PZ, etc.)
```

where [instrumentation_component] = sensors, preamplifiers or dataloggers.

Another possibility is to arrange instrument components under instrumentation, but that may cause extra typing while writing references (\$ref) in information files. That would be something like this:

..code-block:: /instrumentations/sensors

The hierarchy is completely up to the user, including the names of the folders/directories. In fact, it is perfectly possible to put all the information in a single file, although it is not recommended as reusability of filters, stages or components depends on independent files.

To reference a file from within another file, use the *JREF* syntax:

```
authors: - {$ref: "authors/Wayne_Crawford.author.yaml#author"}
```

The effect of this directive is at the core of the philosophy of *obsinfo*, as it is this mechanism which allows reuse: it substitutes the content of the key `author` within the file `authors/Wayne_Crawford.author.yaml` as the value of the key `authors`. If you want to include the complete file, remove the `#author` anchor.

How to express absolute and relative paths, and their meaning, are discussed later.

5.2.2 File Naming Convention

While there is flexibility about the folder hierarchy, information files **must** follow the following naming convention:

<descriptive file name>.<obsinfo file type>.<file format>

where

<descriptive file name> usually includes vendor and configuration shorthand to make the file easily identifiable by users,

<obsinfo file type> is one of campaign, network, instrumentation, sensor, preamplifier, datalogger, stage, filter

<file format> is one of yaml or json.

Examples:

- `TI_ADS1281_FIR1.stage.yaml` is a stage with a Texas Instruments FIR filter nested in it, in YAML format.
- `BBOBS.INSU-IPGP.network.json` is a network of broad-band stations deployed by INSU IPGP, in JSON format.

There are two resources, other than this tutorial, to look up the exact syntax for each information file. The different pages have a complete explanation of all the attributes in the class. Unless otherwise noted, YAML keys or labels (also called fields or sections) in the file have the same exact name (case sensitive) as the attributes in the class. The other resource is the formal syntax of the file, which is a JSON schema, which is always referenced in the Class page.

5.2.3 File Metadata

All information files contain or may contain common metadata. `format_version` and `revision` are required.

- **format_version:** - This is a required field. It reflects the template version against which this file must be validated
- **revision:** - Revision information (date in particular) to keep change control of file.
 - **date:** - date of revision
 - **authors:** - authors of revision, usually a reference to an author file

- **notes:** - Optional extra information which will not be put in final metadata.
- **yaml_anchors:** - YAML anchors to avoid redundancy, for advanced YAML users. [Here](#) is a guide on how to use YAML anchors.
- *Next page: Building a simple network file*
- *Previous page*
- *Back to start*

5.3 Building a simple network file with stations

5.3.1 Fundamentals of an information file

Under the folder which will contain all your information files, create a folder called **network**. Network information files can be part of or referenced by a campaign information file, but since we are not dealing with campaigns in *obsinfo* we will start with a network file.

Use your favourite text editor to create a network information file. Select as filename something meaningful, for example, single period OBS from INSU-IPGP would be SPOBS.INSU-IPGP. Then add the type of information file, **network** and the type of format, **yaml**:

```
SPOBS.INSU-IPGP.network.yaml
```

The file should start with the three dashes; the next line must specify the *obsinfo* version (required to know how to process the file). There are several optional fields which we will omit in this first exercise. It's a good idea, though, to include a revision with a date and an author.

```
---
format_version: "0.110"
revision:
  authors:
    -
      first_name: "Wayne"
      last_name: "Crawford"
      email: "crawford@ipgp.fr"
  date: "2017-10-04"
```

Note the dash in a lonely line which indicates **authors** is a list. Alternatively, since the author is probably an information that will be repeated several times, this information can be stored in another file. Let's say we create a folder named **authors** at the same level as **network** and then create a file inside that folder named **Wayne_Crawford.author.yaml**. We put the three fields previously under the **authors** key in that file, so the file will look like:

```
---
format_version: "0.110"
author:
  first_name: "Wayne"
  last_name: "Crawford"
  email: "crawford@ipgp.fr"
```

And then we reference the file with a **\$ref** field in the original network file:


```

---
format_version: "0.110"
revision:
  authors:
    - $ref: 'authors/Wayne_Crawford.author.yaml'
  date: "2017-10-04"

```

The effect of this, from the point of view of *obsinfo*, is to insert said part of the `Wayne_Crawford.author.yaml` file instead of the `$ref` line. Now, observe that if you do this the *totality* of the contents will be inserted. This is undesirable as it will cause a syntax error; the system expects to see the fields `first_name`, `last_name` and `email`, not the three dashes, the version, etc. The solution to this is using an *anchor* or *fragment*, which will only insert the contents of the field referenced by the anchor. In this case, the anchor should be the field `author`, so the final syntax is the following:

```

---
format_version: "0.110"
revision:
  authors:
    - $ref: 'authors/Wayne_Crawford.author.yaml#author'
  date: "2017-10-04"

```

This will be regular practice to avoid repeating information. In fact, it is the heart of the *obsinfo* information file hierarchy.

File discovery

How to find the information files is one of the most relevant features in *obsinfo*. Notice the pathnames in the examples above are not absolute (i.e. they don't give an absolute reference such as `/home/luis/obsinfo/authors/Wayne_Crawford.author.yaml#author`). In regular POSIX practice it would be assumed that not including the parent directory means that file lies in the directory where the application is executed out of (called the current working directory or `cwd`). This is not the case with *obsinfo*. What the application does is try to *discover* the file in one of several directories which are specified by the variable `obsinfo_datapath`, set by the `obsinfo-setup` application and found in the configuration file `.obsinfo.rc`, which is in turn found in your home directory. This works much in the same way executables are found in Linux, MacOS or Windows with the variable `PATH`.

Whenever a file in a `$ref` is found without an absolute or relative path, *obsinfo* will **sequentially** look for the file in all the directories specified as a Python list in `obsinfo_datapath`. A special keyword, `GITLAB`, is used to specify a remote repository in GitLab. Let's see an example:

`obsinfo_datapath`, as specified above, will always look in a local directory where the current examples are installed (via `pip` or `conda`) and then, if not found, in the remote repository. This gives the user extreme flexibility, as (s)he can override an existing remote information file with a local one, change the order of discovery, etc.

Of course, you will create your own information files in a directory selected by you. Then you will have to edit the `obsinfo_datapath` variable to reflect the directory you want to put your information files in.

On the other hand, it is possible, although not recommended, to use absolute and relative paths.

5.3.2 Network

The next field key starts with the actual information. It's **network**. You may specify several attributes which are listed in the *Network*, but let's stick to the absolute fundamentals:

```
network:
  facility:
    reference_name: "INSU-IPGP"
    full_name: "INSU-IPGP OBS Park"

    campaign_ref_name: "SPOBS"

  network_info:
    code: "4G" #This is either an FDSN provided network code or "XX" if no such code_
    ↪exists.
    name: "Short period OBSs" #Use the FDSN name if applicable
    start_date: "2007-07-01"
    end_date: "2025-12-31"
    description: "Short period OBS network example"
```

This section describes the network itself in **network_info**. It also adds a reference to the campaign name and the description of the facility (the institution responsible), which has in turn a reference (short) name and a long name.

If the network has been declared to the FDSN, the information in **network_info** should correspond to the values on the FDSN site. For information on how to request a network code or use a temporary code, see this [link](#).

Stations

Stations belonging to the network are specified next. This could of course belong to a different file in a **stations** folder, but it is a best practice to leave the station information here to avoid excessive clog.

The following attribute is, therefore, **stations**. We can specify several in the same format, but in order to keep the example simple we will only specify one. Stations are identified by a one to five character code, which can be an acronym of the longer description, although the content of the field is completely arbitrary. This code acts like a key, but be careful to surround it by quotes as it is not a part of the syntax. Stations have an optional start date and end date, which may differ from the network's, as they may be deployed at different times.

The site is described as a text field, and a location code is specified too. More on locations later.

```
stations:
  "LSVW":
    site: "Lucky Strike Volcano West"
    start_date: "2015-04-22T12:00:00Z"
    end_date: "2016-05-28T21:01:00Z"
    location_code: "00"
    instrumentation:

    ...
```

5.3.3 Instrumentation

Most importantly, stations must have an instrumentation, which is where the actual devices which provide source data, signal processing and data logging are specified. The best practice is to specify these in a separate file, but they must be referenced here to make the link. The way to reference an instrumentation is the following:

```
instrumentation:
  $ref: "instrumentation/SPOBS2.instrumentation.yaml#instrumentation"
```

\$ref a different file in a folder called `instrumentation`. This is, again, best practice, as it will allow the reuse of instrumentations in several stations. It is possible to specify several instrumentations in list format:

```
instrumentations:
- $ref: "instrumentation/SPOBS2.instrumentation.yaml#instrumentation"
- $ref: "instrumentation/BBOBS1_2012+.instrumentation.yaml#instrumentation"
```

However, this has little use for OBS equipment as there is usually a single instrumentation per station.

5.3.4 Locations

Now let's go back to locations. There must be at least one location in the file. This is the position in geographical coordinates of the station, and is usually referred as location "00", which is the `location_code` of the station. Locations are specified as follows:

```
locations:
  "00":
    base: {$ref: 'location_bases/SURFACE_DROP.location_base.yaml#location_base'}
    position: {lon: -32.32504, lat: 37.29744, elev: -2030}
```

There is a base, which is again a referenced file in a different folder (as best practice). The syntax shown here will be common to all the file references in *obsinfo* information files. The folder here is not at the same level of the `network` folder, but rather under it. That's the meaning of the way the file is referenced as a *pathname*, which is, on the other hand, regular POSIX syntax. Use of slashes (/) instead of Windows backslashes (\) are recommended for uniformity, so a file can be migrated among different operation systems. It should be noted Mac uses POSIX syntax. However, if you use backslashes, *obsinfo* will understand them.

Observe the difference between a list and a dictionary. List items are separated by dashes, dictionaries need a key/value pair. Authors is a list. Locations is a dictionary.

However, there can be several locations. That's the reason we have a `location_code` to specify the location corresponding to the station itself. The rest of locations are used for channels, as will be seen shortly. Those channels may have different geographical coordinates, but, more importantly, this is the way in which we can distinguish channels among themselves.

5.3.5 Channel modifications

Next to instrumentation there is the possibility of making modifications to channels in the instrumentation. For example, a change in the configuration, which will select one of several possible configurations for the instrument components of a channel (more about this in the *Next page, Building a simple instrumentation file*). If both are specified, the one in the network file takes precedence.

Channel modifications are what makes *obsinfo* so flexible. We can specify several different configurations for the same components, and then select one of them. We can also directly change other attributes in the instrumentation. This allows components to be regarded almost as virtual descriptions which, when a particular configuration is selected, are

instantiated into a specific, actual component. Thus instrumentation files can be authentic libraries with little changes, while the changes of configuration are specified for each station in a network in a specific campaign.

Furthermore, these libraries can reside in a central GitLab repository, which is updated by authorized users and, being public, is available for reuse by all users. A user can even clone the repository in the regular GitLab way (see [here](#)) in order to work offline with the latest version of the repository.

5.3.6 Complete example

You can skip this section in a first reading.

This is an actual network information file with the above information and some optional fields which have not been described. Note the additions:

- 1) Addition of a second station
- 2) Use of `yaml_anchors` to avoid repeating information *in the same file*
- 3) Comments field in `network_info`
- 4) Processing field in `stations`. For more information on this, see *Processing*

```
---
format_version: "0.110"

yaml_anchors:
  obs_clock_correction_linear_defaults: &LINEAR_CLOCK_DEFAULTS
    time_base: "Seascan MCX0, ~1e-8 nominal drift"
    reference: "GPS"
    start_sync_instrument: 0
revision:
  authors:
    - $ref: 'authors/Wayne_Crawford.author.yaml#author'
  date: "2017-10-04"
network:
  facility:
    reference_name: "INSU-IPGP"
    full_name: "INSU-IPGP OBS Park"
  campaign_ref_name: "EMSO-MOMAR_I"
  network_info:
    code: "4G" #This is an FDSN provided network code.
    name: "Short period OBSs"
    start_date: "2007-07-01"
    end_date: "2025-12-31"
    description: "Short period OBS network example"
    comments: ["Lucky Strike Volcano, North Mid-Atlantic Ridge"]
  stations:
    "LSVWI":
      site: "Lucky Strike Volcano West"
      start_date: "2015-04-22T12:00:00Z"
      end_date: "2016-05-28T21:01:00Z"
      location_code: "00"
      instrumentation:
        $ref: "instrumentation/SPOBS2.instrumentation.yaml#instrumentation"
      channel_modifications:
```

(continues on next page)

(continued from previous page)

```

    "*_*": {datalogger_configuration: "125sps"}
  locations:
    "00":
      base: {$ref: 'location_bases/SURFACE_DROP.location_base.yaml#location_
↪base'}
      position: {lon: -32.32504, lat: 37.29744, elev: -2030}
  processing:
    - clock_correction_linear_drift:
      <<: *LINEAR_CLOCK_DEFAULTS
      start_sync_reference: "2015-04-21T21:06:00Z"
      end_sync_reference: "2016-05-28T20:59:00.32Z"
      end_sync_instrument: "2016-05-28T20:59:03Z"
  "LSVEI":
    site: "Lucky Strike Volcano East"
    start_date: "2015-04-22T12:00:00Z"
    end_date: "2016-05-28T21:01:00Z"
    location_code: "00"
    instrumentation:
      $ref: "instrumentation/SPOBS2.instrumentation.yaml#instrumentation"
    channel_modifications:
      "*_*": {datalogger_configuration: "125sps"}
    locations:
      "00":
        base: {$ref: 'location_bases/ACOUSTIC_SURVEY.location_base.yaml
↪#location_base'}
        position: {lon: -32.02504, lat: 37.25744, elev: -2130}
    processing:
      - clock_correct_linear_drift:
        <<: *LINEAR_CLOCK_DEFAULTS
        start_sync_reference: "2015-04-21T21:06:00Z"
        end_sync_reference: "2016-05-28T20:59:00.32Z"
        end_sync_instrument: "2016-05-28T20:59:01Z"

```

In all *obsinfo* information files, you can always add notes as a list of strings. Notes are not reflected in the output StationXML file. They only serve documentation purposes within the information file. On the other hand, comments can only be added in selected places which correspond to the places in a StationXML file which admit comments. They will be copied into the StationXML output.

You can also add extras, which are key:value pairs that you wish to document but are not taken into account by the *obsinfo* syntax. When possible, they will be added as comments to the output StationXML file.

- *Next page, Building a simple instrumentation file*
- *Previous page*
- *Back to start*

5.4 Building a simple instrumentation file with channels

As seen in the last section, instrumentation are usually referred to with `$ref` from a network / station information file. This is a best practice, but it is not mandatory. It does allow for easier reuse.

The file starts as usual:

```
---
format_version: "0.110"
revision:
  date: "2019-12-19"
  authors:
    - {$ref: "authors/Wayne_Crawford.author.yaml#author"}
    - {$ref: "authors/Romuald_Daniel.author.yaml#author"}
```

Observe that we have added an author to the list of authors, and that lists are separated by dashes.

5.4.1 Equipment

The main part of the file is the instrumentation section. First, we have the `equipment` section, which details the manufacturer, model and serial number of the whole instrumentation, and which can be overridden for individual components.

```
instrumentation:

  equipment:
    model: "BBOBS1"
    type: "Broadband Ocean Bottom Seismometer"
    description: "LCHEAPO 2000 BBOBS 2012-present"
    manufacturer: "Scripps Inst. Oceanography - INSU"
    vendor: "Scripps Inst. Oceanography - UNSU"
```

As most OBS are assembled with parts from different manufacturers, the only required fields of the equipment section are the type (a free-form text field) and the description.

5.4.2 Channels and channel default

In second place, we have channels. Channels are the most relevant part of the instrumentation. A channel is the ensemble of the three instrument components (called an instrument): sensor, optionally a preamplifier and a datalogger, plus an orientation. Orientation codes are explained [here](#) in the *Geographic orientation subsurface codes* section. They are dictated by FDSN standards.

The `channels` are the actual channels in the instrumentation. They all have string labels, which are usually channel numbers. These must be in quotes as they are not keys in the *obsinfo* syntax. They must specify an orientation.

However, to avoid duplication, a wildcard, default channel identified with label `default` is used to declare common elements to all channels. This is not an actual channel. It's just a place to hold default attributes. If an attribute is not specified in an actual channel but exists in the default channel then it will be added to the final configuration of the channel.

Let's see an example:

```

channels:
  default:
    sensor: {$ref: "sensors/NANOMETRICS_T240_SINGLESIDED.sensor.yaml#sensor"}
    preamplifier:
      {$ref: "preamplifiers/LCHEAPO_BBOBS.preamplifier.yaml#preamplifier"}
    datalogger: {$ref: "dataloggers/LC2000.datalogger.yaml#datalogger"}

    "1": {orientation_code: "2": {azimuth.deg: [90, 0], dip.deg: [0, 0]}} +
    "2":
      orientation_code:
        "1":
          azimuth.deg: [0, 9]
          dip.deg: [0, 0]

```

This code specifies two channels as a dictionary. Each channel specifies the two or three instrument components and the `orientation_code`. The `orientation_code` will become the third character in the SEED code identification (see *SeedCodes*, and thus must follow FDSN standards.

Again, these are real, physical channels. But the default specifies three instrument components: `sensor`, `preamplifier` and `datalogger`. The `preamplifier` is optional. All three files reference an information file in separate directory, which, in the example, are just under the directory `Information_Files`. All fields in the default are applied to each one of the channels *unless* the channel already has one of those components. In other words, fields in the individual channels override fields in the default. So this is the same as typing:

```

channels:
  "1": {
    sensor: {$ref: "sensors/NANOMETRICS_T240_SINGLESIDED.sensor.yaml#sensor"}
    preamplifier:
      {$ref: "preamplifiers/LCHEAPO_BBOBS.preamplifier.yaml#preamplifier"}
    datalogger: {$ref: "dataloggers/LC2000.datalogger.yaml#datalogger"}

    orientation_code: {"2": {azimuth.deg: [9, 0], dip.deg: [0, 0]}}
  }
  "2": {
    sensor: {$ref: "sensors/NANOMETRICS_T240_SINGLESIDED.sensor.yaml#sensor"}
    preamplifier:
      {$ref: "preamplifiers/LCHEAPO_BBOBS.preamplifier.yaml#preamplifier"}
    datalogger: {$ref: "dataloggers/LC2000.datalogger.yaml#datalogger"}

    orientation_code:
      "1":
        azimuth.deg: [90, 0]
        dip.deg: [0, 0]
  }
}

```

Observe that in this case we have used curly parentheses to specify elements in a dictionary. They can be left out, letting simple indentation do the job of determining the items in the dictionary.

Now, if a channel specifies any of the fields in the default, it will override that field. Let's assume we have a channel 4 with a different sensor and preamplifier:

```

"4":
  orientation_code : {"H": {{azimuth.deg: [0, 0], dip.deg: [90, 0]}}
  preamplifier: {$ref: "preamplifiers/LCHEAPO_DPG.preamplifier.yaml#preamplifier"}

```

(continues on next page)

(continued from previous page)

```
sensor: {$ref: "sensors/SIO_DPG.sensor.yaml#sensor"}
```

Then, assuming the same default field as above, the result of channel 4 would be:

```
"4":
  orientation_code : {"H": {azimuth.deg: [0, 0], dip.deg: [90, 0]}}
  preamplifier: {$ref: "preamplifiers/LCHEAPO_DPG.preamplifier.yaml#preamplifier"}
  sensor: {$ref: "sensors/SIO_DPG.sensor.yaml#sensor"}
  datalogger: {$ref: "dataloggers/LC2000.datalogger.yaml#datalogger"}
```

Only datalogger maintains the default value. The other components are overridden with the values specified in channel 4.

5.4.3 Orientation Codes

Now we need to explain the orientation codes. Orientation codes are a FDSN standard. By convention, if the orientation code is **X**, **Y** or **Z**, these represent the regular coordinates in space following the right-hand rule, within five degrees of the actual directions. So **X** corresponds to an azimuth of 0° and a dip of 0°, **Y** corresponds to an azimuth of 90° and a dip of 0°, and **Z** corresponds to an azimuth of 0° and a dip of 90° (the positive **Z** direction is towards the bottom). However, if **1**, **2** or **3** are specified, these represent three linearly independent directions but not necessarily coincidental with the regular coordinates, so an azimuth and a dip *must* be specified, depending on the type of code. The same is true of the **H** (hydrophone) code. See reference above for details.

Note also how we freely mix the two syntactic ways of specifying a dictionary in YAML, with curly parentheses or indentation. You can use whatever syntax you prefer. You only need to pay attention to curly parentheses if you choose to use that syntax.

The order of the keys `sensor`, `preamplifier` and `datalogger` are arbitrary, as there can be no ambiguity, but it is good to remember that their stages will always be processed in the logical order: `sensor`, `preamplifier` and `datalogger`.

5.4.4 Configurations

So far so good. We have learnt how to specify default components through the default and how to override them. This is pretty flexible, but we can get more flexible still. This is done through *configuration definitions* and *configuration selection*. Configuration definitions are specified at the instrument component level, and thus will be treated in the next section. Suffice it to say here that configuration definitions can take any field at the lower levels and change it either by overriding it or by adding it to a preexisting default configuration. This is by *selecting* a configuration at the channel level. A configuration selection field can specify a configuration for each of the three instrument components in a channel: `sensor`, `preamplifier` and `datalogger`. The configuration *must* exist as a configuration definition. Let's add two things to the example: a default configuration for all channels in `default` and a channel 3 that overrides this configuration.

```
channels:
  default:
    sensor: {$ref: "sensors/NANOMETRICS_T240_SINGLESIDED.sensor.yaml#sensor"}
    preamplifier:
      {$ref: "preamplifiers/LCHEAPO_BB0BS.preamplifier.yaml#preamplifier"}
    datalogger: {$ref: "dataloggers/LC2000.datalogger.yaml#datalogger"}

    preamplifier_configuration: "0.225x"

  "1": {orientation_code: {"2": {azimuth.deg: [9, 0], dip.deg: [0, 0]}} }
```

(continues on next page)

(continued from previous page)

```

"2":
  orientation_code:
    "1":
      azimuth.deg: [90, 9]
      dip.deg: [0, 0]

"3":
  orientation_code: "Z"
  preamplifier_configuration: "1x"

"4":
  orientation_code : {"H": {azimuth.deg: [0, 0], dip.deg: [90, 0]}}
  preamplifier: {$ref: "preamplifiers/LCHEAPO_DPG.preamplifier.yaml#preamplifier"}
  sensor: {$ref: "sensors/SIO_DPG.sensor.yaml#sensor"}

```

This code specifies configurations, which can be for sensor, preamplifier or datalogger; in this case, simply for the preamplifier. The configuration selected is called “0.225x” and is a gain multiplier, as will be seen shortly. This value will be used in all channels, except channel 3, where it will be changed to “1x”. In the end, the four channels specified above will be the same as typing this:

```

channels:
  "1":
    sensor: {$ref: "sensors/NANOMETRICS_T240_SINGLESIDED.sensor.yaml#sensor"}
    preamplifier:
      {$ref: "preamplifiers/LCHEAPO_BBOBS.preamplifier.yaml#preamplifier"}
    datalogger: {$ref: "dataloggers/LC2000.datalogger.yaml#datalogger"}

    orientation_code: {"2": {azimuth.deg: [90, 9], dip.deg: [0, 0]}}

    preamplifier_configuration: "0.225"

  "2":
    sensor: {$ref: "sensors/NANOMETRICS_T240_SINGLESIDED.sensor.yaml#sensor"}
    preamplifier:
      {$ref: "preamplifiers/LCHEAPO_BBOBS.preamplifier.yaml#preamplifier"}
    datalogger: {$ref: "dataloggers/LC2000.datalogger.yaml#datalogger"}

    orientation_code: {"1": {azimuth.deg: [0, 0], dip.deg: [0, 0]}}

    preamplifier_configuration: "0.225x"

  "3":
    sensor: {$ref: "sensors/NANOMETRICS_T240_SINGLESIDED.sensor.yaml#sensor"}
    preamplifier:
      {$ref: "preamplifiers/LCHEAPO_BBOBS.preamplifier.yaml#preamplifier"}
    datalogger: {$ref: "dataloggers/LC2000.datalogger.yaml#datalogger"}

    orientation_code: Z
    preamplifier_configuration: "1x"

  "4":

```

(continues on next page)

(continued from previous page)

```

sensor: {$ref: "sensors/SIO_DPG.sensor.yaml#sensor"}
preamplifier: {$ref: "preamplifiers/LCHEAPO_DPG.preamplifier.yaml#preamplifier"}
datalogger: {$ref: "dataloggers/LC2000.datalogger.yaml#datalogger"}

orientation_code: {"H": {azimuth.deg: [0, 0], dip.deg: [90, 0]}}

preamplifier_config: "1x"

```

Channel modifications

As seen in the last chapter, channel configurations can also be modified. The rationale behind this feature is that the user has a stable database of instruments which may occasionally suffer last-minute or punctual modifications, for example, in the serial number of an equipment (when it is replaced by another one as an emergency measure when it's malfunctioning or gain of a stage). OBS equipment is highly malleable and *obsinfo* is conceived to reflect that malleability. Channel modifications are performed at the *station* level but can potentially change *any* field from instrumentation level down. However, this is a more complex topic that falls outside of a beginner's tutorial. It will be treated in the [Advanced Topics](#) documentation.

5.4.5 Notes and extras

This file is complex, so it's a good place to talk about notes and extras. These are optional fields. Notes can occur in any information file. They are documentation that can be used to remind users of the specifics of the information file. They will **not** be reflected in StationXML to avoid clutter.

Extras are key:value pairs that document attributes that do not exist in the information file specification. They **are** reflected as remarks in the StationXML file. For this reason and to avoid clutter, they are only available at three levels: network, station and channel, which are the levels in StationXML which permit remarks. An earlier version of *obsinfo* permitted extras at any level. The result was considerable clutter at the response stages level, as stages are repeated and the same information is therefore repeated over and over.

As an example, let's assume we have an "octopus" sensor where the serial number of the sensor is not specified. This is because we have actually several sensors with different serial numbers, enclosed in spheres. How do we convey that information? There are two ways. One is in a list of notes at the end of the *sensor* file:

```

notes:
- "INSU-IPGP OBS park sphere sensor pairs are: Sphere01-133, Sphere02-132,"
- "Sphere03-134, Sphere04-138, Sphere05-137, Sphere06-830, Sphere07-136,"
- "Sphere08-829, Sphere09-826"

```

This associates serial numbers to the spheres. However, this will not be reflected in the StationXML file. The other YAML syntax for lists is possible too:

```

notes: ["INSU-IPGP OBS park sphere sensor pairs are: Sphere01-133, Sphere02-132,", "Sphere03-134, Sphere04-138, Sphere05-137, Sphere06-830, Sphere07-136,", "Sphere08-829, Sphere09-826"]

```

Or we can use the *extras* dictionary, but not in the sensor file but in the instrumentation one:

```

extras:
  "Description": "Serial numbers for sensors"
  "Sphere03": "134"

```

(continues on next page)

(continued from previous page)

```

"Sphere04": "138"
"Sphere05": "137"
"Sphere06": "830"
"Sphere07": "136"
"Sphere08": "829"
"Sphere09": "826"

```

Complete example

This is a real file. The order of the fields may change from the examples above. As mentioned, this is immaterial.

```

---
format_version: "0.110"
revision:
  date: "2019-12-19"
  authors:
    - {$ref: "authors/Wayne_Crawford.author.yaml#author"}
    - {$ref: "authors/Romuald_Daniel.author.yaml#author"}
instrumentation:
  operator: {$ref: "operators/INSU-IPGP.facility_info.yaml#facility_info"}
  equipment:
    model: "BBOBS1"
    type: "Broadband Ocean Bottom Seismometer"
    description: "LCHEAPO 2000 BBOBS 2012-present"
    manufacturer: "Scripps Inst. Oceanography - INSU"
    vendor: "Scripps Inst. Oceanography - UNSU"

  channels:
    default:
      datalogger: {$ref: "dataloggers/LC2000.datalogger.yaml#datalogger"}
      preamplifier:
        {$ref: "preamplifiers/LCHEAPO_BBOBS.preamplifier.yaml#preamplifier"}
      sensor: {$ref: "sensors/NANOMETRICS_T240_SINGLESIDED.sensor.yaml#sensor"}
      preamplifier_configuration: "0.225x"

    "1": {orientation_code: {"2": {azimuth.deg: [90, 0], dip.deg: [0, 0]} }
    "2": {orientation_code: {"1": {azimuth.deg: [0, 0], dip.deg: [0, 0]} }
    "3":
      orientation_code: "Z"
      selections:
        preamplifier: "1x"
    "4":
      orientation_code : {"H": {azimuth.deg: [0, 0], dip.deg: [90, 0]}
      preamplifier: {$ref: "preamplifiers/LCHEAPO_DPG.preamplifier.yaml#preamplifier"}
      sensor: {$ref: "sensors/SIO_DPG.sensor.yaml#sensor"}

```

- [Next page, Building instrument component files](#)
- [Previous page](#)
- [Back to start](#)

5.5 Building instrument component files with response stages: sensors and preamplifiers

Sensor, preamplifier and datalogger are all instrument components. While `InstrumentComponent` is not a key in information files, it is a class in Python used to inherit attributes and methods to all three component classes. All instrument components share the same attributes and sensor and datalogger add one each on their own. Components in an instrument are always understood to come in the same order, and are processed in that order: first the sensor, then possibly a preamplifier, usually analog, and then the datalogger.

What characterizes all components is that they have an ordered list of response stages. While the order of the components themselves is predetermined, the order of the stages *must* be specified. The order of all stages is then determined as sensor stage 1, sensor stage 2,..., preamplifier stage 1, preamplifier stage 2,..., datalogger stage 1, datalogger stage 2,...

5.5.1 A simple sensor component

A sensor is, as it is well-known, any kind of transducer that senses a seismic signal and transduces it to an electrical signal, typically an analog one.

Let's see an example of a sensor component.

```
---
format_version: "0.110"
revision:
  date: "2017-11-30"
  authors:
    - {$ref: "authors/Wayne_Crawford.author.yaml#author"}
sensor:
  equipment:
    model: "Trillium T240"
    type: "Broadband seismometer"
    description: "Trillium T240 seismometer, single-sided connection"
    manufacturer: "Nanometrics, Inc"
    vendor: "Nanometrics, Inc"
```

We have an equipment section, just as the instrumentation level, as sensors can have different manufacturers from the rest of the equipment. The description allows to add enough detail so we can identify this sensor. Then we have the seed codes section. Seed codes are coded descriptions of such elements as the band base, the instrument type and the orientation. The codes of the first two follow the FDSN standard, as explained [here](#).

```
seed_codes:
  band_base: "B"
  instrument: "H"
```

Seed codes are only present in sensors. No other component has them. Seed codes are based on an FDSN standard and consist of three characters. The first specifies the band_base, the second the instrument type. A third one, orientation, with azimuth and dip, is specified at the channel level, although in the StationXML file it will part of the seed code.

The value of `polarity` should be “+” if an upward motion or a pressure increase results in a positive voltage, and “-” otherwise.

5.5.2 Stages

Now, let's take a look at the next section, response stages. As is readily seen in the example, `response_stages` are a list of stages. Being a list, individual stages have no label or key, which would make them dictionary items rather than list items. As they are (usually) not referenced elsewhere (the glaring exception being channel modifications), this simplifies the syntax. In this case, we only include a single stage, as a reference to a stage file, which is the recommended best practice. Stages are found in a stage folder.

```
response_stages
  - $ref: "stages/Trillium_T240_SN1-399-singlesided_theoretical.stage.yaml#stage"
```

Response stages are used in all three components. While StationXML lists all stages separately from the components, *obsinfo* associates conceptually stages to components by way of their functionality. In the end, however, stages will be grouped together and numbered from the sensor stages to the datalogger ones, all in sequence.

This ends the presentation of a simple sensor file. But the important part of components, their flexibility, lies ahead.

5.5.3 Configuration definitions

This is the place where the full power of *obsinfo* manifests itself. The application allows several configuration definitions to coexist in any component file. This means that we can have a virtual sensor or datalogger which can potentially have any number of configurations, so we can form a library of component files. Only when they are added to an instrument (or, if you like to think it that way, to a channel), will one particular configuration be “instantiated” and a real component will be described by the file. This occurs with the field `configuration_selections` in the instrumentation file. That value selects one configuration among all the configuration definitions. But we also allow a default configuration, so if no configuration is selected at the channel level, this will be the actual configuration selected. Let's modify our simple sensor file adding configurations:

```
---
format_version: "0.110"
revision:
  date: "2017-11-30"
  authors:
    - {$ref: "authors/Wayne_Crawford.author.yaml#author"}
sensor:
  equipment:
    model: "Trillium T240"
    type: "Broadband seismometer"
    description: "Trillium T240 seismometer, negative shorted to ground"
    manufacturer: "Nanometrics, Inc"
    vendor: "Nanometrics, Inc"

  seed_codes:
    band_base: "B"
    instrument: "H"

  configuration_default: "SINGLE-SIDED_SN1-399"

  configuration_definitions:
    "SINGLE-SIDED_SN1-399" :
      configuration_description: "serial numbers 1-399"
      response_stages:
        - $ref: "responses/Trillium_T240_SN1-399-singlesided_theoretical.stage.yaml
#stage"
```

(continues on next page)

(continued from previous page)

```

"single-sided_sn400plus" :
  configuration_description: "serial numbers 400+"
  response_stages:
    - $ref: "responses/Trillium_T240_SN400-singlesided_theoretical.stage.yaml
↪#stage"

```

This file requires a lot of commentary. Let's start with the resulting configuration. Note that we have added two configuration definitions, which are specified as a dictionary (i.e. they have labels, key/value pairs), "SINGLE-SIDED_SN1-399" and "SINGLE-SIDED_SN400plus". This is a real example in which a component has different behaviour depending on its serial number (below or above 400), which calls for two differently configured stages. If no sensor configuration is selected in the instrumentation file, the result would be to use the default configuration, so the file above would be the same as this:

```

---
format_version: "0.110"
revision:
  date: "2017-11-30"
  authors:
    - {$ref: "authors/Wayne_Crawford.author.yaml#author"}

sensor:
  equipment:
    model: "Trillium T240"
    type: "Broadband seismometer"
    description: "Trillium T240 seismometer, negative shorted to ground [config:↪
↪serial numbers 1-399]"
    manufacturer: "Nanometrics, Inc"
    vendor: "Nanometrics, Inc"

  seed_codes:
    band_base: "B"
    instrument: "H"

  response_stages:
    - $ref: "responses/Trillium_T240_SN1-399-singlesided_theoretical.stage.yaml#stage"

```

response_stages is added from the default configuration definition. No surprises here. But look at what happened in description. We didn't override the existing description, we *concatenated* the new one to the old one. This is an exception to the way all other fields behave. The idea is to be more specific about the description according to the configuration. This could possibly be achieved with YAML anchors, but unfortunately YAML does not concatenate strings, so we need to do it this way, with an exception to the general overriding rule.

Now, if we had selected configuration "SINGLE-SIDED_SN400plus" in the instrumentation file (in the config_selections section), the result would be:

```

---
format_version: "0.110"
revision:
  date: "2017-11-30"
  authors:
    - {$ref: "authors/Wayne_Crawford.author.yaml#author"}

sensor:

```

(continues on next page)

(continued from previous page)

```

equipment:
  model: "Trillium T240"
  type: "Broadband seismometer"
  description: "Trillium T240 seismometer, negative shorted to ground [config:↵
serial numbers 400+]"
  manufacturer: "Nanometrics, Inc"
  vendor: "Nanometrics, Inc"

seed_codes:
  band_base: "B"
  instrument: "H"

response_stages:
  - $ref: "responses/Trillium_T240_SN400-singlesided_theoretical.stage.yaml#stage"

```

At any rate, the philosophy is to have all these configurations added to the component file from the start, so we don't change the file much; but, of course, if needs be, you can add more configurations anytime.

5.5.4 Complete example sensor file

```

---
format_version: "0.110"
revision:
  date: "2017-11-30"
  authors:
    - {$ref: "authors/Wayne_Crawford.author.yaml#author"}
sensor:
  equipment:
    model: "Trillium T240"
    type: "Broadband seismometer"
    description: "Trillium T240 seismometer, negative shorted to ground"
    manufacturer: "Nanometrics, Inc"
    vendor: "Nanometrics, Inc"

  seed_codes:
    band_base: "B"
    instrument: "H"

  configuration_default: "SINGLE-SIDED_SN1-399"

  configuration_definitions:
    "SINGLE-SIDED_SN1-399" :
      configuration_description: "serial numbers 1-399"
      response_stages:
        - $ref: "responses/Trillium_T240_SN1-399-singlesided_theoretical.stage.yaml
↵#stage"
    "SINGLE-SIDED_SN400plus" :
      configuration_description: "serial numbers 400+"
      response_stages:
        - $ref: "responses/Trillium_T240_SN400-singlesided_theoretical.stage.yaml
↵#stage"

```

(continues on next page)

(continued from previous page)

notes:

- "INSU-IPGP OBS park sphere sensor pairs are: Sphere01-133, Sphere02-132,"
- "Sphere03-134, Sphere04-138, Sphere05-137, Sphere06-830, Sphere07-136,"
- "Sphere08-829, Sphere09-826"

5.5.5 Preamplifier configuration definitions

Preamplifiers are, in fact, the simplest components. They only have `equipment`, `response_stages`, `configuration_default` and `configuration_definitions`, already explained above. Thus, we limit ourselves to showing an example, noting the the configuration definitions are based on gain, not serial number as in the sensor example before. Remember that labels for configuration definitions are totally arbitrary, so you can make your own choice as to how to characterize the configurations.

```
---
format_version: "0.110"
revision:
  date: "2017-11-30"
  authors:
    - $ref: "authors/Wayne_Crawford.author.yaml#author"

preamplifier:
  equipment:
    model: "BBOBS-GAIN"
    type: "Analog gain card"
    description: "INSU BBOBS gain card"
    manufacturer: "SIO or IPGP"
    vendor: ~

  configuration_default: "1x"

  configuration_definitions:
    "0.225x":
      configuration_description: "0.225x gain"
      response_stages:
        - $ref: "responses/INSU_BBOBS_gain0.225_theoretical.stage.yaml#stage"
    "1x":
      configuration_description: "1x gain"
      response_stages:
        - $ref: "responses/INSU_BBOBS_gain1.0_theoretical.stage.yaml#stage"
```

In the next section we will see how to configure a datalogger information file.

- *Next page, Building a datalogger information file*
- *Previous page*
- *Back to start*

5.6 Building a datalogger information file

Dataloggers are the components used to record the data treated by the instrument stages before. Their configuration files might get quite complex due to the number of necessary stages.

Dataloggers have the same common fields of any other instrument component, with two extra fields: `delay_correction` and `sample_rate`, which is the overall sample rate of the complete instrument.

delay_correction

All stages can have nominal delays, but these delays sometimes need to be corrected. The `delay_correction` field accomplishes this. In StationXML **correction** is an attribute of each individual stage. However, we find it is not realistic to be able to apply each individual correction and opt instead to apply a general correction to all stages. We accomplish this by using two attributes in *Stage*: the (nominal) StationXML **delay** and **correction**. We consider two cases:

1. `delay_correction` is **not** specified in the datalogger information file. Then, the corresponding StationXML **correction** attribute in each stage for all components will be set to the **delay** value corresponding to that stage.
2. `delay_correction` is specified in the datalogger information file. Then, the corresponding StationXML **correction** attribute in each stage for all components is set to zero, *except* for the last stage (which is of course the last datalogger stage), which is set to the `delay_correction` value, thus becoming the total correction value.

Aside from this, the rest of the attributes have been discussed in the last section.

5.6.1 Datalogger configuration definitions

The following paragraph requires the reader to have a minimal knowledge of signal treatment.

The code below is a real datalogger configuration file. We see that this example has several response stages in each configuration, based this time on the sample rate. This is due to the fact that each stage with the FIR2 and FIR3 filters has a decimation factor of 2: each one divides the sample rate by two. FIR1 is actually an ADC, an analog to digital converter, all previous stages in this instrument being analog, in particular the one in the previous component, the preamplifier. FIR1 outputs a 32000 sps sample rate. Thus, to get to a final 1000 sps sample rate we need four FIR2 and one FIR3, each halving the sample rate. FIR2 and FIR3 have different coefficients and thus both are necessary. This means we need at least one FIR1, one FIR2 and one FIR3. To keep halving the sample rate we simply add more FIR2. So it's simple now to see now the difference in configurations: it's simply adding an extra FIR2 each time.

```
---
format_version: "0.110"
revision:
  date: "2019-12-20"
  authors:
    - $ref: 'authors/Wayne_Crawford.author.yaml#author'
notes:
  - "Delay correction is hard-coded to 29 samples in LCHEAPO software"

datalogger:
  equipment:
    model: "CS5321/22"
    type: "delta-sigma A/D converter + digital filter"
    description: "CS5321/22 delta-sigma A/D converter + FIR digital filter"
    manufacturer: "Cirrus Logic"
    vendor: "various"

  configuration_default: "125 sps"
```

(continues on next page)

(continued from previous page)

```

configuration_definitions:
  "62.5sps":
    sample_rate: 62.5
    delay_correction: 0.464
    response_stages:
      - $ref: "responses/CS5321_FIR1.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR3.stage.yaml#stage"
  "125sps":
    sample_rate: 125
    delay_correction: 0.232
    response_stages:
      - $ref: "responses/CS5321_FIR1.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR3.stage.yaml#stage"
  "250sps":
    sample_rate: 250
    delay_correction: 0.116
    response_stages:
      - $ref: "responses/CS5321_FIR1.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR3.stage.yaml#stage"
  "500sps":
    sample_rate: 500
    delay_correction: 0.058
    response_stages:
      - $ref: "responses/CS5321_FIR1.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR3.stage.yaml#stage"

```

(continues on next page)

(continued from previous page)

```

"1000sps":
  sample_rate: 1000
  delay_correction: 0.029
  response_stages:
    - $ref: "responses/CS5321_FIR1.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
    - $ref: "responses/CS5322_FIR3.stage.yaml#stage"

```

As can be seen, configuration definition labels are flexible and can suit any purpose imagined by the user. The best practice is to keep them short, explicit and consistent among different selectable configurations in the same group.

Next we will see stages and filters in more detail.

- [Next page, Building stage information files](#)
- [Previous page](#)
- [Back to start](#)

5.7 Building a stage information file with different filters

Conceptually, stages are each of an electronic block diagram “boxes”. They are usually implemented as a single printed circuit connected to the instrument bus. An instrument component has an ordered list of stages. Each stage has certain signal treatment characteristics.

It is important to note that, since stages are chained,

- Output units of stage and input units of the next stage must match.
- In digital stages, the output sample rate of one stage must match the input sample rate of the next one.
- Declared sample rate of the instrument as a whole must match calculated sample rate of the response ensemble.

To allow flexible use of stages, as exemplified in the datalogger information file above, it is a good idea *not* to specify input sample rates (output sample rates cannot be specified, see below) for all but the first stage. For example, the FIR2 stages in the datalogger example above only specify decimation factor. This means that, irrespective of input sample rate, they will have the output sample rate specified by the decimation factor.

Both conditions are checked by *obsinfo*.

In the current implementation, all stages have one, and exactly one filter associated. This may seem a little strange, as some stages are not properly filters, but rather analog and digital amplifiers (they only increment gain) or ADCs, analog-digital converters. This is idiosyncratic. It seems StationXML does not allow stages that are not some sort of filter. Therefore, as will be seen, these stages are implemented as filters in *obsinfo*.

Let's profit from this to see what a stage with a very simple “filter” in the above sense looks like. This is a stage for a preamplifier. It is analog and only has a gain, with no other processing. We have a specification of input and output units and a gain, composed of a value and a frequency where that gain value is measured. It has an associated “filter” of type ANALOG. All these are required fields. We also have an optional description, which is nonetheless recommended.

```

---
format_version: "0.110"
stage:

```

(continues on next page)

(continued from previous page)

```
description : "PREAMPLIFIER: BBOBS 0.225x"
input_units : {name: "V", description: "VOLTS"}
output_units : {name: "V", description: "VOLTS"}
gain : {value : 0.225, frequency: 0}
filter :
  type : "ANALOG"
polarity: "+"
```

Next we can see another pseudo-filter, an analog to digital converter used as the first stage of a datalogger:

```
---
format_version: "0.110"
revision:
  date: "2017-11-30"
  authors:
    - $ref: "authors/Wayne_Crawford.author.yaml#author"

notes:
  - "From CS5321-22_F3.pdf"

stage:
  description : "DIGITIZER - CS5321 Delta-Sigma A/D converter" # optional
  input_units : { name : "V", description: "Volts"}
  output_units : { name : "counts", description: "Digital Counts"}
  input_sample_rate : 256000
  decimation_factor : 8
  gain :
    frequency: 0
    value: 1165084 # counts/V
  filter:
    type : "AD_CONVERSION"
    input_full_scale : 9 # 9 V pp
    output_full_scale : 10485760 # 4FFFFFF@Vref and B000000@-Vref
```

Although it is obvious that the input is analog, we specify an `input_sample_rate` in agreement with StationXML specifications. The output rate, as mentioned above, is never specified, but can easily be obtained from this and the `decimation_factor` by division. In the example, the output sample rate will be 32000 sps. For the time being, we will ignore the other fields in `filter`, which will be discussed in the next section.

Note the use of the `notes` attribute, which will **not** be processed as comments in StationXML. Other optional attributes don't appear here to keep the discussion simple but can be readily consulted in the corresponding *Stage*. However, there are three attributes that should be noticed.

5.7.1 delay, offset and correction attributes

Digital filters can have an `offset`, which specifies how samples the peak of an impulse will be offset by the filter. This is specified at the filter level.

The stage level has an attribute called `delay`. If `delay` is not specified but `offset` is in the associated filter, `delay` is calculated by dividing `offset` by the input sample rate. If both `delay` and `offset` are specified, their specified values are kept untouched.

`correction` is an attribute in StationXML which is calculated, as mentioned in the previous section, using the datalogger field `delay_correction`. It should **not** be specified by the user.

We see in this example a stage without `input_sample_rate` but with `decimation_rate` of 2, which divide the `output_sample_rate` of the previous stage by 2, as mentioned in the introduction to this section. This is precisely the stage FIR3 that was used in the previous page as a datalogger stage example. The other thing worth noting is the reference to a filter file in the folder `filters/`.

```
---
format_version: "0.110"
revision:
  date: "2017-11-30"
  authors:
    - $ref: "authors/Wayne_Crawford.author.yaml#author"
notes: ["From CS5322_Filter.pdf"]
stage:
  decimation_factor : 2
  gain :             {value: 1, frequency: 0}
  input_units :     { name : "counts", description: "Digital Counts"}
  offset: 50

  description : "DECIMATION - CS5322 FIR3 (linear phase)"
  filter:
    $ref: "filters/CirrusLogic_CS5322_FIR3.filter.yaml#filter"
  extras:
    DBIRD_response_type : "THEORETICAL"
```

5.7.2 Polarity

Each stage can have a polarity: if omitted it is assumed to be "+". A "+" polarity means:

- For a seismometer, a positive voltage corresponds to a movement **UP**.
- For a hydrophone, a positive voltage corresponds to an **INCREASE** in pressure
- For a non-sensor stage, the passband output has the **SAME** polarity as the input in the passband

A "-" polarity means:

- For a seismometer, a positive voltage corresponds to a movement **DOWN**.
- For a hydrophone, a positive voltage corresponds to a **DECREASE** in pressure
- For a non-sensor stage, the passband output has the **OPPOSITE** polarity as the input in the passband

By multiplying the polarities in a channel's stages we get the channel's polarity. For seismometer and hydrophone channels, a positive polarity corresponds to dip = -90° and a negative polarity to dip = 90°

- [Next page, Building a filter information file](#)
- [Previous page](#)
- [Back to start](#)

5.8 Building a filter information file for different filters

As pointed out in the last section, all stages have an associated filter, even if we can't properly call it a filter, due to the ideosyncratic behavior of StationXML. Some of the normal filters are familiar:

- *PolesZeros* - Any kind of digital filter specified by its poles and its zeros. Use this for Laplace transforms and IIR filters.
- *FIR* - Finite Input Response digital filter
- *Coefficients* - A FIR expressed with coefficients of transfer function
- *ResponseList* - A digital filter with all responses expressed as frequency, amplitude and phase.

Others are not properly filters:

- *ADConversion* - Analog to digital conversion stage
- *Analog* - gain only analog stage-
- *Digital* gain only digital stage

For more details on each one go to their corresponding Class pages. Here are some examples on how to express different filters (for examples of Analog and ADConversion, see last section).

All filters have a type, which is specified in the list above, and digital ones have an `offset`, which is the number of pulses to be skipped at the start of the signal input. `offset` is used to calculate delay, as explained in the last section.

5.8.1 PolesZeros

A PolesZeros digital filter (as defined in the field `type`) specifies a transfer function type, a normalization factor and frequency, and the respective poles and zeros:

```
---
format_version: "0.110"
revision:
  date: "2017-11-30"
  authors:
    - $ref: "authors/Wayne_Crawford.author.yaml#author"

filter:
  type: "PolesZeros"
  transfer_function_type: "LAPLACE (RADIANS/SECOND)"
  normalization_factor : 1
  normalization_frequency : 0
  zeros :
    - [0.0, 0.0]
    - [0.0, 0.0]
    - [-72.5, 0.0]
    - [-159.3, 0.0]
    - [-251, 0.0]
```

(continues on next page)

(continued from previous page)

```

- [-3270.0, 0.0]
poles :
- [-0.017699, 0.017604]
- [-0.017699, -0.017604]
- [-85.3, 0.0]
- [-155.4, 210.8]
- [-155.4, -210.8]
- [-713, 0]
- [-1140, -0]
- [-4300, -0]
- [-5800, -0]
- [-4300, 4400]
- [-4300, -4400]
offset: 0

```

notes:

- poles et zeros d'un Trillium T240 no de serie 400+
- d'apres le fichier Trillium240_UserGuide_15672R7.pdf de Nanometrics.

5.8.2 FIR

A FIR filter can be specified by a symmetry and a set of numerator coefficients of the transfer function. The symmetry specification permits to specify the filter without having to repeat values. But a FIR filter can also be specified by the numerator *and* denominator coefficients of the transfer function, in which case the type “Coefficients” is used. For more information, see the corresponding class. Finally, the specification can take the form of a list of responses, which uses the type “ResponseList”. Again, this is straightforward. The syntax can be looked up in the corresponding class.

```

---
format_version: "0.110"
revision:
  date: "2017-11-30"
  authors:
    - $ref: "authors/Wayne_Crawford.author.yaml#author"

filter:
  type: "FIR"
  symmetry: "NONE"
  offset: 6
  coefficients:
    - 2.44141E-4
    - 0.00292969
    - 0.0161133
    - 0.0537109
    - 0.12085
    - 0.193359
    - 0.225586
    - 0.193359
    - 0.12085
    - 0.0537109
    - 0.0161133
    - 0.00292969

```

(continues on next page)

(continued from previous page)

```
- 2.44141E-4
```

5.8.3 Analog and Digital

Analog and digital “filters” are stages that basically amplify the signal, whether analog or digital, with the gain specified, as usual, at the stage level. Analog filters can invert the polarity, so this needs to be specified with a `polarity` attribute.

```
---
format_version: "0.110"
revision:
  date: "2017-11-30"
  authors:
    - $ref: "authors/Wayne_Crawford.author.yaml#author"

filter:
  type: "Analog"
  polarity: "-"
```

5.8.4 ANALOG to DIGITAL Converter

This is another type of pseudo-filter which has the input voltage range and the output voltage range as attributes:

```
---
format_version: "0.110"
revision:
  date: "2017-11-30"
  authors:
    - $ref: "authors/Wayne_Crawford.author.yaml#author"

filter:
  type: "ADConversion"
  input_full_scale : 9 # 9 V pp
  output_full_scale : 10485760 # 4FFFFFF@Vref and B000000@-Vref
```

- [Next page, Conclusion](#)
- [Previous page](#)
- [Back to start](#)

5.9 Summary

As a summary, remember *obsinfo* strives for reuse and flexibility.

1. Try to use the hierarchy referencing files as much as possible. This allows reuse across files and different campaigns.
2. Use `yaml_anchors` for reuse in the *same file*
3. Start from simple files with only required fields and build up from that

4. You can add notes at any level. You can add extras only in network, station and instrumentation files. Only extras and comments will be reflected as comments in StationXML.
5. Make extensive use of the * default channel to avoid repeating redundant information in different channels.
6. Make extensive use of configuration_definitions to avoid duplicating the same components with little variations
7. Remember: * channel default preferably for complete substitutions of a component or of a default configuration. config_selection for changing the list of response_stages, filter parameters, equipment fields.

File templates are available to help you start writing your information files.

5.9.1 Conclusion

This finishes our tutorial. For more detailed information please review the *Classes* hierarchy where attributes are described in detail and some optional attributes that were not discussed here for the sake of brevity are explained. Have fun using *obsinfo*!

If you find any issues or have any questions please use the Issues functionality of gitlab: <https://www.gitlab.com/resif/obsinfo/issues>

- *Previous page*
- *Back to start*

INFORMATION FILES

Here are examples of information files, from the most basic to the most complete. You can also see the schemas at ...

6.1 Overview

We present here the hierarchy of information files. These files are generally divided into sub-files using the *\$ref*: operator inherited from JSONref. We will present:

- 1) the “standard” but not required structure of these files
- 2) A full description of the format, divided into these structures
- 3) A full, undivided, tree of all possible fields

6.1.1 Source

The definition (JSON Schemas) for the information files are in `obsinfo/data/schemas`

6.1.2 1) Basic level diagram

Only required fields are shown, except a few very commonly-used fields, surrounded by “*”. atomic objects described below are surrounded by <>

```
network: <network>
  operator:
    <operator>
  network_info:
    <network_info>
  stations:
    {STATIONNAME1}: <station>
      site: string
      start_date: string
      end_date: string
      location_code: string
      locations:
        {LOCATION_1}: <location>
        {LOCATION_2}: <location>
        ...
  instrumentation: <instrumentation>
```

(continues on next page)

(continued from previous page)

```

equipment:
  <equipment object>
channels:
  default: <channel>
    datalogger: <datalogger>
    equipment:
      <equipment>
    sample_rate: number
    response_stages:
      - <stage>
        input_units: <units>
        name: string
        description: string
        output_units: <units>
        gain:
          value: float
        filter: <filter>
        type: string
      - <stage>
      - ...
    *preamplifier: <preamplifier>*
    equipment:
      <equipment>
    response_stages:
      - <stage>
      - <stage>
      - ...
    sensor: <sensor>
    equipment:
      <equipment>
    seed_codes:
      band_base: "B" or "S"
      instrument: single character
    response_stages:
      - <stage>
      - <stage>
      - ...
    {SPECIFIC-CHANNEL1}: subset of <channel>
    {SPECIFIC-CHANNEL2}: subset of <channel>
    ...
  {STATIONNAME2}:
    ...

```

6.1.3 2) Atomic level diagram with comments

Starred fields are optional. If you put a level in a separate file, it is good practice to use the following file structure:

```
format_version: <format_version>
*revision: <revision>*
*notes: list of strings*
level: <level>
```

6.1.4 Major objects

network

```
operator: <operator>
*stations_operator: <operator>*
*restricted_state: {}*
*comments: list of strings*
*extras: <free-form object>*
campaign_ref_name: string
network_info: <network_info>
stations:
  <STATIONNAME1>:
    <station>
  <STATIONNAME2>:
    <station>
  ...
```

station

```
site: string
start_date: string
end_date: string
location_code: string
*serial_number: string*
*operator: <operator>*
instrumentation: <instrumentation>
*channel_modifications: {}*
locations: object list of <location>
*notes: list of strings*
*comments: list of strings or <stationxml_comment>*
*extras: <free-form object>*
*processing:*
  - *clock_correction_linear: {}*
  - *clock_correction_leapsecond: {}*
```

instrumentation

```
*operator: {}*  
equipment: {}  
channels:  
  default:  
    <channel>  
  <SPECIFIC-CHANNEL1>:  
    subset of <channel>  
  <SPECIFIC-CHANNEL2>:  
    subset of <channel>  
  ...
```

channel

```
*orientation_code: string or {}*  
datalogger:  
  <datalogger>  
*preamplifier:*  
  <preamplifier>  
sensor:  
  <sensor>  
*preamplifier_configuration: string*  
*sensor_configuration: string*  
*datalogger_configuration: string*  
*location_code: string* # otherwise inherits from station  
*comments: list of strings*  
*extras: <free-format object>*
```

datalogger

```
<GENERIC_COMPONENT>  
sample_rate: number  
*delay_correction: number*
```

preamplifier

```
<GENERIC_COMPONENT>
```

sensor

```
<GENERIC_COMPONENT>
seed_codes:
```

GENERIC_COMPONENT

```
equipment:
  <equipment>
*configuration_default: string*
*response_modifications: {}*
*notes: list of strings*
*response_stages:*
  - <stage object>
  - <stage object>
  - ...
*configuration_definitions:*
  {CONFIGURATION_1}: <subset of datalogger, sensor, or preamplifier object>
  {CONFIGURATION_2}: <subset of datalogger, sensor, or preamplifier object>
  ...
```

stage

```
input_units: string
output_units string
gain: float
*name: string*
*description: string*
*decimation_factor: integer*
*delay: number*
*calibration_date: string*
*polarity: '+' or '-'*      # default is '+'
*input_sample_rate: number*
*filter:*
  <filter>
```

filter

fields depend on type:

```
type: "PoleZeros"
poles: list
zeros: list
*offset: float*
*transfer_function_type*: "LAPLACE (RADIANS/SECOND)", "LAPLACE (HERTZ)", or "DIGITAL (Z-
↳ TRANSFORM)"
*normalization_frequency: float*
*normalization_factor: float*
```

```
type: "FIR"
symmetry: "EVEN", "ODD" or "NONE"
coefficients: list
offset: float
"coefficient_divisor": float
```

```
type: "Coefficients"
numerator_coefficients: list
denominator_coefficients: list
*transfer_function_type*: "ANALOG (RADIANS/SECOND)", "ANALOG (HERTZ)" or "DIGITAL"*
*offset: float*
```

```
type: "ResponseList"
elements: list
*offset: float*
```

```
type: "Polynomial"
# Not implemented, had no examples to verify with
```

```
type: "ADConversion"
input_full_scale: float
output_full_scale: float
```

```
type: "Analog"
```

```
type: "Digital"
```

Minor objects

author_, **operator_**, **location_base_** and **network_info_** are often in separate files, should probably have their own schema file.

equipment_ is widespread enough that it should probably have its own schema file as well

author:

```

first_name: string
last_name: string"
*institution: string*
*email: string*
*phones: array*

```

equipment:

```

type: string
description: string
manufacturer: string
model: string
*vendor: string*
*serial_number: string*
*installation_date: string, date formatted*
*removal_date: string, date formatted*
*calibration_dates: array of strings, date format*

```

location:

```

base:
  uncertainties.m:
    lat: number (in meters)
    lon: number (in meters)
    elev: number (in meters)
  depth.m: number
  geology: string
  vault: string
  *localisation_method: string*
position:
  lat: number (in degrees)
  lon: number (in degrees)
  elev: number (in meters)

```

network_info:

```

code: string
name: string
start_date: string, date formatted
end_date: string, date formatted
description: string
*comments: list of strings or <stationxml_comment>*
*website: string*

```

operator:

```
reference_name: string
*full_name: string*
*contact: <author object>*
*phone_number: string*
*email: string*
*website: string*
```

revision:

```
date: string
authors: list of <author object>
```

stationxml_comment:

```
value: string
*begin_effective_time: string, date formatted*
*end_effective_time: string, date formatted*
*authors: list of <author>*
```

6.1.5 3) Full level diagram

6.1.6 Structural units

A full obsinfo network description consists of the following fields (starred fields are optional):

```
format_version: {}
*revision: {}*
*notes: []*
network:
  operator: {}
  *stations_operator: {}*
  *restricted_state: {}*
  *comments: []*
  *extras: {}*
  campaign_ref_name: string
  network_info: {}
  stations:
    <STATIONNAME1>:
      site: string
      start_date: string
      end_date: string
      location_code: string
      *serial_number: string*
      *operator: {}*
      instrumentation:
        *operator: {}*
```

(continues on next page)

(continued from previous page)

```

equipment: {}
channels:
  default:
    *orientation_code: string or {}*
    datalogger:
      << GENERIC_COMPONENT
      sample_rate: number
      *delay_correction: number*
    *preamplifier:*
      *<< GENERIC_COMPONENT*
    sensor:
      << GENERIC_COMPONENT
      seed_codes:
    *preamplifier_configuration: string*
    *sensor_configuration: string*
    *datalogger_configuration: string*
    *location_code: string* # otherwise inherits from station
    *comments: []*
    *extras: {}*
    <SPECIFIC-CHANNEL1>: {}
    <SPECIFIC-CHANNEL2>: {}
    ...
  *channel_modifications: {}*
locations: {}
*notes: []*
*comments: []*
*extras: {}*
*processing:*
  - *clock_correction_linear: {}*
  - *clock_correction_leapsecond: {}*
<STATIONNAME2>:
  ...

```

Where GENERIC_COMPONENT is:

```

equipment:
  <equipment>
  *configuration_default: string*
  *configuration_definitions: {}*
  *response_modifications: {}*
  *notes: []*
  *response_stages:*
    - stage:
      input_units: string
      output_units string
      gain: float
      *name: string*
      *description: string*
      *decimation_factor: integer*
      *delay: number*
      *calibration_date: string*
      *polarity: '+' or '-'* # default is '+'

```

(continues on next page)

(continued from previous page)

```
*input_sample_rate: number*
*filter:*
  <filter>
- stage:
- ...
```

6.2 Basic - two channel

Here is an example of a valid information file containing only required fields, all in one file. To save space, this is a two-channel instrument. Generally, to save space and avoid repetition, information files are split into parts connected by JSON references: you can find the exact same information as below, but spit into “atomic” files, here.

The file below can be found in `obsinfo/_examples/examples/BASIC_FLAT.network.yaml`

6.3 Basic atomic - two channel

Here is an example of a the same information as in `basic_flat`, divided across the standard obsinfo file structure. You can see that, despite the existence of headers in each file, there are many less lines and less repetition.

The network below is found in `obsinfo/_examples/examples/BASIC_ATOMIC.network.yaml` and the other information files are found in the `obsinfo/_examples` hierarchy, although any configuration definitions have been removed here for simplicity. To see what configuration definitions can do, see the `basic_configuration` document.

6.4 Basic atomic + configuration - two channel

Using the same basic instrumentation as above, here is an example of configuration, which allows us to:

- change instrument components from the default (sensors, preamplifiers, dataloggers)
- modify parameters of one or more instrument components (serial numbers, response stage, datalogger sampling frequency and/or digital filter)

The network below is found in `obsinfo/_examples/examples/BASIC_CONFIGURATION.network.yaml` and the other information files are found in the `obsinfo/_examples` hierarchy

6.5 Comparison with StationXML

obsinfo information fields are as close to StationXML as feasible, but the need to reduce duplicated information requires some changes.

6.5.1 Summary of differences

YAML_ instead of XML

Files are easier to read (as long as they are kept small) and we can take advantage of the **JSONref_** standard for importing files

arrays instead of multiply-used fields:

XML allows the same field name to be used repeatedly, **YAML_** does not. So multiply-used fields in **StationXML** are replaced with arrays, usually with an “s” tacked onto the field name. Some examples (noted as they are in the following tables) are:

StationXML field	obsinfo array
Comment: (0+)	comments: []
CalibrationDate: (0+)	calibration_dates: []

Use of \$ref s to insert other files

Inherited from **JSONref_** and is the key to reducing information to atomic components

Use of *base* channels and specific modifiers

Many channels have much the same information (position, start and enddates, even sensors). obsinfo therefore allows a *base* channel definition, which is then modified by values placed in the specific modifiers fields

Default inheritance of some fields

In many cases, the *operator*, *location_code*, *start_date* and *end_date* values are the same at the *network*, *station* and *channel* levels. In StationXML they must be redefined in each level. In obsinfo, if they are not defined at a level, they are inherited from the level above.

No *sensitivity* stage in obsinfo

The StationXML sensitivity stage is supposed to correspond to the sum of the sensitivities of the underlying stages. *obsinfo* therefore simply calculates this value from the provided stages

Handling of InstrumentComponents

StationXML Channel/Datalogger, **Channel/Sensor** and **Channel/Preamplifier** elements are actually **Equipment** objects. These elements' response stages are mixed into the **Channel/Stages** list.

obsinfo_ datalogger, sensor and (optional) preamplifier elements contain equipment definitions, response_stage lists and elements specific to each InstrumentComponent (sensor : seed_code, azimuth, dip. datalogger: sampling_rate) which are given flatly in the **StationXML Channel** level

The **StationXML InstrumentSensitivity** stage should equal the sum of the sensitivities of the underlying stages at the given frequency. **obsinfo_** therefore does not ask for this value, but calculates it from the provided stages

Handling of positions

StationXML 1.2 specifies Station and Channel positions using the elements **Latitude**, **Longitude** and **Elevation**, each of which is based on the **FloatType_** which includes **units**, **plusError**, **minusError** and **measurementMethod**

Latitudes and Longitudes are generally given in degrees and Elevation in meters, but most most OBS (and land station) positions have an uncertainty that is approximately constant (in meters), depending on the measurementMethod.

For this reason, **obsinfo_** expresses positions and their uncertainties using three elements: - position: {lat:, lon:, elev.m:} - position_uncertainty: {lat.m, lon.m, elev.m} - position_measurement_method: string

This allows the position_uncertainty values to be associated with a measurement method and entered separately from the actual instrument position. **obsinfo_** then translates these values into StationXML FloatTypes with the appropriate units arc-degrees, arc-degrees and meters), with the arc-degrees value depending on the station latitude.

6.5.2 Line-by-line comparison of differences

Below is a line by line naming of StationXML 1.2 fields and their equivalent name in obsinfo information files:

StationXML <i>FDSNStationXML</i>	obsinfo
Source	<i>None</i>
Sender	<i>None</i>
Module	<i>None</i>
ModuleURI	<i>None</i>
Created	<i>Automatically calculated</i>
Network	<i>See Network_level</i>

StationXML Network	obsinfo network
code (attribute)	code
startDate (attribute)	start_date:
endDate (attribute)	end_date:
sourceID (attribute)	<i>None (should automatically generate?)</i>
restrictedStatus (attribute)	<i>embargo.a???</i>
alternateCode (attribute)	<i>None</i>
historicalCode (attribute)	<i>None</i>
Description	description
Comment (0+)	comments: []
Operator	operator
Identifier	<i>None</i>
*DataAvailability	<i>None</i>
SelectedNumberStations	<i>N/A</i>
Station (0+)	stations <i>see station_level</i>

StationXML Station	obsinfo station	Notes
Description		
Identifier		
DataAvailability		
Latitude	position: {lat:, lon:, elev:}	
Longitude		
Elevation		
Site:	site (only fills "Name")	
WaterLevel	<i>(None)</i>	
Vault	vault	
Equipment:	equipment (see equipment_)	
Operator:	agency (see agency_)	change in obsinfo in v0.111)
TerminationDate	<i>None</i>	(likely to be removed from StationXML)
SelectedNumberChannels	<i>None</i>	(likely to be removed from StationXML)
TotalNumberChannels	<i>None</i>	(likely to be removed from StationXML)
ExternalReference: []	<i>None</i>	
Channel: (0+)	channels: (see channels_)	

StationXML Equipment	obsinfo equipment
Description	description
Type	type
Manufacturer	manufacturer
Model	model
Vendor	vendor
SerialNumber	serial_number
InstallationDate	installation_date
RemovalDate	removal_date
CalibrationDate: (0+)	calibration_dates: []

The [StationXML Response](#) level is replaced in **obsinfo_** by the datalogger preamplifier and sensor levels

StationXML <i>Response</i>	obsinfo channel	Notes
resourceId (<i>attribute</i>)		
InstrumentSensitivity	<i>calculated from Stages</i>	
<i>InstrumentPolynomial</i>	<i>None</i>	StationXML allows either InstrumentSensitivity or InstrumentPolynomial
Stage (0+)	datalogger_ preamplifier_ sensor_	

StationXML <i>Stage</i>	obsinfo stage	Notes
number (attribute)	<i>calculated by obsinfo</i>	
resourceId (attribute)		
{Type}	filter/type:	Can be PolesZeros, Coefficients, ResponseList, FIR or Polynomial
Decimation		<i>DecimationType_</i> , Not in “Polynomials”
StageGain		<i>GainType</i> : Value, Frequency. Not in “Polynomials”

StationXML <i>DecimationType</i>	obsinfo stage	Notes
InputSampleRate		
Factor		
Offset		
Delay		
Correction		

StationXML <i>PolesZeros</i>	obsinfo filter	Notes
name (attribute)		
resourceId (<i>attribute</i>)		
<i>Description</i>		
InputUnits		
OutputUnits		
PzTransferFunctionType		
NormalizationFactor		
NormalizationFrequency		
Zero: (0+)		
Pole: (0+)		

StationXML <i>Coefficients</i>	obsinfo filter	Notes
name (attribute)		
resourceId (<i>attribute</i>)		
<i>Description</i>		
InputUnits		
OutputUnits		
CfTransferFunctionType		
Numerator: (0+)		
Denominator: (0+)		

StationXML <i>ResponseList</i>	obsinfo filter	Notes
name (attribute)		
resourceId (attribute)		
Description		
InputUnits		
OutputUnits		
ResponseListElement: (0+)		StationXML: Frequency, Amplitude, Phase

StationXML <i>FIR</i>	obsinfo filter	Notes
name (attribute)		
resourceId (attribute)		
Description		
InputUnits		
OutputUnits		
Symmetry		NONE, EVEN or ODD
NumeratorCoefficient: (0+)		

StationXML <i>Polynomial</i>	obsinfo filter	Notes
name (attribute)		
resourceId (attribute)		
Description		
InputUnits		
OutputUnits		
ApproximationType		default="MACLAURIN">
FrequencyLowerBound		
FrequencyUpperBound		
ApproximationLowerBound		
ApproximationUpperBound		
MaximumError		
Coefficient: (0+)		

StationXML <i>Operator</i>	obsinfo agency
Agency: string	
Contact: []	
WebSite: URI	

StationXML <i>PersonType</i>	obsinfo author
Name: (0+)	
Agency: (0+)	
Email: (0+)	
Phone: (0+)	PhoneNumberType_

StationXML <i>PhoneNumberType</i>	obsinfo
CountryCode	
AreaCode	
PhoneNumber	
name	

StationXML <i>uncertaintyDouble</i>	obsinfo
<i>plusError</i> (attribute)	
<i>minusError</i> (attribute)	
<i>measurementMethod</i> (attribute)	

```

FDSNStationXML:
  Network:
    Station:
      Channel:
        - *Description: string*
        *Identifier:*
          - string
        *Comment:*
          - Value: string
            BeginEffectiveTime: '2017-08-25T05:02:50.47'
            EndEffectiveTime: '2016-03-07T01:49:19.16'
            Author: ''
              - ''
          - Value: string
        *DataAvailability:*
          Extent: ''
          Span:
            - ''
      ExternalReference:
        URI: <URI>
        Description: string
      Latitude: number
      Longitude: number
      Elevation: number
      Depth: number
      *Azimuth: number*
      Dip: number
      Type:
        - [CONTINUOUS, HEALTH, SYNTHESIZED, GEOPHYSICAL, TRIGGERED, or WEATHER]
      SampleRate: number
      SampleRateRatio:
        NumberSamples: number
        NumberSeconds: number
      ClockDrift: number
      *CalibrationUnits:*
        Name: string
        Description: string
      Sensor:
        <equipment>
      *PreAmplifier:*
        <equipment>
      DataLogger:
        <equipment>
      Equipment:
        <equipment>
      Response:
        AnyElementYouLike: Some Data Or Other Elements

```

(continues on next page)

(continued from previous page)

```

- Identifier: string
- ''
- ''

```

```

<DecimationType>
  InputSampleRate
  Factor
  Offset
  Delay
  Correction

```

```

<BaseNodeType>
  code *(attribute)*
  *startDate *(attribute)*
  *endDate* *(attribute)*
  *sourceID* *(attribute)*
  *restrictedStatus* *(attribute)*
  *alternateCode* *(attribute)*
  *historicalCode* *(attribute)*
  *Description:*
  *Identifier (0+)*
  *Comment (0+)*
  *DataAvailability*

```

```
.. code-block:: yaml
```

```

<BaseFilterType>
  name (attribute)
  resourceId (*attribute*)
  *Description*
  InputUnits
  OutputUnits

```

stationxml 1.2 schema, with <annotations> and their contained <documentation>s removed

```

<xs:element name="FDSNStationXML" type="fsx:RootType"/>
<xs:complexType name="RootType">
  <xs:sequence>
    <xs:element name="Source" type="xs:string">
      <xs:element name="Sender" type="xs:string" minOccurs="0">
        <xs:element name="Module" type="xs:string" minOccurs="0">
          <xs:element name="ModuleURI" type="xs:anyURI" minOccurs="0">
            <xs:element name="Created" type="xs:dateTime">
              <xs:element name="Network" type="fsx:NetworkType" maxOccurs=
↳ "unbounded"/>
            <xs:any namespace="##other" processContents="lax" minOccurs="0"
↳ maxOccurs="unbounded"/>
          </xs:sequence>
          <xs:attribute name="schemaVersion" type="xs:decimal" use="required">
            <xs:anyAttribute namespace="##other" processContents="lax"/>
          </xs:complexType>
        <xs:complexType name="NetworkType">

```

(continues on next page)

(continued from previous page)

```

        <xs:complexContent>
          <xs:extension base="fsx:BaseNodeType">
            <xs:sequence>
              <xs:element name="Operator" type="fsx:OperatorType"
↳minOccurs="0"
                                maxOccurs="unbounded">
              <xs:element name="TotalNumberStations" type=
↳"fsx:CounterType" minOccurs="0">
              <xs:element name="SelectedNumberStations" type=
↳"fsx:CounterType" minOccurs="0">
              <xs:element name="Station" type="fsx:StationType"
↳minOccurs="0"
                                maxOccurs="unbounded"/>
            </xs:sequence>
          </xs:extension>
        </xs:complexContent>
      </xs:complexType>
    <xs:complexType name="StationType">
      <xs:complexContent>
        <xs:extension base="fsx:BaseNodeType">
          <xs:sequence>
            <xs:element name="Latitude" type="fsx:LatitudeType">
            <xs:element name="Longitude" type="fsx:LongitudeType">
            <xs:element name="Elevation" type="fsx:DistanceType">
            <xs:element name="Site" type="fsx:SiteType">
            <xs:element name="WaterLevel" type="fsx:FloatType" minOccurs="0">
            <xs:element name="Vault" type="xs:string" minOccurs="0">
            <xs:element name="Geology" type="xs:string" minOccurs="0">
            <xs:element name="Equipment" type="fsx:EquipmentType" minOccurs="0"
              maxOccurs="unbounded">
            </xs:element>
            <xs:element name="Operator" type="fsx:OperatorType" minOccurs="0"
              maxOccurs="unbounded">
            </xs:element>
            <xs:element name="CreationDate" type="xs:dateTime" minOccurs="0">
            <xs:element name="TerminationDate" type="xs:dateTime" minOccurs="0">
            <xs:element name="TotalNumberChannels" type="fsx:CounterType" minOccurs=
↳"0">
            <xs:element name="SelectedNumberChannels" type="fsx:CounterType"
↳minOccurs="0">
            <xs:element name="ExternalReference" type="fsx:ExternalReferenceType"
              minOccurs="0" maxOccurs="unbounded">
            <xs:element name="Channel" type="fsx:ChannelType" minOccurs="0"
              maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
    <!-- End StationType-->
    <xs:complexType name="ChannelType">
      <xs:complexContent>
        <xs:extension base="fsx:BaseNodeType">

```

(continues on next page)

(continued from previous page)

```

<xs:sequence>
  <xs:element name="ExternalReference" type="fsx:ExternalReferenceType"
    minOccurs="0" maxOccurs="unbounded">
  </xs:element>
  <xs:element name="Latitude" type="fsx:LatitudeType">
  <xs:element name="Longitude" type="fsx:LongitudeType">
  <xs:element name="Elevation" type="fsx:DistanceType">
  <xs:element name="Depth" type="fsx:DistanceType">
  <xs:element name="Azimuth" type="fsx:AzimuthType" minOccurs="0">
  <xs:element name="Dip" type="fsx:DipType" minOccurs="0">
  <xs:element name="WaterLevel" type="fsx:FloatType" minOccurs="0">
  <xs:element name="Type" minOccurs="0" maxOccurs="unbounded">
    <xs:simpleType>
      <xs:restriction base="xs:NMTOKEN">
        <xs:enumeration value="TRIGGERED"/>
        <xs:enumeration value="CONTINUOUS"/>
        <xs:enumeration value="HEALTH"/>
        <xs:enumeration value="GEOPHYSICAL"/>
        <xs:enumeration value="WEATHER"/>
        <xs:enumeration value="FLAG"/>
        <xs:enumeration value="SYNTHESIZED"/>
        <xs:enumeration value="INPUT"/>
        <xs:enumeration value="EXPERIMENTAL"/>
        <xs:enumeration value="MAINTENANCE"/>
        <xs:enumeration value="BEAM"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
  <xs:group ref="fsx:SampleRateGroup" minOccurs="0"/>
  <xs:element name="ClockDrift" minOccurs="0">
    <xs:complexType>
      <xs:simpleContent>
        <xs:restriction base="fsx:FloatType">
          <xs:minInclusive value="0"/>
          <xs:attribute name="unit" type="xs:string" use="optional"
→ " fixed="SECONDS/SAMPLE">
          </xs:attribute>
        </xs:restriction>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="CalibrationUnits" type="fsx:UnitsType" minOccurs="0">
  <xs:element name="Sensor" type="fsx:EquipmentType" minOccurs="0">
  <xs:element name="PreAmplifier" type="fsx:EquipmentType" minOccurs="0">
  <xs:element name="DataLogger" type="fsx:EquipmentType" minOccurs="0">
  <xs:element name="Equipment" type="fsx:EquipmentType" minOccurs="0"
→ maxOccurs="unbounded">
  <xs:element name="Response" type="fsx:ResponseType" minOccurs="0"/>
</xs:sequence>
  <xs:attribute name="locationCode" type="xs:string" use="required">
</xs:extension>
</xs:complexContent>

```

(continues on next page)

(continued from previous page)

```

</xs:complexType>
<!-- End ChannelType -->
<xs:complexType name="GainType">
  <xs:sequence>
    <xs:element name="Value" type="xs:double">
    <xs:element name="Frequency" type="xs:double">
  </xs:sequence>
</xs:complexType>
<xs:group name="FrequencyRangeGroup">
  <xs:sequence>
    <xs:element name="FrequencyStart" type="xs:double">
    <xs:element name="FrequencyEnd" type="xs:double">
    <xs:element name="FrequencyDBVariation" type="xs:double">
  </xs:sequence>
</xs:group>
<xs:complexType name="SensitivityType">
  <xs:complexContent>
    <xs:extension base="fsx:GainType">
      <xs:sequence>
        <xs:element name="InputUnits" type="fsx:UnitsType">
        <xs:element name="OutputUnits" type="fsx:UnitsType">
        <xs:group ref="fsx:FrequencyRangeGroup" minOccurs="0">
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="EquipmentType">
  <xs:sequence>
    <xs:element name="Type" type="xs:string" minOccurs="0">
    <xs:element name="Description" type="xs:string" minOccurs="0">
    <xs:element name="Manufacturer" type="xs:string" minOccurs="0">
    <xs:element name="Vendor" type="xs:string" minOccurs="0">
    <xs:element name="Model" type="xs:string" minOccurs="0">
    <xs:element name="SerialNumber" type="xs:string" minOccurs="0">
    <xs:element name="InstallationDate" type="xs:dateTime" minOccurs="0">
    <xs:element name="RemovalDate" type="xs:dateTime" minOccurs="0">
    <xs:element name="CalibrationDate" type="xs:dateTime" minOccurs="0" maxOccurs=
    ↪ "unbounded">
    <xs:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs=
    ↪ "unbounded"/>
  </xs:sequence>
  <xs:attribute name="resourceId" type="xs:string" use="optional">
  </xs:attribute>
  <xs:anyAttribute namespace="##other" processContents="lax"/>
</xs:complexType>
<xs:complexType name="ResponseStageType">
  <xs:sequence>
    <xs:choice>
      <xs:sequence>
        <xs:choice>
          <xs:element name="PolesZeros" type="fsx:PolesZerosType" minOccurs="0
          ↪ ">

```

(continues on next page)

(continued from previous page)

```

    <xs:element name="Coefficients" type="fsx:CoefficientsType"
    ↪ minOccurs="0"/>
    <xs:element name="ResponseList" type="fsx:ResponseListType"
    ↪ minOccurs="0"/>
    <xs:element name="FIR" type="fsx:FIRType" minOccurs="0">
    </xs:choice>
    <xs:element name="Decimation" type="fsx:DecimationType" minOccurs="0"/>
    <xs:element name="StageGain" type="fsx:GainType">
    </xs:sequence>
    <xs:element name="Polynomial" type="fsx:PolynomialType">
    </xs:choice>
    <xs:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs=
    ↪ "unbounded"/>
    </xs:sequence>
    <xs:attribute name="number" type="fsx:CounterType" use="required">
    <xs:attribute name="resourceId" type="xs:string">
    <xs:anyAttribute namespace="##other" processContents="lax"/>
</xs:complexType>
<xs:complexType name="CommentType">
    <xs:sequence>
    <xs:element name="Value" type="xs:string">
    <xs:element name="BeginEffectiveTime" type="xs:dateTime" minOccurs="0">
    <xs:element name="EndEffectiveTime" type="xs:dateTime" minOccurs="0">
    <xs:element name="Author" type="fsx:PersonType" minOccurs="0" maxOccurs=
    ↪ "unbounded">
    </xs:sequence>
    <xs:attribute name="id" type="fsx:CounterType" use="optional">
    <xs:attribute name="subject" type="xs:string" use="optional">
</xs:complexType>
<xs:complexType name="PolesZerosType">
    <xs:complexContent>
    <xs:extension base="fsx:BaseFilterType">
    <xs:sequence>
    <xs:element name="PzTransferFunctionType">
    <xs:simpleType>
    <xs:restriction base="xs:string">
    <xs:enumeration value="LAPLACE (RADIANS/SECOND)"/>
    <xs:enumeration value="LAPLACE (HERTZ)"/>
    <xs:enumeration value="DIGITAL (Z-TRANSFORM)"/>
    </xs:restriction>
    </xs:simpleType>
    </xs:element>
    <xs:element name="NormalizationFactor" type="xs:double" default="1.0">
    <xs:element name="NormalizationFrequency" type="fsx:FrequencyType">
    <xs:element name="Zero" type="fsx:PoleZeroType" minOccurs="0" maxOccurs=
    ↪ "unbounded">
    <xs:element name="Pole" type="fsx:PoleZeroType" minOccurs="0" maxOccurs=
    ↪ "unbounded">
    </xs:sequence>
    </xs:extension>
    </xs:complexContent>
</xs:complexType>

```

(continues on next page)

(continued from previous page)

```

<xs:complexType name="FIRType">
  <xs:complexContent>
    <xs:extension base="fsx:BaseFilterType">
      <xs:sequence>
        <xs:element name="Symmetry">
          <xs:simpleType>
            <xs:restriction base="xs:NMTOKEN">
              <xs:enumeration value="NONE"/>
              <xs:enumeration value="EVEN"/>
              <xs:enumeration value="ODD"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
        <xs:element name="NumeratorCoefficient" minOccurs="0" maxOccurs=
↪ "unbounded">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:double">
                <xs:attribute name="i" type="xs:integer"/>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="CoefficientsType">
  <xs:complexContent>
    <xs:extension base="fsx:BaseFilterType">
      <xs:sequence>
        <xs:element name="CfTransferFunctionType">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="ANALOG (RADIANS/SECOND)"/>
              <xs:enumeration value="ANALOG (HERTZ)"/>
              <xs:enumeration value="DIGITAL"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
        <xs:element name="Numerator" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="fsx:FloatNoUnitType">
                <xs:attribute name="number" type="fsx:CounterType"/>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
        <xs:element name="Denominator" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:simpleContent>

```

(continues on next page)

(continued from previous page)

```

        <xs:extension base="fsx:FloatNoUnitType">
            <xs:attribute name="number" type="fsx:CounterType"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:extension>
</xs:complexContent>
</xs:complexType>
<xs:complexType name="ResponseListElementType">
    <xs:sequence>
        <xs:element name="Frequency" type="fsx:FrequencyType"/>
        <xs:element name="Amplitude" type="fsx:FloatType"/>
        <xs:element name="Phase" type="fsx:AngleType"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="ResponseListType">
    <xs:complexContent>
        <xs:extension base="fsx:BaseFilterType">
            <xs:sequence>
                <xs:element name="ResponseListElement" type="fsx:ResponseListElementType"
                    minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="PolynomialType">
    <xs:complexContent>
        <xs:extension base="fsx:BaseFilterType">
            <xs:sequence>
                <xs:element name="ApproximationType" default="MACLAURIN">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:enumeration value="MACLAURIN"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:element>
                <xs:element name="FrequencyLowerBound" type="fsx:FrequencyType">
                <xs:element name="FrequencyUpperBound" type="fsx:FrequencyType">
                <xs:element name="ApproximationLowerBound" type="xs:double">
                <xs:element name="ApproximationUpperBound" type="xs:double">
                <xs:element name="MaximumError" type="xs:double">
                <xs:element name="Coefficient" maxOccurs="unbounded">
                    <xs:complexType>
                        <xs:simpleContent>
                            <xs:extension base="fsx:FloatNoUnitType">
                                <xs:attribute name="number" type="fsx:CounterType"/>
                            </xs:extension>
                        </xs:simpleContent>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>

```

(continues on next page)

(continued from previous page)

```

        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="DecimationType">
    <xs:sequence>
      <xs:element name="InputSampleRate" type="fsx:FrequencyType"/>
      <xs:element name="Factor" type="xs:integer">
      <xs:element name="Offset" type="xs:integer">
      <xs:element name="Delay" type="fsx:FloatType">
      <xs:element name="Correction" type="fsx:FloatType">
    </xs:sequence>
  </xs:complexType>
  <!-- The following elements represent numbers. -->
  <xs:attributeGroup name="uncertaintyDouble">
    <xs:attribute name="plusError" type="xs:double" use="optional">
    <xs:attribute name="minusError" type="xs:double" use="optional">
    <xs:attribute name="measurementMethod" type="xs:string" use="optional"/>
  </xs:attributeGroup>
  <xs:complexType name="FloatNoUnitType">
    <xs:simpleContent>
      <xs:extension base="xs:double">
        <xs:attributeGroup ref="fsx:uncertaintyDouble"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="FloatType">
    <xs:simpleContent>
      <xs:extension base="xs:double">
        <xs:attribute name="unit" type="xs:string" use="optional">
        <xs:attributeGroup ref="fsx:uncertaintyDouble"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <!-- Derived from FloatType. -->
  <xs:complexType name="SecondType">
    <xs:simpleContent>
      <xs:restriction base="fsx:FloatType">
        <xs:attribute name="unit" type="xs:string" fixed="SECONDS">
        <xs:attributeGroup ref="fsx:uncertaintyDouble"/>
      </xs:restriction>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="VoltageType">
    <xs:simpleContent>
      <xs:restriction base="fsx:FloatType">
        <xs:attribute name="unit" type="xs:string" fixed="VOLTS">
        <xs:attributeGroup ref="fsx:uncertaintyDouble"/>
      </xs:restriction>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="AngleType">

```

(continues on next page)

(continued from previous page)

```

<xs:simpleContent>
  <xs:restriction base="fsx:FloatType">
    <xs:minInclusive value="-360"/>
    <xs:maxInclusive value="360"/>
    <xs:attribute name="unit" type="xs:string" use="optional" fixed="DEGREES">
    <xs:attributeGroup ref="fsx:uncertaintyDouble"/>
  </xs:restriction>
</xs:simpleContent>
</xs:complexType>
<xs:complexType name="LatitudeBaseType">
  <xs:simpleContent>
    <xs:restriction base="fsx:FloatType">
      <xs:minInclusive value="-90"/>
      <xs:maxExclusive value="90"/>
      <xs:attribute name="unit" type="xs:string" use="optional" fixed="DEGREES">
      <xs:attributeGroup ref="fsx:uncertaintyDouble"/>
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="LatitudeType">
  <xs:simpleContent>
    <xs:extension base="fsx:LatitudeBaseType">
      <xs:attribute name="datum" type="xs:NMTOKEN" use="optional" default="WGS84"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="LongitudeBaseType">
  <xs:simpleContent>
    <xs:restriction base="fsx:FloatType">
      <xs:minInclusive value="-180"/>
      <xs:maxInclusive value="180"/>
      <xs:attribute name="unit" type="xs:string" use="optional" fixed="DEGREES">
      <xs:attributeGroup ref="fsx:uncertaintyDouble"/>
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="LongitudeType">
  <xs:simpleContent>
    <xs:extension base="fsx:LongitudeBaseType">
      <xs:attribute name="datum" type="xs:NMTOKEN" use="optional" default="WGS84"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="AzimuthType">
  <xs:simpleContent>
    <xs:restriction base="fsx:FloatType">
      <xs:minInclusive value="0"/>
      <xs:maxExclusive value="360"/>
      <xs:attribute name="unit" type="xs:string" use="optional" fixed="DEGREES">
      <xs:attributeGroup ref="fsx:uncertaintyDouble"/>
    </xs:restriction>
  </xs:simpleContent>

```

(continues on next page)

(continued from previous page)

```

</xs:complexType>
<xs:complexType name="DipType">
  <xs:simpleContent>
    <xs:restriction base="fsx:FloatType">
      <xs:minInclusive value="-90"/>
      <xs:maxInclusive value="90"/>
      <xs:attribute name="unit" type="xs:string" use="optional" fixed="DEGREES">
      <xs:attributeGroup ref="fsx:uncertaintyDouble"/>
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="DistanceType">
  <xs:simpleContent>
    <xs:restriction base="fsx:FloatType">
      <xs:attribute name="unit" type="xs:string" use="optional" default="METERS">
      <xs:attributeGroup ref="fsx:uncertaintyDouble"/>
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="FrequencyType">
  <xs:simpleContent>
    <xs:restriction base="fsx:FloatType">
      <xs:attribute name="unit" type="xs:string" use="optional" fixed="HERTZ">
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>
<xs:group name="SampleRateGroup">
  <xs:sequence>
    <xs:element name="SampleRate" type="fsx:SampleRateType">
    <xs:element name="SampleRateRatio" type="fsx:SampleRateRatioType" minOccurs="0">
  </xs:sequence>
</xs:group>
<xs:complexType name="SampleRateType">
  <xs:simpleContent>
    <xs:restriction base="fsx:FloatType">
      <xs:attribute name="unit" type="xs:string" use="optional" fixed="SAMPLES/S">
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="SampleRateRatioType">
  <xs:sequence>
    <xs:element name="NumberSamples" type="xs:integer">
    <xs:element name="NumberSeconds" type="xs:integer">
  </xs:sequence>
</xs:complexType>
<xs:complexType name="PoleZeroType">
  <xs:sequence>
    <xs:element name="Real" type="fsx:FloatNoUnitType">
    <xs:element name="Imaginary" type="fsx:FloatNoUnitType">
  </xs:sequence>
  <xs:attribute name="number" type="xs:integer">
</xs:complexType>

```

(continues on next page)

(continued from previous page)

```

<xs:simpleType name="CounterType">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="0"/>
  </xs:restriction>
</xs:simpleType>
<xs:complexType name="OperatorType">
  <xs:sequence>
    <xs:element name="Agency" type="xs:string">
    <xs:element name="Contact" type="fsx:PersonType" minOccurs="0" maxOccurs=
    ↪ "unbounded"/>
    <xs:element name="WebSite" type="xs:anyURI" minOccurs="0">
  </xs:sequence>
</xs:complexType>
<xs:complexType name="PersonType">
  <xs:sequence>
    <xs:element name="Name" type="xs:string" minOccurs="0" maxOccurs="unbounded">
    <xs:element name="Agency" type="xs:string" minOccurs="0" maxOccurs="unbounded">
    <xs:element name="Email" type="fsx:EmailType" minOccurs="0" maxOccurs="unbounded
    ↪ ">
    <xs:element name="Phone" type="fsx:PhoneNumberType" minOccurs="0" maxOccurs=
    ↪ "unbounded">
  </xs:sequence>
</xs:complexType>
<xs:complexType name="SiteType">
  <xs:sequence>
    <xs:element name="Name" type="xs:string">
    <xs:element name="Description" type="xs:string" minOccurs="0">
    <xs:element name="Town" type="xs:string" minOccurs="0">
    <xs:element name="County" type="xs:string" minOccurs="0">
    <xs:element name="Region" type="xs:string" minOccurs="0">
    <xs:element name="Country" type="xs:string" minOccurs="0">
    <xs:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs=
    ↪ "unbounded"/>
  </xs:sequence>
  <xs:anyAttribute namespace="##other" processContents="lax"/>
</xs:complexType>
<xs:complexType name="ExternalReferenceType">
  <xs:sequence>
    <xs:element name="URI" type="xs:anyURI">
    <xs:element name="Description" type="xs:string">
  </xs:sequence>
</xs:complexType>
<!-- Simple types -->
<xs:simpleType name="NominalType">
  <xs:restriction base="xs:NMTOKEN">
    <xs:enumeration value="NOMINAL"/>
    <xs:enumeration value="CALCULATED"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="EmailType">
  <xs:restriction base="xs:string">
    <xs:pattern value="[\w\.\-_\+][\w\.\-_\+]" />

```

(continues on next page)

(continued from previous page)

```

    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="PhoneNumberType">
    <xs:sequence>
      <xs:element name="CountryCode" type="xs:integer" minOccurs="0">
      <xs:element name="AreaCode" type="xs:integer">
      <xs:element name="PhoneNumber">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:pattern value="[0-9]+-[0-9]+"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="description" type="xs:string" use="optional"/>
  </xs:complexType>
  <xs:simpleType name="RestrictedStatusType">
    <xs:restriction base="xs:NMTOKEN">
      <xs:enumeration value="open"/>
      <xs:enumeration value="closed"/>
      <xs:enumeration value="partial"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="UnitsType">
    <xs:sequence>
      <xs:element name="Name" type="xs:string">
      <xs:element name="Description" type="xs:string" minOccurs="0">
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="IdentifierType">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="type" type="xs:string">
        </xs:attribute>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="BaseFilterType">
    <xs:sequence>
      <xs:element name="Description" type="xs:string" minOccurs="0">
      <xs:element name="InputUnits" type="fsx:UnitsType">
      <xs:element name="OutputUnits" type="fsx:UnitsType">
      <xs:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs=
↪ "unbounded"/>
    </xs:sequence>
    <xs:attribute name="resourceId" type="xs:string">
    <xs:attribute name="name" type="xs:string">
    <xs:anyAttribute namespace="##other" processContents="lax"/>
  </xs:complexType>
  <xs:complexType name="ResponseType">
    <xs:sequence>
      <xs:choice minOccurs="0">

```

(continues on next page)

(continued from previous page)

```

        <xs:element name="InstrumentSensitivity" type="fsx:SensitivityType"
↪ minOccurs="0">
        <xs:element name="InstrumentPolynomial" type="fsx:PolynomialType" minOccurs=
↪ "0">
        </xs:choice>
        <xs:element name="Stage" type="fsx:ResponseStageType" minOccurs="0"
            maxOccurs="unbounded">
        </xs:element>
        <xs:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs=
↪ "unbounded"/>
        </xs:sequence>
        <xs:attribute name="resourceId" type="xs:string">
        <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:complexType>
    <xs:complexType name="DataAvailabilityExtentType">
        <xs:attribute name="start" type="xs:dateTime" use="required">
        <xs:attribute name="end" type="xs:dateTime" use="required">
        <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:complexType>
    <xs:complexType name="DataAvailabilitySpanType">
        <xs:attribute name="start" type="xs:dateTime" use="required">
        <xs:attribute name="end" type="xs:dateTime" use="required">
        <xs:attribute name="numberSegments" type="xs:integer" use="required">
        <xs:attribute name="maximumTimeTear" type="xs:decimal" use="optional">
        <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:complexType>
    <xs:complexType name="DataAvailabilityType">
        <xs:sequence>
            <xs:element name="Extent" type="fsx:DataAvailabilityExtentType" minOccurs="0"/>
            <xs:element name="Span" type="fsx:DataAvailabilitySpanType" minOccurs="0"
↪ maxOccurs="unbounded"/>
            <xs:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs=
↪ "unbounded"/>
        </xs:sequence>
        <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:complexType>

```

6.6 Comparison with AROL/YASMINE

The goals and technologies of obsinfo are similar to those found in YASMINE and the format of the Atomic Response Object Library (AROL) files used by YASMINE is the very similar to that of the Datalogger, Sensor and Preamplifier files (and underlying filter files) used by obsinfo. The major differences are: - obsinfo is a completely file-based solution, designed for processing large

sets of similar instruments. Yasmine is a GUI (YASMINE_EDITOR) or a command-line interface (YASMINE_CLI) to modify existing StationXML files. obsinfo is thus best compared with a combination of YASMINE_EDITOR (to create a base StationXML file for a given configuration), followed by YASMINE_CLI (to enter station-specific modifications to this configuration)

- obsinfo holds its configuration information inside the instrument component (DataLogger, PreAmplifier, Sensor) files, whereas AROL has a file for each configuration and uses an accompanying configuration file to choose between them

- obsinfo adds processing and ocean-bottom seismology specific fields, to facilitate the notation and correction for clock problems such as drift and leap-seconds.
- obsinfo uses `instrumentation`` files to represent specific OBSs
- obsinfo uses `network` files to control all this

The YASMINE Command-line interface (`_YASMINE_CLI`)

6.6.1 Nomenclature

We use the term “InstrumentComponent” for Dataloggers, Preamplifiers and Sensors.

6.6.2 AROL versus obsinfo configuration methods

Let’s compare the AROL implementation of a Nanometrics Taurus datalogger with that of obsinfo. My AROL example comes from `gitlab/RESIF/AROL/sources/Dataloggers`.

The AROL configuration file is called `nanometrics.yaml` and includes Centaur, Taurus and .. dataloggers. For brevity, I only include the Taurus loggers:

```
mandatory_filters: [digitizer_manufacturer, digitizer_model, sampling_rate]
```

filters:

- **name:** Digitizer manufacturer
code: digitizer_manufacturer
help: Select the manufacturer of your digitizer
- **name:** Digitizer model
code: digitizer_model
help: Select the model of your digitizer
- **name:** Version
code: digitizer_version
help: Select the version of digitizer you have
- **name:** DC removal on/off
code: dc_removal
help: Is DC removal filter activated or not
- **name:** Samples per seconds
code: sampling_rate
help: Select the sampling rate for this channel
- **name:** Frontend gain
code: preamplifier_gain
help: Select the preamplifier gain (frontend)

responses:

- **path:** nanometrics/TAURUS-G-1.response.yaml
applicable_filters:
 - digitizer_manufacturer:** Nanometrics
 - digitizer_model:** Taurus
 - preamplifier_gain:** Gain 1 (1x - 0dB - 16Vpp)

(continues on next page)

(continued from previous page)

```
- path: nanometrics/TAURUS-G-04.response.yaml
  applicable_filters:
    digitizer_manufacturer: Nanometrics
    digitizer_model: Taurus
    preamplifier_gain: Gain 0.4 (0.4x - 0dB - 40Vpp)

- path: nanometrics/TAURUS-G-2.response.yaml
  applicable_filters:
    digitizer_manufacturer: Nanometrics
    digitizer_model: Taurus
    preamplifier_gain: Gain 2 (1x - 0dB - 8Vpp)

- path: nanometrics/TAURUS.10.response.yaml
  applicable_filters:
    digitizer_manufacturer: Nanometrics
    digitizer_model: Taurus
    sampling_rate: 10 sps

- path: nanometrics/TAURUS.40.response.yaml
  applicable_filters:
    digitizer_manufacturer: Nanometrics
    digitizer_model: Taurus
    sampling_rate: 40 sps

- path: nanometrics/TAURUS.50.response.yaml
  applicable_filters:
    digitizer_manufacturer: Nanometrics
    digitizer_model: Taurus
    sampling_rate: 50 sps

- path: nanometrics/TAURUS.100.response.yaml
  applicable_filters:
    digitizer_manufacturer: Nanometrics
    digitizer_model: Taurus
    sampling_rate: 100 sps

- path: nanometrics/TAURUS.200.response.yaml
  applicable_filters:
    digitizer_manufacturer: Nanometrics
    digitizer_model: Taurus
    sampling_rate: 200 sps

- path: nanometrics/TAURUS.250.response.yaml
  applicable_filters:
    digitizer_manufacturer: Nanometrics
    digitizer_model: Taurus
    sampling_rate: 250 sps
```

and a few examples of the AROL response files are:

nanometrics/TAURUS-G-1.response.yaml

```
---
format_version: '0.106'
response:
  decimation_info:
    delay_correction: true
  stages:
    - input_units:
        name: V
        description: Volts
      output_units:
        name: V
        description: Volts
      gain:
        value: 1.0
        frequency: 0.0
      filter:
        type: ANALOG
        name: 'AMPLIFIER_FILTER'
        delay: 0.0
  extras:
    DBIRD_response_type: THEORETICAL
    Number_of_zeroes: '0'
    Number_of_poles: '0'
  notes:
    - '# Diviseur par 1 en preampli'
    - 'Response_type : THEORETICAL # optional'
    - 'Input_unit : V # optional'
    - 'Output_unit : V # optional'
    - 'Transfer_normalization_frequency : 0 # optional'
```

nanometrics/TAURUS-G-2.response.yaml

```
---
format_version: '0.106'
response:
  decimation_info:
    delay_correction: true
  stages:
    - input_units:
        name: V
        description: Volts
      output_units:
        name: V
        description: Volts
      gain:
        value: 2.0
        frequency: 0.0
      filter:
        type: ANALOG
```

(continues on next page)

(continued from previous page)

```

    name: ' AMPLIFIER_FILTER'
    delay: 0.0
extras:
  DBIRD_response_type: THEORETICAL
  Number_of_zeroes: '0'
  Number_of_poles: '0'
notes:
  - '#   Diviseur par 1 en preampli'
  - 'Response_type           : THEORETICAL   # optional'
  - 'Input_unit              : V             # optional'
  - 'Output_unit             : V             # optional'
  - 'Transfer_normalization_frequency : 0      # optional'

```

nanometrics/TAURUS-G-04.response.yaml

```

---
format_version: '0.106'
response:
  decimation_info:
    delay_correction: true
  stages:
    - input_units:
        name: V
        description: Volts
      output_units:
        name: V
        description: Volts
      gain:
        value: 0.4
        frequency: 0.0
      filter:
        type: ANALOG
        name: ' AMPLIFIER_FILTER'
        delay: 0.0
extras:
  DBIRD_response_type: THEORETICAL
  Number_of_zeroes: '0'
  Number_of_poles: '0'
notes:
  - '#   Diviseur par 1 en preampli'
  - 'Response_type           : THEORETICAL   # optional'
  - 'Input_unit              : V             # optional'
  - 'Output_unit             : V             # optional'
  - 'Transfer_normalization_frequency : 0      # optional'

```

nanometrics/TAURUS.100.response.yaml

```

---
format_version: '0.106'
response:
  decimation_info:
    delay_correction: true
  stages:
    - input_units: &id001
      name: V
      description: Volts
      output_units: &id002
      name: counts
      description: Digital Counts
      gain:
        value: 1000012.875
        frequency: 0.0
      filter:
        type: AD_CONVERSION
        input_full_scale: "16.777"
        output_full_scale: "16777216.000000"
      name: ' DIGITIZER'
      input_sample_rate: 0.0
      output_sample_rate: 30000.0003000000003
      delay: 0.0
    - input_units: *id002
      output_units: *id002
      filter:
        $ref: include/tau_FirSym2_s1_100.filter.yaml#filter
      name: ' DECIMATION'
      input_sample_rate: 30000.0003000000003
      output_sample_rate: 2000.0
      delay: 0.0
      decimation_factor: 15
    - input_units: *id002
      output_units: *id002
      filter:
        $ref: include/tau_FirSym2_s2_100.filter.yaml#filter
      name: ' DECIMATION'
      input_sample_rate: 2000.0
      output_sample_rate: 200.0
      delay: 0.0
      decimation_factor: 10
    - input_units: *id002
      output_units: *id002
      filter:
        $ref: include/tau_FirSym2_s3_100.filter.yaml#filter
      name: ' DECIMATION'
      input_sample_rate: 200.0
      output_sample_rate: 100.0
      delay: 0.0
      decimation_factor: 2
  extras:

```

(continues on next page)

(continued from previous page)

```

DBIRD_response_type: "THEORETICAL"
Output_sampling_interval: '1.0000000e-02'
notes:
- '# Generated by G. Cougoulat LGIT '
- '# On: 10/25/2010'
- "Response_name           : DIGITIZER# optionnel"
- "Response_type          : THEORETICAL# optionnel"
- "Input_sampling_interval : 0# optionnel  "
- "Output_sampling_interval : 3.3333333e-05# (= 5.10^-4 / 15)  "
- "Input_full_scale        : 16.777# 16.777 Vpp , gain = 1"
- "Output_full_scale       : 16777216.0000000# 2^24  "
- "Sensitivity             : 1000012.875# 1\xB5v"
- '# stage 1 decimation par 15'
- "Response_name           : DECIMATION# optionnel"
- "Response_type          : THEORETICAL# optionnel"
- "Input_sampling_interval : 3.3333333e-05# 3.0000000e-04# ~ 30000\
  \ / 15 = "
- "Output_sampling_interval : 5.0000000e-04# 2.0000000e-03# optionnel"
- '# stage 2 decimation par 10'
- "Response_name           : DECIMATION# optionnel"
- "Response_type          : THEORETICAL# optionnel"
- "Input_sampling_interval : 5.0000000e-04# 2.0000000e-03"
- "Output_sampling_interval : 5.0000000e-03# 2.0000000e-02# optionnel"
- '# stage 3 decimation par 2'
- "Response_name           : DECIMATION# optionnel"
- "Response_type          : THEORETICAL# optionnel"
- "Input_sampling_interval : 5.0000000e-03#2.0000000e-02"
- "Output_sampling_interval : 1.0000000e-02# optionnel"

```

nanometrics/TAURUS.200.response.yaml

```

---
format_version: '0.106'
response:
  decimation_info:
    delay_correction: true
  stages:
    - input_units: &id001
      name: V
      description: Volts
      output_units: &id002
      name: counts
      description: Digital Counts
      gain:
        value: 1048576.0
        frequency: 0.0
      filter:
        type: AD_CONVERSION
        input_full_scale: "16.0"
        output_full_scale: "16777216.0000000"

```

(continues on next page)

(continued from previous page)

```

name: ' DIGITIZER'
input_sample_rate: 0.0
output_sample_rate: 30000.0003000000003
delay: 0.0
- input_units: *id002
  output_units: *id002
  filter:
    $ref: include/tau_FirSym2_s1_200.filter.yaml#filter
name: ' DECIMATION'
input_sample_rate: 30000.0003000000003
output_sample_rate: 2000.0
delay: 0.0
decimation_factor: 15
- input_units: *id002
  output_units: *id002
  filter:
    $ref: include/tau_FirSym2_s2_200.filter.yaml#filter
name: ' DECIMATION'
input_sample_rate: 2000.0
output_sample_rate: 400.0
delay: 0.0
decimation_factor: 5
- input_units: *id002
  output_units: *id002
  filter:
    $ref: include/tau_FirSym2_s3_200.filter.yaml#filter
name: ' DECIMATION'
input_sample_rate: 400.0
output_sample_rate: 200.0
delay: 0.0
decimation_factor: 2
extras:
  DBIRD_response_type: "THEORETICAL"
  Output_sampling_interval: '5.00000000e-03'
notes:
- '# Generated by G. Cougoulat LGIT '
- '# On: 10/25/2010'
- "Response_name                : DIGITIZER# optionnel"
- "Response_type                : THEORETICAL# optionnel"
- "Input_sampling_interval      : 0# optionnel "
- "Output_sampling_interval     : 3.3333333e-05# (= 5.10^-4 / 15) "
- "Input_full_scale             : 16.0# 16 Vpp , gain = 1"
- "Output_full_scale            : 16777216.0000000# 2^24 "
- "Sensitivity                  : 1048576.0# 1\xB5v"
- '# stage 1 decimation par 15'
- "Response_name                : DECIMATION# optionnel"
- "Response_type                : THEORETICAL# optionnel"
- "Input_sampling_interval      : 3.3333333e-05# 3.0000000e-04# ~ 30000\
  \ / 15 = "
- "Output_sampling_interval     : 5.0000000e-04# 2.0000000e-03# optionnel"
- '# stage 2 decimation par 5'
- "Response_name                : DECIMATION# optionnel"

```

(continues on next page)

(continued from previous page)

```

- "Response_type           : THEORETICAL# optionnel"
- "Input_sampling_interval : 5.00000000e-04# 2.00000000e-03"
- "Output_sampling_interval : 2.50000000e-03# 2.00000000e-02# optionnel"
- '# stage 3 decimation par 2'
- "Response_name           : DECIMATION# optionnel"
- "Response_type           : THEORETICAL# optionnel"
- "Input_sampling_interval : 2.50000000e-03#2.00000000e-02"
- "Output_sampling_interval : 5.00000000e-03# optionnel"

```

In obsinfo, we would have 2 choices for implementing the two choices here: sampling rate and gain: 1) put the gain into a “preamplifier” file and the “sampling rate” into a

datalogger file

2) put both in a datalogger file

The second is more complicated but also easier to directly translate from AROL and expandable to other configuration dimensions such as the choice between minimum phase and linear phase filtering, or the implementation of a DC removal filter. We show this option below.

First, we would break the different stages into their own files:

tau_DIGITIZER.100.stage.yaml .. code-block:: yaml

— format_version: ‘0.110’ stage:

- **input_units: &id001** name: V description: Volts
- output_units: &id002** name: counts description: Digital Counts
- gain:** value: 1000012.875 frequency: 0.0
- filter:** type: AD_CONVERSION input_full_scale: “16.777” output_full_scale: “16777216.000000”
- name: ‘DIGITIZER’ input_sample_rate: 0.0 output_sample_rate: 30000.000300000003 delay: 0.0

tau_DIGITIZER.200.stage.yaml .. code-block:: yaml

— format_version: ‘0.110’ stage:

- **input_units: &id001** name: V description: Volts
- output_units: &id002** name: counts description: Digital Counts
- gain:** value: 1000012.875 frequency: 0.0
- filter:** type: AD_CONVERSION input_full_scale: “16.777” output_full_scale: “16777216.000000”
- name: ‘DIGITIZER’ input_sample_rate: 0.0 output_sample_rate: 30000.000300000003 delay: 0.0

tau_FirSym2_s1_100.stage.yaml .. code-block:: yaml

— format_version: ‘0.110’ stage:

- **input_units: *id002** name: counts description: Digital Counts
- output_units: *id002** name: counts description: Digital Counts
- filter:** \$ref: include/tau_FirSym2_s1_100.filter.yaml#filter
- name: ‘DECIMATION’ delay: 0.0 decimation_factor: 15

tau_FirSym2_s2_100.stage.yaml .. code-block:: yaml

— format_version: '0.110' stage:

- **input_units:** *id002 name: counts description: Digital Counts
- output_units:** *id002 name: counts description: Digital Counts
- filter:** \$ref: include/tau_FirSym2_s2_100.filter.yaml#filter
- name: 'DECIMATION' delay: 0.0 decimation_factor: 10

tau_FirSym2_s3_100.stage.yaml .. code-block:: yaml

— format_version: '0.110' stage:

- **input_units:** *id002 name: counts description: Digital Counts
- output_units:** *id002 name: counts description: Digital Counts
- filter:** \$ref: include/tau_FirSym2_s3_100.filter.yaml#filter
- name: 'DECIMATION' delay: 0.0 decimation_factor: 2

Notice that we lost the sampling rates and had to specify the units in each file rather than using YAML anchors. Here is the modified 100 sps datalogger file:

```
---
format_version: '0.110'
datalogger:
  sample_rate: 100
  delay_correction: true
  response_stages:
    - {$ref: 'tau_DIGITIZER.stage.yaml#stage'}
    - {$ref: 'tau_FirSym2_s1.stage.yaml#stage'}
    - {$ref: 'tau_FirSym2_s2.stage.yaml#stage'}
    - {$ref: 'tau_FirSym2_s3.stage.yaml#stage'}
```

This file is ALMOST AROL-compatible, except that

- the output `sample_rate` is specified
- `response` is renamed `datalogger` and can include equipment (where is this in AROL?)
- `stages` is renamed `response_stages` (change in v0.111?).
- `delay_correction` is not under `decimation_info` (change in v0.111?)

Many of these differences come from our “flattening” of the StationXML *Response*, should we move back for compatibility?

We can add one of the gains as well, here before the digitizer:

```
---
format_version: '0.110'
datalogger:
  sample_rate: 100
  delay_correction: true
  response_stages:
    - {$ref: "TAURUS-G-1.response.yaml#response/stage"}
    - {$ref: 'tau_DIGITIZER.stage.yaml#stage'}
    - {$ref: 'tau_DIGITIZER.stage.yaml#stage'}
    - {$ref: 'tau_FirSym2_s1.stage.yaml#stage'}
```

(continues on next page)

(continued from previous page)

- `{ $ref: 'tau_FirSym2_s2.stage.yaml#stage' }`
- `{ $ref: 'tau_FirSym2_s3.stage.yaml#stage' }`

Now, putting configurations inside the Datalogger file we have:

```

---
format_version: '0.110'
datalogger:
  sample_rate: 100
  delay_correction: true
  response_stages:
    - { $ref: "TAURUS-G-1.response.yaml#response/stage" }
    - { $ref: 'tau_DIGITIZER.stage.yaml#stage' }
    - { $ref: 'tau_DIGITIZER.stage.yaml#stage' }
    - { $ref: 'tau_FirSym2_s1.stage.yaml#stage' }
    - { $ref: 'tau_FirSym2_s2.stage.yaml#stage' }
    - { $ref: 'tau_FirSym2_s3.stage.yaml#stage' }
  configuration_default: '100sps_G1'
  configuration_definitions:
    "100sps_G1":
      configuration_description: "100 sps, gain=1"
    "100sps_G2":
      configuration_description: "100 sps, gain=2"
      response_stages:
        - { $ref: "TAURUS-G-2.response.yaml#response/stage" }
        - { $ref: 'tau_DIGITIZER.100.stage.yaml#stage' }
        - { $ref: 'tau_FirSym2_s1_100.stage.yaml#stage' }
        - { $ref: 'tau_FirSym2_s2_100.stage.yaml#stage' }
        - { $ref: 'tau_FirSym2_s3_100.stage.yaml#stage' }
    "100sps_G04":
      configuration_description: "100 sps, gain=0.4"
      response_stages:
        - { $ref: "TAURUS-G-04.response.yaml#response/stage" }
        - { $ref: 'tau_DIGITIZER.100.stage.yaml#stage' }
        - { $ref: 'tau_FirSym2_s1_100.stage.yaml#stage' }
        - { $ref: 'tau_FirSym2_s2_100.stage.yaml#stage' }
        - { $ref: 'tau_FirSym2_s3_100.stage.yaml#stage' }
    "200sps_G1":
      sample_rate: 200
      configuration_description: "200 sps, gain=1"
      response_stages:
        - { $ref: "TAURUS-G-1.response.yaml#response/stage" }
        - { $ref: 'tau_DIGITIZER.200.stage.yaml#stage' }
        - { $ref: 'tau_FirSym2_s1_200.stage.yaml#stage' }
        - { $ref: 'tau_FirSym2_s2_200.stage.yaml#stage' }
        - { $ref: 'tau_FirSym2_s3_200.stage.yaml#stage' }
    "200sps_G2":
      sample_rate: 200
      configuration_description: "200 sps, gain=2"
      response_stages:
        - { $ref: "TAURUS-G-2.response.yaml#response/stage" }
        - { $ref: 'tau_DIGITIZER.200.stage.yaml#stage' }

```

(continues on next page)

(continued from previous page)

```
- {$ref: 'tau_FirSym2_s1_200.stage.yaml#stage'}
- {$ref: 'tau_FirSym2_s2_200.stage.yaml#stage'}
- {$ref: 'tau_FirSym2_s3_200.stage.yaml#stage'}
"200sps_G04":
  sample_rate: 200
  configuration_description: "200 sps, gain=0.4"
  response_stages:
    - {$ref: "TAURUS-G-04.response.yaml#response/stage"}
    - {$ref: 'tau_DIGITIZER.200.stage.yaml#stage'}
    - {$ref: 'tau_FirSym2_s1_200.stage.yaml#stage'}
    - {$ref: 'tau_FirSym2_s2_200.stage.yaml#stage'}
    - {$ref: 'tau_FirSym2_s3_200.stage.yaml#stage'}
  extras:
    DBIRD_response_type: "THEORETICAL"
    Output_sampling_interval: '1.00000000e-02'
```

This is not the best example, as the Taurus appears to use different FIRs for each sampling rate: in many cases we can further reduce repetition by giving the same FIRs for different sampling rates, simply adding FIRs to account for the different sampling rates (e.g. CS5321 converters) or having a different sampling rate for the first stage (e.g. AD128* converters). One of the advantages is that we only specify a given FIR (or IIR) sequence once in the hierarchy

6.6.3 Converting obsinfo files to AROL

This should be fairly straightforward via a program that “explodes” an InstrumentComponent file into individual files and an AROL-style configuration file.

6.6.4 Converting AROL files to obsinfo

It may be possible to write a routine that combines a given set of files into an obsinfo file with different configurations, although there would be much repetition in this file.

6.6.5 Directly using AROL files in obsinfo

For now, this is not directly possible because of the lack of configuration information in the AROL InstrumentComponent files. See below for converting AROL files to obsinfo

Another, more complicated option would be to allow obsinfo to read AROL-style configuration files (\$arol operator instead of \$ref?), but the top-level configuration would have to allow several fields for the configuration

DATACITE INFORMATION FILES

7.1 Introduction

datacite information files are used to allow the principal scientist to provide information for DOI datacite files. The DataCite standard is described at <https://schema.datacite.org/meta/kernel-4.3/> and there are some tools for directly creating Datacite files, such as <https://github.com/claودیsf/datacite-metadata-generator>

The datacite information files simplify entering values for principal scientists, by proposing a limited but usually sufficient list of fields and values for these fields, as shown below.

One of the simplifications is that only a few identifier schemes are proposed: ORCID for people and ROR or GRID for organizations. To find the ROR URI for your organization, you can go to <https://ror.org> and search for your organization, or just do a web search on “{your institution} ROR”

The fields provided are:

Name	#	Type	Description
<i>title</i>	1	string	Title for the experiment
<i>description</i>	1	string	Description of the experiment
<i>creators</i>	1	list of <i>Entity</i>	List of authors (Entity objects), in publication order
<i>subjects</i>	1	list of string	Keywords
<i>contributors</i>	1	object	
<i>`data_collectors`</i>	1	list of <i>Entity</i>	Data collectors not in <i>creators</i> (engineers, scientists)
<i>`project_leader`</i>	0-1	<i>Entity</i>	use only if not the same as the first creator
<i>`project_members`</i>	0-1	list of <i>Entity</i>	Participants not in <i>creators</i> or <i>data_collectors</i>
<i>related_identifiers</i>	0-1	list of <i>RelIdent</i>	Things like articles, other DOIs, etc
<i>place</i>	1	string	The experiment location
<i>funders</i>	0-1	list of <i>Funder</i>	Experiment funders

7.1.1 Entity

describes people or organizations in the lists of creators or contributors:

Name	#	Type	Description
<i>name</i>	1	string	“family, given” if Personal name
<i>type</i>	0-1	string	‘Organization’, if not a Personal name
<i>identifier</i>	0-1	string	affiliation unique identifier (i.e. ‘ https://orcid.org/0000-0002-3260-1826 ’)
<i>scheme</i>	1	string	identifier scheme, required if <i>identifier</i> provided (i.e. ‘ORCID’)
<i>affiliations</i>	0-1	list	list of affiliations
`name`	1	string	affiliation name (i.e ‘IPGP’)
`identifier`	0-1	string	affiliation unique identifier (i.e. ‘ https://ror.org/004gzgz66 ’)
`scheme`	1	string	identifier scheme, required if <i>identifier</i> provided (i.e. ‘ROR’)

7.1.2 RelIdent

describes related identifiers, such as DOIs and articles

Name	#	Type	Description
<i>identifier</i>	1	string	affiliation unique identifier (i.e. ‘ https://orcid.org/0000-0002-3260-1826 ’)
<i>scheme</i>	1	sc_string	identifier scheme
<i>relation</i>	1	rel_string	relation between the identifier and the current datacite:

An *sc_string* is a string chosen from: ARK arXiv bibcode DOI EAN13 EISSN Handle IGSN ISBN ISSN ISTC LISSN LSID PMID PURL UPC URL URN w3id

A *rel_string* is a string chosen from: IsCitedBy Cites IsSupplementTo IsSupplementedBy IsContinuedBy Continues IsDescribedBy Describes HasMetadata IsMetadataFor HasVersion IsVersionOf IsNewVersionOf IsPreviousVersionOf IsPartOf HasPart IsPublishedIn IsReferencedBy References IsDocumentedBy Documents IsCompiledBy Compiles IsVariantFormOf IsOriginalFormOf IsIdenticalTo IsReviewedBy Reviews IsDerivedFrom IsSourceOf IsRequiredBy Requires IsObsoletedBy Obsoletes

7.1.3 Funder

describes funders

Below is an example datacite information file from `_examples/datacite/EMSO-MOMAR_OBS.datacite.yaml`

```
---
format_version: "0.110"
datacite:
  title : EMSO-MOMAR
  description : "Seismology component of a multi-year multidisciplinary
    geophysical observatory on Lucky Strike volcano,
    Mid-Atlantic Ridge (37°N, 32°W)"
  creators:
    - name : Cannat, Mathilde
      identifier: https://orcid.org/0000-0002-5157-8473
      scheme: ORCID
    - name : Crawford, Wayne
      identifier: https://orcid.org/0000-0002-3260-1826
      scheme: ORCID
    - name : IPGP Marine Geosciences Team
      type : Organization
      affiliations :
```

(continues on next page)

(continued from previous page)

```

- name: IPGP
  identifier: https://ror.org/004gzqz66
  scheme : ROR
subjects :
- Mid-ocean ridge volcanos
- Hydrothermal fields
dates_collected: 2007-07-18/2022-08-24
contributors:
  data_collectors:
    - name: Daniel, Romuald
      affiliations:
        - name: INSU-IPGP OBS Facility
    - name: Besancon, Simon
      affiliations:
        - name: INSU-IPGP OBS Facility
    - name: INSU-IPGP OBS Facility
      type: Organization
      identifier: "Need to make an ROR for the facility"
      scheme: ROR
  project_members:
    - name: Bohidar, Soumya
      affiliations:
        - name: IPGP
related_identifiers:
- identifier: www.doi.org/1234567
  scheme: DOI
  relation: HasMetadata
place: Lucky Strike volcano, Mid-Atlantic Ridge
funders:
- name: ANR
  identifier: https://ror.org/00rbzpz17
  scheme: ROR
  award_URI: https://anr.fr/Project-ANR-14-CE02-0008
  award_title: "Magma chamber to micro-habitats : dynamics of deep sea
    hydrothermal ecosystems - LuckyScales"

```


8.1 Channel modifications

The full power of **obsinfo** is achieved using channel modifications. OBS are subject to a lot of variations in configuration, even changes of components in the field. The aim of *obsinfo* is to have a relatively stable database of information files for instrumentation. But in actual operation it is common to reassemble the instrumentation, substituting stages and even whole components. This needs to be reflected in the metadata without compromising the stability of the instrumentation database. The way to do this is through channel modifications, which can change *any* attribute in the configuration, down to serial numbers or pole/zeros of filters at the channel level.

The attribute `channel_modifications` is used for this. Under this keyword the user can specify a complete hierarchy down to the filter level. Only the value(s) specified will be modified. So if a “leaf” value is changed, such as gain value, only the gain value for a particular stage will be changed. But if a complete sensor is specified, the whole component along with its stages and filters will be modified.

8.1.1 Specifying the channel to change

A channel must be identified in order to apply particular modifications to it. Channel identification is performed **not** by the channel label in the `channels` attribute, but by a channel code formed by the orientation and the location in the form:

```
"<orientation code>-<location code>"
```

Recall `orientation codes` are defined as an FDSN standard.

For example,

```
"Z-02"
```

If the location code is omitted, a location code of “00” is assumed. It is also possible to select all channels using “*”:

Channel Code	Result
“*” or “*-”	All channels
“*-00”	All orientations with location “00”
“H-”	Channel with “orientation” H and all locations
“H”	Channel with “orientation” H and location “00”
“1-01”	Channel with orientation 1 and location “01”

Channel codes have priorities. The more specific code will take precedence over the less specific code, and the orientation code specification will take precedence over the location code specification. That is, if both “*-” and “1-01” are specified, “1-01” will take precedence. And if “*-00” and “H-” are specified, “*-00” will take precedence.

8.1.2 Specifying an element to change

The elements under a given **Channel Code** can be:

- datalogger
- sensor
- preamplifier
- orientation_code

The first three allow you to change elements of one of these instrument components, and/or the entire instrument component and `orientation_code` lets you change the orientation code.

The first three can be filled with any valid entities of their parent entity, plus any of following keywords:

- `base`, which replaces the instrument component before applying the other entities.
- `configuration`, which applies the configuration specified in the instrument component's `configuration_definitions` field
- `serial_number`, which sets the serial number for the given instrument component. This is simply a shortcut for `{equipment: {serial_number: }}`.
- `response_modifications`, which allow you to change parameters in individual stages

For example, if you want to specify the sensor's serial number you could enter:

```
sensor:
  equipment: {serial_number: 'A1542'}
```

or

```
sensor:
  serial_number: 'A1542'
```

If you want to change the type of sensor, specify its serial number and use a custom configuration, you could enter:

```
sensor:
  base: {$ref: 'sensors/T240.nanometrics.sensor.yaml#sensor'}
  serial_number: '235'
  configuration: "SINGLE-SIDED"
```

8.1.3 Specifying a stage to change

Since stages have no name, they are referenced by their number, which specifies the order of the stage (starting at 0) within a given instrument component (sensor, preamplifier or datalogger). Modifications to stages are specified using the keyword `response_modifications`, as the keyword `response_stages` will overwrite all stages.

So, if we want to change the gain the third stage of the sensor, the hierarchy would look like this:

```
channel_modifications:
  sensor:
    response_modifications:
      "2": {gain: {value: 17}}
```

If we, however, want to replace all of the response stages, the file would look like this:


```
channel_modifications:
  datalogger:
    response_stages:
      - $ref: "responses/CS5321_FIR1.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR2.stage.yaml#stage"
      - $ref: "responses/CS5322_FIR3.stage.yaml#stage"
```

Response modifications are very flexible. The label can be any of several regular expressions. Either the single number, as above, a list, a range or a wildcard "*", which means all:

Stage Number	Result
"*"	All stages
"[1,2,3]"	Second, third and fourth stages (stages start at 0)
"[1-3]"	Same as above

8.1.4 Example

Here is a complete example of a network file with channel_modifications:

```
---
format_version: "0.110"
yaml_anchors:
  obs_clock_correction_linear_defaults: &LINEAR_CLOCK_DEFAULTS
    time_base: "Seascan MCXO, ~1e-8 nominal drift"
    reference: "GPS"
    start_sync_instrument: 0
revision:
  authors:
    - {$ref: "authors/Wayne_Crawford.author.yaml#author"}
  date: "2019-12-19"
network:
  facility:
    reference_name: "INSU-IPGP"
    full_name: "INSU-IPGP OBS Park"
    contact:
      first_name: "Wayne"
      last_name: "Crawford"
      email: "crawford@ipgp.fr"
      website: "http://ipgp.fr"
    campaign_ref_name: "MYCAMPAIGN"
  network_info:
    $ref: "network_info/EMSO-MOMAR.network_info.yaml#network_info"

  stations:
    "BB_1":
      site: "My favorite site"
      start_date: "2015-04-23T10:00:00"
      end_date: "2016-05-28T15:37:00"
      location_code: "00"
      instrumentation:
```

(continues on next page)

(continued from previous page)

```

    $ref: "instrumentation/BBOBS1_2012+.instrumentation.yaml#instrumentation"
  channel_modifications:
    "*_*": {datalogger: {configuration: "62.5sps"}}
  locations:
    "00":
      base: {$ref: 'location_bases/BUC_DROP.location_base.yaml#location_base
→ '}
      position: {lon: -32.234, lat: 37.2806, elev: -1950}
  processing:
    - clock_corrections:
      linear_drift:
        <<: *LINEAR_CLOCK_DEFAULTS
        start_sync_reference: "2015-04-23T11:20:00"
        start_sync_instrument: 0
        end_sync_reference: "2016-05-27T14:00:00.2450"
        end_sync_instrument: "2016-05-27T14:00:00"
"BB_2":
  notes: ["An example of changing the sensor"]
  site: "My other favorite site"
  start_date: "2015-04-23T10:00:00Z"
  end_date: "2016-05-28T15:37:00Z"
  location_code: "00"
  instrumentation:
    $ref: "instrumentation/BBOBS1_2012+.instrumentation.yaml#instrumentation"
  channel_modifications:
    "*_*":
      sensor: {configuration: "Sphere06"}
      datalogger: {configuration: "62.5sps"}
    "H-*":
      sensor:
        serial_number: "IP007"
        response_modifications:
          "3": {gain: {value: 15}}
  locations:
    "00":
      base: {$ref: 'location_bases/BUC_DROP.location_base.yaml#location_base
→ '}
      position: {lon: -32.29756, lat: 37.26049, elev: -1887}
  processing:
    - clock_correct_linear_drift:
      <<: *LINEAR_CLOCK_DEFAULTS
      start_sync_reference: "2015-04-22T12:24:00"
      start_sync_instrument: 0
      end_sync_reference: "2016-05-28T15:35:00.3660"
      end_sync_instrument: "2016-05-28T15:35:02"

```

8.2 AROL compatibility

One of the objectives of *obsinfo* is to be compatible with the AROL instrumentation database. AROL is a yaml-based instrumentation database which can be explored through the Yasmine application. Its syntax is heavily based on version v0.106 of *obsinfo*. Efforts are underway to make the syntax of the current version of *obsinfo* and of AROL be as close as possible. However, since the philosophy is somewhat different, some discrepancies will be inevitable. AROL builds a configuration out of user choices made with the Yasmine tool. *obsinfo* lists all available configurations and lets the user choose using the configuration fields (`sensor_configuration`, `preamplifier_configuration`, `datalogger_configuration`) in a station or network information file.

The current effort is to make AROL yaml files readable by *obsinfo*. However, there are some outstanding issues:

1. AROL does not have an offset field in its filter information files. It has a field called `delay.samples` which fulfills the same function. Proposed solution: let AROL change name. If not possible, read the AROL file and change within the *obsinfo* application.
2. AROL uses units instead of `transfer_function_type` in Pole/Zero filters. Their value is directly translatable, via a table, to the `transfer_function_type` enumeration used by StationXML (see table below). Proposed solution: let AROL change names. If not possible, read the AROL file and change within the *obsinfo* application.

AROL unit	obsinfo/StationXML equivalent
“rad/s”	“LAPLACE (RADIANS/SECOND)”
“hz”	“LAPLACE (HERTZ)”
“z-transform”	“DIGITAL (Z-TRANSFORM)”

3. AROL names of “fake” filters ANALOG, DIGITAL and AD_CONVERSION are in CamelCase in *obsinfo*: Analog, Digital, ADConversion to be consistent with StationXML. **Proposed solution:** let AROL change name. If not possible, read the AROL file and change within the *obsinfo* application.
4. AROL specifies both `input_sample_rate` and `output_sample_rate` for all stages. *obsinfo* only specifies the input sample rate for the first stage in the whole instrument. It calculates all the other values out of decimation factors. This gives more flexibility to the definition of each individual stage in the `response_stages` field of an information file. **Proposed solution:** read the AROL file and ignore these fields within the *obsinfo* application.
5. AROL specifies response stages thus:

```
response:
  decimation_info:
    delay_correction: true
  stages:
```

obsinfo simply specifies `response_stages` and the `delay_correction` attribute is specified at the `datalogger` level, as it is the only place where it makes sense for the global instrument. Also, `delay_correction` is specified as either boolean in AROL or as a real number. In *obsinfo* a value of `None` is equivalent to AROL `False` and a numeric value is equivalent to AROL `True`. **Proposed solution:** make *obsinfo* read the AROL file and interpret this attribute. If found in a response other than the datalogger, give a warning and ignore.

8.3 Best practices

8.3.1 Place instrumentation information files in a central repository

One of the main pillars of *obsinfo* philosophy is reuse and the DRY (don't repeat yourself) principle. In order to achieve this every effort has been made to ensure reusability, but the ultimate responsibility for this lies with the user. It is strongly recommended to create a central repository of instrumentation information files which can then be reused by several campaigns. Instrumentations should be flexible enough, with several typical configurations, so the need to make modifications through `channel_modifications` is minimized.

The use of a central repository will also permit information to be protected assigning modification writes only to the responsible parties.

Campaign, network and station files can then be placed in different directories according to users and teams.

8.3.2 Use a file hierarchy for different objects

Although *obsinfo* gives you total flexibility to organize your files as you see fit, it is recommended to use a hierarchy of directories and the `obsinfo_datapath` variable setup with the `obsinfo-setup` application, whose use is explained in the *Installation and Startup Guide*.

8.3.3 Validate all information files bottom up

Before trying to create a Station XML file, all information files should be validated individually. The best way to do this is to proceed bottom up: first validate filters, then stages, then components, then instrumentations, then networks. This way one can avoid very large output messages which are difficult to parse.

Files in central repositories should never be uploaded unless they're previously validated. Conversely, users can assume they don't have any need to validate central repositories.

8.3.4 Verification of response_stages in information file

All files in central repositories must be validated before being uploaded. It is good practice to validate your files from the bottom up. That is, validate filter files first, stage next, and so on to network, unless you're using (already verified) central repository files. This is to avoid long and unreadable messages from the validator.

8.3.5 Reuse information files

Either create a repository of your own or use one of the central repository. If you plan on working offline, you can clone the GitLab repository locally.

8.3.6 Document information files with notes, extras and comments

A choice of documentation options are available in information files. Aside of the “#” comment mechanism of the YAML language, **notes** are used for *obsinfo* documentation that will **not** be reflected in the resulting StationXML file. On the other hand, **comments** can be used at the **network**, **station** and **channel** levels which will be incorporated into the StationXML file. The reason to not extend this to **stage** and **filter** is that the resulting StationXML file would be cluttered with repeated comments. Similarly, at the same levels, **extras** can be specified. These take the form of key/value pairs to simulate attributes which are not actually present in StationXML but can be used for documentation. A typical example is: .. code-block:: yaml

```
DBIRD_response_type : "CALIBRATED"
```

which is used in several filters. It should be specified at the channel level, though, perhaps specifying to which filters it applies.

8.3.7 Placement of response_stages in information file

Although **response_stages** can be specified outside the **configuration_definitions**, this is discouraged unless there is a single configuration. If there are several configurations, **response_stages** should be specified in the **configuration_definitions**. Nevertheless, if **response_stages** is specified outside and there are several **configuration_definitions**, the one selected will overwrite the **response_stages**. If no configuration is specified, the **response_stages** outside will be used.

In the following example, “responses/INSU_BBOBS_gain0.225_theoretical.stage.yaml” will *always* be overwritten by “responses/INSU_SPOBS_L28x128_theoretical.stage.yaml#stage”, as there is a configuration default which corresponds to the configuration definition “128x gain”.

```
preamplifier:
  equipment:
    model: "GEOPHONE-GAIN"
    type: "Analog gain/filter card"
    description: "SIO gain/filter card, seismo channel"
    manufacturer: "SIO or IPGP"
    vendor: ""
    configuration_default: "128x gain"
    response_stages:
      - $ref: "responses/INSU_BBOBS_gain0.225_theoretical.stage.yaml#stage" # THIS_
↪ WILL BE OVERWRITTEN
    configuration_definitions:
      "128x gain":
        response_stages:
          - $ref: "responses/INSU_SPOBS_L28x128_theoretical.stage.yaml#stage"
```

Due to this, the best practice dictates that, in the presence of **configuration_definitions**, the **response_stages** in bold should be omitted. If **configuration_default** were not present and no configuration is selected at the channel/instrument level, a warning will be issued and the **response_stages** in bold will be used. If no response stages can be selected then an exception will be raised and the program will exit with error.

The following example, though, is perfectly OK, as there is a single configuration and no need to specify **configuration_definitions**:

```
preamplifier:
```

equipment: model: "GEOPHONE-GAIN" type: "Analog gain/filter card" description: "SIO gain/filter card, seismo channel" manufacturer: "SIO or IGP" vendor: ""

response_stages:

- \$ref: "responses/INSU_BBOBS_gain0.225_theoretical.stage.yaml#stage"

8.3.8 How to modify individual stages

Individual stage modification must always be specified in `response_modifications` at the instrument component level. This will NOT work:

```
preamplifier:
  {$ref: "preamplifiers/LCHEAPO_BBOBS.preamplifier.yaml#preamplifier"}
  gain:
    value: 34
```

This will:

```
preamplifier:
  {$ref: "preamplifiers/LCHEAPO_BBOBS.preamplifier.yaml#preamplifier"}
  response_modifications:
    "": gain:
      value: 34
```

8.3.9 Use of channel modifications

It should be the aim to create a large and flexible instrumentation database with lots of possible configurations. This way, channel modifications will be rarely used. In fact, it is recommended to use channel modifications sparingly. If they must be used, remember to **always** use the adequate syntax. No shortcuts are allowed. All the hierarchical syntax must be used. For example, to change the gain of a sensor stage you need to write:

```
channel_modifications:
  sensor:
    {$ref: "sensors/SIO_DPG.sensor.yaml#sensor"}
    response_modifications:
      "": gain:
        value: 34
```

and not a shortcut such as:

```
channel_modifications:
  gain:
    value: 34
```

as there is no way *obsinfo* can determine to which stage of which component to apply the modification in gain.

8.3.10 How to use notes

There is a single `notes` attribute in JSON/YAML files, which contains a list of notes, in order to make documentation more systematic. However, sometimes a user may want to comment a piece of the file (for example, a single station in a network file). To do so we recommend using the YAML notation for comments, “#” followed by the comment text. Currently there is no way to do this in JSON files.

8.3.11 Base your info files in templates

While syntax may be a challenge, we recommend strongly that, when starting a new information file, you use a template. That way at least you can guarantee that the indentation and spelling of attributes is right.

8.4 Notes

8.4.1 Date formats

Dates can be entered in regular “dd/mm/yyyy” format or in UTC format, either “yyyy-mm-dd” or “yyyy-mm-ddThh:mm:ss” or “yyyy-mm-ddThh:mm:ssZ”, according to norm [ISO 8601](#). The difference between the latter two formats is that the first represents local time and the second UTC time. The norm allows you to specify hours respective to UTC by adding or subtracting. This particular format is not allowed in *obsinfo*. Separators can be either “/” or “-”.

Dates in other formats will result in an exception.

No effort is made to validate if dates are legal, i.e., to reject dates such as “31/02/2021”.

8.5 Caveats

8.5.1 Caveat: Effect of `$ref`

In every case, the use of `$ref` is to totally substitute that line by the content referenced in the file path (under the “#” tag). It is completely equivalent to write the content of referenced file instead of the `$ref` line. This should be taken into account for syntax purposes. If syntax is not validated, a very good chance is that the `$ref` file syntax is either wrong or in the wrong place. Not finding the file also causes problems which are not evident at first glance; if you keep getting errors, check if the file is in the right place.

8.5.2 Caveat: Cryptic syntax messages

Unfortunately, the JSON/YAML parser is very terse reporting syntax errors such as unmatched parentheses, brackets or curly brackets, single items instead of lists, etc., usually result in an exception with a cryptic message. It is *strongly* recommended that YAML files are edited in a suitable editor which can check at least basic syntax errors. See also the Troubleshooting section below.

8.5.3 Caveat: Use of response stages

Although response stages are specified at the component (sensor, preamplifier and datalogger) level, in the end they are considered as a single response for the whole instrument. Response stages are taken in this order: sensor stages, preamplifier stages and datalogger stages, irrespective of the order in which components appear in the information file. Within a component, they are taken in the order specified in the information file. In the end, they are numbered from one to n for the whole response.

8.5.4 Caveat: Treatment of sample rates in response stages

Only the input sample rate should be specified for a response, starting in the ADConversion stage. All other input and output rate are calculated using the decimation factor of each digital stage. Therefore, `input_sample_rate` and `output_sample_rate` should never be specified for later digital stages, and decimation factor should *always* be specified for them.

Total sample rate is calculated for the whole response and checked against the sample rate specified in the datalogger. A warning will be issued if they are different.

8.5.5 Caveat: ALWAYS follow the syntax and beware of \$ref overwriting your attributes

A naïve approach to syntax might think that we can add fields, for example, to a `$ref` information file. For example, the file could be an instrumentation file and we could decide to add a datalogger configuration which is not present in the file:

```
..block-code:: yaml
```

```
    instrumentation: "$ref" :    "instrumentations/BBOBS1_2012+.instrumentation.yaml"    datalogger_config: "62.5sps"
```

This is wrong. First, it is the wrong syntax: what channel with the configuration be applied to? There is no indication of that. Remember: modifications should always follow the same syntax. If `datalogger_config` belongs under a channel, it should always be applied to a channel.

But there is another problem. “`$ref`” will substitute all attributes at the same level, thus erasing the attribute `datalogger_config`. If this happens it will be a silent error, since substitution occurs before validation and the application will never know `datalogger_config` was there. The correct way of applying `datalogger_config` is through `channel_modifications` at the station level.

8.6 Troubleshooting

Sometimes it is a challenge to understand where an error lies in an information file. Messages are sometimes cryptic. We recommend you use the best practice above of using templates to avoid indentation errors. Also, if you can, use the best practice of working with an editor which recognizes elementary YAML errors such as indentation or wrong parentheses balancing. Let’s review the most common sources of error:

1) `JSONEncodeError/EOF`. An error which raises a `JSONEncodeError` exception and/or `EOF` (end of file) reached prematurely. This is the most serious error, as it stops processing of the information file and exits the program. It is usually due to:

- a) Indentation
- b) Unmatched double or single quotes, parentheses, brackets, curly brackets
- c) Extra commas or not enough commas

Check for these, if possible, with an editor which shows matching signs. Use templates if possible. If everything else fails, try to reduce the information file to a bare bones version, see if the error persists. If it doesn't, add new attributes gradually. For example, a network file might have this kind of problem. Temporarily eliminate attributes such as `channel_modifications` and reduce the network to a single station.

2) File not found: `<filename>`. `<filename>` has not been found. Either the directory where the file exists is not correctly specified in the path of the argument or in OBSINFO-DATAPATH, or the `<filename>` is misspelled or the file does not exist. This is typical for files referenced in `$ref`.

3) Validation error: `<YAML expression>` is not valid under any of the given schemas. This means that the information file is recognized and correctly parsed, but the portion `<YAML expression>` is not recognized. This may be due to illegal values, illegal value type (e.g. a string instead of a number, or a string pattern which does not correspond to valid patterns. An example is the wrong version:

```
[ 'format_version': '0.107' ] is not valid under any of the given schemas
```

or a phone number with letters:

```
[ 'revision': { 'date': '2017-11-30', 'authors': [ { 'first_name': 'Wayne', 'last_name': 'Crawford', 'institution': 'IPGP', 'email': 'crawford@ipgp.fr', 'phones': [ '+33A 01 83 95 76 63' ] } ] } ] is not valid under any of the given schemas.
```

or a string instead of a list (in the phones attribute):

```
[ 'revision': { 'date': '2017-11-30', 'authors': [ { 'first_name': 'Wayne', 'last_name': 'Crawford', 'institution': 'IPGP', 'email': 'crawford@ipgp.fr', 'phones': '+33A 01 83 95 76 63' } ] } ] is not valid under any of the given schemas
```

One possibility with validation errors is that the output of the message may be too long and difficult to parse, as it shows the whole produced information files with all its subfiles. The way to avoid this is to apply the best practice of validation bottom-up: first validate filters, then stages, then components, then instrumentations, then networks. This way the output is manageable.

4) Validation error: Additional properties are not allowed (`<attribute name>` was unexpected) An attribute name was not recognized, either because it was misspelled or because it does not exist in the specification.

```
[ 'network': { 'operator': 'fulsdl_name' } ] Additional properties are not allowed ('fulsdl_name' was unexpected)
```

5) Validation error: `<attribute name>` is a required property A required attribute name was not included.

```
[ 'network': { 'operator': 'reference_name' } ] 'reference_name' is a required property
```

8.7 Addons

If you want to add codes allowing you to make scripts according to your needs, use the `obsinfo/addons/` directory. Existing classes/executables in this directory are:

- LCHEAPO: Go from SIO LC2000 data files to basic miniSEED.
- SDPCHAIN: Go from basic miniSEED to datacenter-ready data
- LC2SDS: Generate basic clock-corrected SDS data archives.
- OCA:

8.7.1 LCHEAPO

Creates scripts to generate miniSEED data from SIO LC2000 data files. miniSEED files are deliberately NOT drift corrected so that that step can be done at a data/metadata preparation 'A-node'

Run through the executable *obsinfo-makescripts-LCHEAPO*

the generated scripts need the *sdpchain* module (not publically available)

8.7.2 SDPCHAIN

Makes scripts to go from basic miniSEED to data center ready

Run through the executable *obsinfo-makescripts-SDPCHAIN*

the generated scripts need the *sdpchain* module (not publically available)

8.7.3 LC2SDS

Makes scripts to generate SDS data archives directly from SIO LC2000 data files, including basic clock corrections. To furnish facility users, NOT FDSN/EIDA-level data centers

Runs through the executable *obsinfo-makescripts-LC2SDS*

The generated scripts need the pip-available *lcheapo* module

8.7.4 OCA

Just a stub for metadata conversion between *obsinfo* and *OCA* protocols. Never finished.

TRAINING COURSE

9.1 Introducing obsinfo

9.1.1 Philosophy and comparison to other systems

obsinfo is a syntax and system to create standard seismological metadata files (currently StationXML) as well as processing flows specific to ocean bottom seismometer (OBS) data. It's basic philosophy is:

- 1) break down every component of the system into “atomic”, non-repetitive units.
- 2) Follow StationXML structure where possible, but:
 - a) Add entities missing from StationXML where necessary
 - b) use appropriate units for each component (for example, specifying the **offset** for a digital filter, not the delay, which depends on the sampling rate)
- 3) allow full specification of a deployment using text files, for repeatability and provenance

File formats

Compared to StationXML files

- Minimizes repeated information
 - for example, in StationXML
 - * each channel could have the same datalogger but all of the datalogger specifications are repeated for each channel.
 - * Within a channel's response itself, several of the stages may be identical (except for the **offset**).
- Eliminate fields that can be calculated from other fields, such as:
 - The `<InstrumentSensitivity>` field, which depends on the `Stage` s that follow
 - The `<Delay>` for a digital filter stage, which can be calculated from `<Offset> * <Factor> / <InputSampleRate>`

Compared to RESP files

RESP files (mostly used in the Nominal Reference Library) are just text representations of the Dataless SEED files that preceded the StationXML standard, so they share the repetitive nature of StationXML files and add the complexity of a non-standard text format.

Compared to AROL

The Atomic Response Objects Library ([AROL](#)) replaces the RESP-based Nominal Response Library in the new YASMINE system. Files use the same atomic concept and YAML structure as *obsinfo*, in fact the AROL format was based on a previous version of *obsinfo* and we try to keep the two compatible.

AROL lacks the `network`, `station` and `instrumentation` levels as these are assembled by YASMINE.

Metadata creation systems

Compared to PDCC

PDCC is a graphical user interface allowing one to assemble different components (sensors, dataloggers, amplifiers) and then add in deployment information. Components can be added from the Nominal Response Library ([NRL](#)), which combines RESP files with textual configuration files which allow the user to select the exact component and configuration they used. *obsinfo* uses a fully textual description of instruments and deployments rather than a graphical user interface.

Compared to IRIS DMC IRISWS

I don't know much about this, it looks like a webservice to obtain component responses but I'm not sure how you're supposed to assemble them. It might just be a more modern way to access the NRL components that is supposed to be used by newer systems.

Compared to YASMINE

YASMINE is a new StationXML metadata creation tool. It's major difference from PDCC is its use of atomic response files, which should be compatible with *obsinfo* files. It provides a graphical user interface ([YASMINE-EDITOR](#)) and a command-line interface ([YASMINE-CLI](#)). The major differences from *obsinfo* are the lack of `instrumentation`, `station` and `network` levels, as well as processing information such as instrument clock drift

9.1.2 File formats

All information files can be written in [YAML](#) or [JSON](#) format. Use whichever you prefer. [YAML](#) is generally easier to write and read by humans, whereas [JSON](#) is easier for computers. The tutorial includes a section describing [YAML](#) files as used in *obsinfo* (tutorial:tutorial-1). There are many sites for converting from one format to the other and for validating either format: including this [json-to-yaml-converter](#) and this [yaml-validator](#).

9.1.3 The Tutorial

This training course is meant to accompany an instructor. The tutorial provides a more detailed step-by-step explanation and we refer to sections of the Tutorial throughout this training course.

9.1.4 Structural units

A full obsinfo network description consists of the following entities (starred fields are optional):

```
format_version: {}
*revision: {}*
*notes: []*
network:
  operator: {}
  *stations_operator: {}*
  *restricted_state: {}*
  *comments: []*
  *extras: {}*
  campaign_ref_name: string
  network_info: {}
  stations:
    <STATIONNAME1>:
      site: string
      start_date: string
      end_date: string
      location_code: string
      *serial_number: string*
      *operator: {}*
      instrumentation:
        *operator: {}*
        equipment: {}
        channels:
          default:
            *orientation_code: string or {}*
            datalogger:
              << GENERIC_COMPONENT
              sample_rate: number
              *delay_correction: number*
            *preamplifier:*
            *<< GENERIC_COMPONENT*
            sensor:
              << GENERIC_COMPONENT
              seed_codes:
                *preamplifier_configuration: string*
                *sensor_configuration: string*
                *datalogger_configuration: string*
                *location_code: string* # otherwise inherits from station
                *comments: []*
                *extras: {}*
            <SPECIFIC-CHANNEL1>: {}
            <SPECIFIC-CHANNEL2>: {}
            ...
```

(continues on next page)

(continued from previous page)

```

    *channel_modifications: {}*
    locations: {}
    *notes: []*
    *comments: []*
    *extras: {}*
    *processing:*
      - *clock_correction_linear: {}*
      - *clock_correction_leapsecond: {}*
    <STATIONNAME2>:
      . . .

```

Where GENERIC_COMPONENT is:

```

equipment: {}
*configuration_default: string*
*configuration_definitions: {}*
*response_modifications: {}*
*notes: []*
*response_stages:*
  - stage:
    input_units: string
    output_units string
    gain: float
    *name: string*
    *description: string*
    *decimation_factor: integer*
    *delay: number*
    *calibration_date: string*
    *polarity: '+' or '-'* # default is '+'
    *input_sample_rate: number*
    *filter:*
      type: string
      <fields depending on type>
  - stage:
  - ...

```

And FILTER is:

```

type: string # one of "PolesZeros", "FIR", "Coefficients",
             # "ResponseList", "Polynomial", "ADConversion",
             # "Analog", "Digital"
*description: string*
*offset: integer* # not for "Analog", "Digital" or "ADConversion"
# other parameters specific to the specified type

```

This could all be in one file, in which case there would be little benefit over StationXML. The power of obsinfo comes from the ability to put any sub-entity into a separate file, which is called from the parent file using the \$ref field.

Standard file levels are: network, instrumentation, {datalogger, preamplifier, sensor}, stage and filter. The schema files are defined at these same levels, allowing the command-line tool obsinfo-validate to validate any file ending with {one of the above}.{yaml,json}. Other elements often put into separate files are author, location_base, network_info and operator.

A common file structure is then (this time showing only the required fields):

- a network file:

```
format_version: {}
network:
  operator: {}
  campaign_ref_name: string
  network_info: {$ref: network_info/xxx.network-info.yaml#network_info}
  stations:
    <STATIONNAME1>:
      site: string
      start_date: string
      end_date: string
      location_code: string
      instrumentation: {$ref: instrumentation/xxx.instrumentation.yaml
↪#instrumentation}
      locations: {}
    <STATIONNAME2>:
      ...
    <STATIONNAME3>:
      ...
    ...
```

- instrumentation files:

```
format_version: {}
instrumentation:
  equipment: {}
  channels:
    default:
      datalogger: {$ref: dataloggers/xxx.datalogger.yaml#datalogger}
      sensor: {$ref: sensors/xxx.sensor.yaml#sensor}
    <SPECIFIC-CHANNEL1>: {}
    <SPECIFIC-CHANNEL2>: {}
    ...
```

- datalogger files:

```
format_version: {}
datalogger:
  << GENERIC_COMPONENT
  sample_rate: float
```

- sensor files:

```
format_version: {}
sensor:
  << GENERIC_COMPONENT
  seed_codes:
```

- stage files: (see examples in classes/stage)
- filter files: There are 4 filter types corresponding directly to their StationXML analogues: PoleZeros, FIR, Coefficients and ResponseList (the Polynomials filter type has not yet been implemented). 3 other filter types allow simpler information entry:
 - **Analog:** An analog stage with no filtering (translated to StationXML PoleZero without any poles or zeros)

- **Digital/**: A digital stage with no filtering (translated to StationXML Coefficients stage without any coefficients)
- **ADConversion**: like an analog stage, plus information about input voltage and output counts limits

(see examples in classes/filter)

You don't actually need to put the information in each file under a field with the filetype name: in fact if you didn't you would save a little typing, as you could specify, for example,

```
{ $ref: xxx.datalogger.yaml }
```

instead of:

```
{ $ref: xxx.datalogger.yaml#datalogger }
```

But the second style is preferred as it allows the files to contain useful provenance and version information at the base level. To incite you to use the second style, obsinfo-validate only accepts this style.

9.1.5 Comments, notes and extras

Comments and notes are both lists of text.

`comments` will be transformed in to StationXML comments. They can be entered at the `network`, `station` and `channel` level and will be transformed into StationXML comments at the same level.

`notes` will not go into the StationXML file, they are for your information only. They can be entered at the `base`, `station`, and `component` levels.

`extras` is a free object-based field. It can be used to add fields that may be useful in a future version of obsinfo. Nothing there is put into the StationXML code unless the obsinfo software is specifically updated to do so (which allows new fields without breaking compatibility or schema rules). They can be entered at the `network`, `station` or `channel` level

9.1.6 Configurations, channel modifications and shortcuts

components can have pre-defined **configurations** and their internal values can be **modified** from higher levels.

The simplest and most common example is specifying each station's sampling rate, which is done as follows:

```
channel_modifications:
  "*": {datalogger_configuration: "125sps"}
```

Configurations

Configurations modify parameters in a given **component** according to an existing `configuration_definition` in the component's information file.

Allowed fields are:

- `datalogger_configuration`
- `sensor_configuration`
- `preamplifier_configuration`

Configurations can be specified at the following levels, in order of priority:

- 1) station:channel_modifications
- 2) instrumentation:channels:{CHNAME}
- 3) instrumentation:channels:default

Configurations are defined in the the component information files under the `configuration_definition` field.

Channel Modifications

`channel_modifications` directly modify one or more parameters in a given element. This gives complete control to the user but assumes knowledge of the obsinfo hierarchy.

Details of `channel_modifications` are provided in the Advanced Topics section `advanced/chan_mods`

Shortcuts

`datalogger_configuration`, `preamplifier_configuration` and `sensor_configuration` are actually **shortcuts** for common `channel_modifications`. **Shortcuts** are hard-coded into obsinfo to allow simpler representation of common configurations or modifications. Other ones may be added, including `XX_serial_number`, where **XX** could be `datalogger`, `sensor`, `preamplifier` or `instrumentation`

Other sources

- Channel modifications are described briefly in `/tutorial/tutorial-3:channel modifications` and in detail in *Channel modifications*
- Component configurations are described in `/tutorial/tutorial-4:configurations` and `/tutorial/tutorial-5:configuration definitions` and `/tutorial/tutorial-6:datalogger configuration definitions`

9.1.7 Details

- Referenced files referenced are searched for starting at the paths given in the `~/ .obsinfoforc` file

9.1.8 delay, offset, and correction

One area where `obsinfo` differs from `StationXML` is in its handling of delays in digital filters. `StationXML` (and `RESP`) have three parameters in each stage, relating to the time delay created by the stage, in each Stage's `Decimation` section:

offset Sample offset chosen for use. If the first sample is used, set this field to zero. If the second sample, set it to 1, and so forth.

delay The estimated pure delay for the stage (in seconds). This value will almost always be positive to indicate a delayed signal.

correction The time shift, if any, applied to correct for the delay at this stage. The sign convention used is opposite the `<Delay>` value; a positive sign here indicates that the trace was corrected to an earlier time to cancel the delay caused by the stage and indicated in the `<Delay>` element.

`StationXML` specifies the **delay** for each stage, leaving the offset equal to zero. A digital filter's true delay is in samples, not seconds, meaning that the **delay** will depend on the sampling rate.

`obsinfo`'s atomic philosophy does not allow a variable delay (in seconds) when there is a constant delay (in samples). `obsinfo` puts **delay** in the stage level but **offset** in the filter level. For digital filters, **offset** should be filled with the delay samples and **delay** should not be provided.

9.1.9 Details

- Referenced files referenced are searched for starting at the paths given in the ~/.obsinfo file

9.1.10 Command-line files

all of the command line files start with **obsinfo-**, so if you have a decent shell you should be able to see them by typing `obsinfo<TAB>`

- `obsinfo-makeStationXML` makes stationXML files from an obsinfo network file and its dependencies
- `obsinfo-validate` validates network, instrumentation, datalogger, sensor, preamplifier, stage and filter files
- `obsinfo-print`
- `obsinfo-print_version`
- `obsinfo-setup` creates the .obsinfo file and can also create an example database.
- `obsinfo-test` runs a series of validation tests

The different `obsinfo-makescripts-*` command-line scripts are used for making IPGP-specific data processing flows, as described below. They could be used as a basis for creating your own data processing flows.

The directory `obsinfo/obsinfo/addons/` contains programs to create processing scripts using the information in the network files.

This is addressed in more detail in the `training_course/4_advanced` module

9.2 Setting up

9.2.1 Installation

Follow the instructions in the [Installation and Startup Guide], up to [Description of obsinfo file structure]. **Note that we have no Windows verification/development system, so Linux or MacOS systems are preferred (or someone who will verify/modify the code for Windows)**

9.2.2 Copying an example database into your folder

run `obsinfo-setup -d example_dir` to create an directory containing a full set of example information files

9.2.3 Creating a StationXML file using the example database

- go into `example_dir/Information/Files/network`
 - you should see several *.network.yaml files, including `SPOBS.INSU-IPGP.network.yaml`
- type `obsinfo-makeStationXML SPOBS.INSU-IPGP.network.yaml`

if all is in order, a valid StationXML file named `SPOBS.INSU-IPGP.station.xml` should be created. An easy way to verify is to try to read into obspy, like this:

Note that `obsinfo-makeStationXML` creates one StationXML file per station.

9.3 Creating a StationXML file using your own instruments and deployments

In each case, use:

- associated example and schema files to help you create the file(s)
- `obsinfo-validate` to validate your file(s)
- `obsinfo-print` for something?
- `obsinfo-makeStationXML` to verify that you can create a StationXML file

9.3.1 Creating a network file (using the example instruments)

Create your own network file, using existing instruments in the `example_directory`.

First-level associated files: `network` Second-level associated files: `author`, `network-info`, `operators`,
`location_bases`

9.3.2 Adding your own sensor/datalogger/analog filter

Add and validate sensor, datalogger and/or preamplifier(s) in the `example_directory`. Modify an example instrumentation file to reference the new component(s) and create a StationXML file

First-level associated files: `datalogger`, `sensor`, `preamplifier`, `stage`, `filter`

Second-level associated files: `author`

9.3.3 Adding your own instrumentation

Create a new instrumentation (OBS), using your newly created components and any other specific information

First-level associated files: `instrumentation` Second-level associated files: `author`, `operators`

9.3.4 Putting it all together

Create a network file corresponding to a deployment of your own instruments.

9.4 Advanced issues

9.4.1 Creating a processing pathway

The directory `obsinfo/obsinfo/addons/` contains programs to create processing scripts using the information in the network files. These scripts depend on the external programs used: `obsinfo` currently has programs for:

Transformation	file	command line
LCHEAPO OBS data to miniSEED using the SDPCHAIN tools	LCHEAPO.py	obsinfo-makescripts_LCHEAPO
uncorrected to corrected miniSEED using the SDPCHAIN tools	SD-PCHAIN.py	obsinfo-makescripts_SDPCHAIN
LCHEAPO OBS data to (poorly) corrected SDS structure	LS2SDS.py	obsinfo-makescripts_LC2SDS

These probably aren't directly applicable to other OBS facilities, but the files can serve as a basis for your own codes. The command line programs are created using the `console_scripts` parameter in `obsinfo/setup.py`

9.4.2 Storing and accessing your instrument database online

I have never done this, Luis discusses it in `tutorial/tutorial-3:file-discovery`

9.5 Future plans

9.5.1 obsinfo v0.111

All of this are “issues” on the gitlab site. All will require changes to input files (major change)

Replace “network” with “subnetwork”

Avoid confusion with FDSN network, more compatible with concept of multiple facilities/deployments possibly composing an FDSN network

Bring instrument modifiers under *station:instrumentation*

And allow instrumentation-level configuration by adding `modifications` and `config` fields.

- This could also just be `equipment_modifications` as I think equipment is all that is modified
- `serial_number` is shorthand for `(serial_number is in equipment)`, or maybe keep it as a shorthand for `equipment_modifications {serial_number: }`: declaring both should be considered an error (can it be done in the schema file?)

v0.110

```
instrumentation:
  {$ref: "xxx.instrumentation.yaml"}
serial_number: string
channel_modifications: object
```

0.111

```
instrumentation:
  base: {$ref: "xxx.instrumentation.yaml"}
  serial_number: string
  config: string
  modifications: {equipment: object of equipment parameters to change}
  channel_modifications: object
```

or

```
instrumentation:
  base: {$ref: "xxx.instrumentation.yaml"}
  serial_number: string
  config: string
  equipment_modifications: object of equipment parameters to change
  channel_modifications: object
```

Make “person” fields compatible with StationXML Person

Currently “author”, change to “person”

v0.110

```
first_name: string
last_name: string
institution: string
email: string
phones: array of strings or [country_code, area_code, phone_number] objects
```

0.111

```
name: string
agency: string
email: string
phone: string or [country_code, area_code, phone_number] objects
```

Make uncertainties into objects with measurement_method

Currently, azimuth.deg and dip.deg are 2-element arrays of number or null, with the first value being the value and the second the uncertainty, e.g.:

```
azimuth.deg: [90, 5]
dip.deg: [0, 1]
```

Change them to be objects with the possible attributes specified in the [`StationXML Documentation<http://docs.fdsn.org/projects/stationxml/en/latest/reference.html#azimuth>`](http://docs.fdsn.org/projects/stationxml/en/latest/reference.html#azimuth):

```
azimuth: {value.deg: 90, uncert.deg: 5, measurement_method: "Ppol"}
dip: {value.deg: 0, uncert.deg: 1, measurement_method: "Ppol"}
```

only *value.deg* should be required.

Should probably enable this for all fields that are defined as “FloatType” in the 1.1 XML schema:

- *WaterLevel* (units=m)
- *ClockDrift* (units=s?)
- *Response:Amplitude* (no units, specified elsewhere?)
- *Response:Phase* (units=deg)
- *Response:Frequency* (units=Hz)
- *Decimation:InputSampleRate* (units=Hz)

- *PzTransferFunctionType:NormalizationFrequency* (units=Hz)
- *ApproximationType:FrequencyLowerBound* (units=Hz)
- *ApproximationType:FrequencyUpperBound* (units=Hz)
- *Decimation:Delay* (units=s)
- *Decimation:Correction* (units=s)
- *DipType* (units=deg)
- *Depth* (units=m)
- *SampleRate* (units=sps)

But allow value-only shortcut definitions, in which case *uncert* and *measurement_method* will be set to *null*, for example *WaterLevel.m: 0*, *ClockDrift.s: 1e-10* and so on...

Latitude.deg Longitude.deg and *Elevation.m* are special cases for which the uncertainties and measurement method can be separately specified, as is currently the case.

Or should I allow a separate *_uncert* variable for each field, in case they are the same for all instances (would only be inserted if the value was otherwise null)?

Perhaps have special *uncertainties* and *measurement_methods* objects that defined default uncertainties:

```
uncertainties: {lat.m: 100, lon.m: 100, elevation.m: 10}
measurement_methods: {lat: "acoustic survey", lon: "acoustic survey", elevation:
  ↳ "acoustic survey", azimuth: "Ppol"}
```

for which I should probably keep the separation of uncertainties, measurement method and values

Remove instrumentation:operator field from schema

Already does nothing, but I kept it there to avoid breaking existing files

Make “operator” conform to StationXML standard

Currently flat:

```
operator:
  reference_name: # Just for link with campaign file, not StationXML
  full_name: # Change to "agency"
  contact: # Currently just a person's name
  phone_number:
  email:
  website:
```

Should be

```
operator:
  reference_name: # Just for link with campaign file, not StationXML
  agency:
  contact:
    name:
    agency:
    phone_number:
```

(continues on next page)

(continued from previous page)

email:
website:

The contact is a Person type and so can just be loaded from an “author” file (change “authors” to “persons”)

StationXML errors to fix in new version 0.111

The following errors, originally 6.3, 6.4, 8, 9, 10 and 12 in issue #3, will be addressed in v0.111. Most require a new version, because we will need to add a *base* field to the *instrumentation* object in order to bring station serial number and *channel_modifications* into this object:

1. No Serial Number shown for Station/Equipment
2. **LS5a is listed as 125 sps even though the network file says 62.5 sps**

And the following (possibly derived) errors:

1. Channel
 - a) Sensor serial number is given as “32793N” (direct from instrumentation files, doesn’t take into account OBS serial number)
2. Preamp/Datalogger/Equipment have no Serial Number

Moreover, I have to decide how to specify instrumentation-level serial numbers and configurations. Should I imitate *channel_modifications* with and *instrumentation_modifications* (or just *modifications*?) field, or should I allow specific common fields such as *serial_number* and *instrumentation_configuration* (or *configuration*)?

Other bugs that I didn’t fix in v0.110.12:

1. Channel
 1. Sensor
 1. Has installation date, removal date and three calibration dates, all after expt (Trillium T240)
 2. PreAmplifier
 1. BBOBS gain card description is not specific enough (1x? 0.225x?)
2. No Comment (or field) saying how the station was located
3. Equipment description does not include configuration-specific information (need *configuration_description* field?)

Should also make all tests work and maybe put my own tests back in

Generalize base-configuration-modification

schema files currently have no means to be configured (or modified?).

One solution would be to add to this level.

Another would to allow base, configuration and modification fields at any level: if “base” is specified, then configuration and or modification can be put at the same level.

DEVELOPER'S CORNER

10.1 Introduction

The sections here concern the coding of some of the more complicated aspects of obsinfo, to help with future modifications and debugging:

delay_correction explains how delay, offset, and delay_correction are implemented and related. This is actually a fairly simple concept but the basic principals must not be lost/forgotten.

channel_modification This is one of the most important/complicated sections because it allows the user to completely modify an instrumentation from the station level. Uses ObsMetadata

file_discovery File discovery allows people to put their databases where they want, even on the web. I don't know what the relationship is between the web address specified in the datapath parameter and the specific gitlab_* parameters, need to figure it out.

ObsMetadata A subclass of dict that allows one to change any value using a specified parallel dict, without modifying other values. Used for channel_modification, but I think also for other stuff.

This class also has all of the information file reading, validating and identifying, which I think should maybe belong to another class

testing An overview of the unittest files used to stabilize/verify the results when the code is updated

10.2 Fundamentals

This is a stub.

Explain here the fundamentals of obsinfo coding:

- **Need to follow StationXML as much as possible**
 - But to add other fields
 - And to eliminate redundancy
- Correlation and non-correlation of classes to StationXML objects and why
- **How an obsinfo file is parsed to obtain an obspy Network object**
 - Including non-standard field stuffing into comments.

10.3 Delay correction

As written in the tutorial, stage-level `corrections` are calculated by OBSINFO using the filter-level `offset`, stage-level `delay` and datalogger-level `delay_correction` fields:

- `offset` can be set at the Filter level. If not, it is set to zero. It is a required field for FIR filters (why not the other digital filters?)
- `delay` can be set at the Stage level. If it is not specified, it is set to the Stage Filter's `offset` divided by the Stage's `input_sample_rate`. This cannot be done when the Stage is first read, because the Stage's `input_sample_rate` usually depends on the preceding Stages.
- `correction` depends on the value of `datalogger.delay_correction`:
 - if `datalogger.delay_correction` is `None` (i.e., not specified in the information file, `correction` = `delay` in every stage
 - otherwise, `correction` = `datalogger.delay_correction` in the last stage and `correction` = `0` in every other stage.

this calculation can only be made after `delay` is calculated

The process for calculating `delay` and `correction` is as follows:

1. An attribute dictionary is passed to `Instrument._init()`, which
 1. creates `Sensor`, `Datalogger`, and `PreAmplifier` properties using their constructors
 - each one creates/contains a `ResponseStages()` property, but only the `Datalogger` object gets a `delay_correction` value
 - The `ResponseStages` constructor creates a list of `Stage`s. Each `Stage`:
 - contains a `delay` (usually not specified in the information file, in which case `delay` = `None`)
 - contains a `Filter` that contains an `offset` (set to `0` if not specified)
 2. calls `self.combine_response_stages()`, which combines the `Sensor`, `PreAmplifier` and `DataLogger` stages into a single, ordered list (`Sensor` stages, then `Preamplifier` and finally `Datalogger`)
 3. calls `self.integrate_response_stages()`, which loops through each stage, checking/creating continuity, calculating `delay` (in `Stage.calculate_delay()`) and setting `correction` = `delay` if `correction` is `None`
 4. creates a list of obspy stages and incorporates them into an obspy `Response` object (including calculating `Sensitivity`)

10.4 Channel Modifications

Channel modifications are at the core of customizing instrumentation in obsinfo. They allow one to completely modify an instrumentation's components, or specific aspects of those components (`advanced/chan_mods`). The process is complicated, here is an explanation of the philosophy, the codes involved and the potential bugs.

10.4.1 Files involved

The `channel_modifications` field is read in `station.py` and passed down to `instrumentation.py` then `channel.py`. Its handling is controlled by the `ObsMetaData` class defined in `obsmetadata.py`

in `station.schema.json`:

The structure of `channel_modifications` in the input file is specified in `station.schema.json`, where it is placed within the `station` object and its format is defined by `channel_mods`:

```
"channel_mods": {
  "type": "object",
  "description": "individual changes specified by das channel",
  "patternProperties": {
    "^[N, E, Z, 1, 2, 3, H, \\*]-[0-9, \\*]+$": { "$ref": "#/definitions/channel_
↪modif" }
  },
  "channel_modif": {
    "type": "object",
    "description": "DAS channel, modifications",
    "properties": {
      "orientation_code": { "$ref": "instrumentation.schema.json#/definitions/
↪orientation_code" },
      "datalogger" :      { "$ref": "datalogger.schema.json#/definitions/datalogger_wo_
↪required_fields" },
      "preamplifier":     { "$ref": "preamplifier.schema.json#/definitions/preamplifier_
↪wo_required_fields" },
      "sensor" :          { "$ref": "sensor.schema.json#/definitions/sensor_wo_required_
↪fields" },
      "datalogger_configuration" : { "type": "string" },
      "preamplifier_configuration": { "type": "string" },
      "sensor_configuration" :      { "type": "string" }
    },
    "additionalProperties" : false
  }
}
```

The definitions for sensor, datalogger and preamplifier are the same as for their initial definition, except they have less requirements. Change to something called `{component}_modifications`? And define each to allow a “base” field? Maybe also a “serial_number” and “configuration” field (could remove `datalogger_configuration`, `preamplifier_configuration` and `sensor_configuration`, though retain for now for compatibility)? And add “serial_number” at the `channel_modif` level, which would allow for a change to the base “equipment” `serial_number`.

There are several `_wo_required_fields` parameters, it looks like they could all be renamed to `_modifications`:

in station.py:

Initializing an “Station” class runs:

```
instr_dict = attributes_dict.get('instrumentation', None)
channel_modifs = attributes_dict.get('channel_modifications', {})
if instr_dict:
    self.instrumentation = Instrumentation(
        instr_dict, self.locations, start_date, end_date,
        channel_modifs, self.serial_number)
```

in instrumentation.py:

Initializing an “Instrumentation” class calls:

```
self.channels = [Channel(label, attributes, locations,
                        start_date, end_date,
                        self.equipment.obspy_equipment,
                        channel_default, channel_modifs)
                 for label, attributes in das_channels.items()]
```

where `das_channels` comes from `instr_dict['channels']` and `channel_default` comes from `das_channels['default']`

in channel.py:

Initializing a Channel class calls

```
selected_channel_modifs = self.get_selected_channel_modifs(
    self.channel_id_code, channel_modifs)
self.instrument = Instrument(self.das_channel, selected_channel_modifs)
```

and `get_selected_channel_modifs()` is:

in instrument.py:

Initializing an Instrument class calls

```
for ic_name in ('datalogger', 'sensor', 'preamplifier'):
    key = ic_name + '_configuration'
    config_selector = attributes.get_configured_element(key, channel_modifs)
    ic_obj = InstrumentComponent.dynamic_class_constructor(
        component, attributes_dict, channel_modifs, config_selector)
    setattr(self, ic_name, ic_obj) # equivalent to self.ic_name = ic_obj
```

where `channel_modifs` is the `selected_channel_modifs` in `Channel.__init__()`

`Obsmetadata.get_configured_element(key, channel_modifs={}, selected_config={}, default=None)` returns the value corresponding to `channel_modifs[key]`, else `selected_config[key]`, else `self[key]`, else default. Weird thing is that it won't return `selected_config[key]` if it's a dict or `ObsMetadata`, though this doesn't matter here.

Here it just returns the value in `channel_modifs['{inst_component}_configuration']` or `None`, as far as I can tell

I changed the code to:

```

for ic_type in ('datalogger', 'sensor', 'preamplifier'):
    ic_config_key = ic_type + '_configuration'
    if ic_type in channel_modifs:
        ic_modifs = channel_modifs[ic_type]
        # Pop out keywords
        config = ic_modifs.pop('configuration', None)
        sn = ic_modifs.pop('serial_number', None)
        base = ic_modifs.pop('base', None)
        # replace ic by channel_modifs[ic_type][base] if it exists
        if base is not None:
            logger.info('Replacing {ic_type}')
            attributes[ic_type] = base
        if sn is not None:
            if 'equipment' in ic_modifs:
                if 'serial_number' in ic_modifs['equipment']:
                    logger.warning('equipment:serial_number and serial_number specified,
↪equipment:serial_number overrides')
                else:
                    ic_modifs['equipment']['serial_number'] = sn
            else:
                ic_modifs['equipment'] = {'serial_number': sn}
        if config is not None:
            # For now, just replace v0.110 "*_configuration" keyword
            if ic_config_key in attributes:
                msg = 'attributes[{}]={} replaced by {}'.format(
                    ic_config_key, attributes[ic_config_key], ic_type, config)
                warnings.warn(msg)
                logger.warning(msg)
                attributes[ic_config_key] = config
            config_selector = attributes.get_configured_element(ic_config_key,
                                                                channel_modifs)
            ic_obj = InstrumentComponent.dynamic_class_constructor(
                ic_type, attributes, channel_modifs, config_selector)
            setattr(self, ic_type, ic_obj) # equivalent to self.ic_type = ic_obj

```

Seems to work

config_selector won't need ic_name + 'configuration' in v0.111, as the configuration will use the same keyword (configuration) for each InstrumentComponent field.

Finally, each component class has a dynamic_class_constructor() that modifies the ObsMetadata as requested and returns the result:

in `instrument_component.py`:

InstrumentComponent

This is the function called at first, it passes on to the specific `dynamic_class_constructor` class

```
@staticmethod
def dynamic_class_constructor(component_type, attributes_dict,
                             channel_modif={}, config_selector=''):
    """
    Creates an appropriate Instrument_component subclass (Sensor,
    Preamplifier, Datalogger) from an attributes_dict

    Args:
        component_type (str): type of component. Used for selection of
            adequate class.
        attributes_dict (dict or :class:`ObsMetadata`): component
            attributes
        channel_modif (dict or :class:`ObsMetadata`): channel modifications
            inherited from station
        delay_correction (float or None): delay correction in seconds:
            if a float: set last component stage to this, others to 0.
            if None: all component stage corrections are set = delay
        config_selector (str): selector of configuration coming from
            Instrument

    Returns:
        object of the adequate subclass
    """
    if not attributes_dict.get(component_type, None):
        if component_type == 'preamplifier': # Only preamps are optional
            return None
        else:
            msg = f'No {component_type}'
            warnings.warn(msg)
            logger.error(msg)
            raise TypeError(msg)

    selected_config = InstrumentComponent.retrieve_configuration(
        component_type, attributes_dict[component_type], config_selector)

    if component_type == 'datalogger':
        theclass = Datalogger
    elif component_type == 'sensor':
        theclass = Sensor
    elif component_type == 'preamplifier':
        theclass = Preamplifier
    else:
        msg = f'Unknown InstrumentComponent "{component_type}"'
        warnings.warn(msg)
        logger.error(msg)
        raise TypeError(msg)
    obj = theclass.dynamic_class_constructor(
```

(continues on next page)

(continued from previous page)

```

        ObsMetadata(attributes_dict[component_type]),
        channel_modif.get(component_type, {}),
        selected_config)
    return obj

```

Sensor

```

def dynamic_class_constructor(cls, attributes_dict, channel_modif={},
                             selected_config={}):
    """
    Create Sensor instance from an attributes_dict

    Args:
        attributes_dict (dict or :class:`ObsMetadata`): the base sensor
        channel_modif (dict or :class:`ObsMetadata`): channel modifications
            inherited from station
        selected_config (dict or :class:`ObsMetadata`): the configuration
            description that will override or complement default values
    Returns:
        (:class:`Sensor`)
    """

    if not attributes_dict:
        return None
    if not selected_config:
        # Avoids a syntax error in the yaml file: two consecutive labels
        # with no response stages
        selected_config = {}

    seed_dict = ObsMetadata(attributes_dict).get_configured_element(
        'seed_codes', channel_modif, selected_config, {})

    # The next line of code will totally override response states in
    # attribute_dict IF there is a selected_config with response_stages
    response_stages_list = attributes_dict.get_configured_element(
        'response_stages', {}, selected_config, None)

    response_stages = ResponseStages(
        response_stages_list,
        channel_modif.get('response_modifications', {}),
        selected_config.get('response_modifications', {}),
        None)

    obj = cls(Equipment(ObsMetadata(attributes_dict.get('equipment',
                                                         None)),
                        channel_modif.get('equipment', {}),
                        selected_config.get('equipment', {})),
              ObsMetadata(seed_dict).get_configured_element(
                  'band_base', channel_modif, selected_config, None),
              ObsMetadata(seed_dict).get_configured_element(

```

(continues on next page)

(continued from previous page)

```

        'instrument', channel_modif, selected_config, None),
        response_stages,
        attributes_dict.get_configured_element(
            'configuration_description', channel_modif,
            selected_config, ''))
    return obj

```

Preamplifier

```

@classmethod
def dynamic_class_constructor(cls, attributes_dict, channel_modif={},
                             selected_config={}):
    """
    Create Preamplifier instance from an attributes_dict

    Args:
        attributes_dict (dict or :class:`ObsMetadata`): attributes of
            component
        channel_modif (dict or :class:`ObsMetadata`): channel modifications
            inherited from station
        selected_config (dict or :class:`ObsMetadata`): the configuration
            description that will override or complement default values
    Returns:
        (:class:`Preamplifier`)
    """

    if not attributes_dict:
        return None
    if not selected_config:
        # Avoids a syntax error in the yaml file: two consecutive labels
        # with no response stages
        selected_config = {}

    # The next line of code will totally override response states in
    # attribute_dict IF there is a selected_config with response_stages
    response_stages_list = attributes_dict.get_configured_element(
        'response_stages', {}, selected_config, None)
    config_description = attributes_dict.get_configured_element(
        'configuration_description', channel_modif, selected_config, '')

    response_stages = ResponseStages(
        response_stages_list,
        channel_modif.get('response_modifications', {}),
        selected_config.get('response_modifications', {}),
        None)

    obj = cls(Equipment(ObsMetadata(attributes_dict.get('equipment',
                                                         None)),
                        channel_modif.get('equipment', {}),
                        selected_config.get('equipment', {})),

```

(continues on next page)

(continued from previous page)

```

        response_stages,
        config_description)

    return obj

```

Datalogger

```

def dynamic_class_constructor(cls, attributes_dict, channel_modif={},
                             selected_config={}):
    """
    Create Datalogger instance from an attributes_dict

    Args:
        attributes_dict (dict or :class:`ObsMetadata`): component
            attributes
        channel_modif (dict or :class:`ObsMetadata`): channel modifications
            inherited from station
        selected_config (dict or :class:`ObsMetadata`): the configuration
            description that will override or complement default values
    Returns:
        (:class:`Datalogger`)
    """
    if not attributes_dict:
        return None
    if not selected_config:
        # Avoids a syntax error in the yaml file: two consecutive labels
        # with no response stages
        selected_config = {}

    sample_rate = attributes_dict.get_configured_element(
        'sample_rate', channel_modif, selected_config, None)
    delay_correction = attributes_dict.get_configured_element(
        'delay_correction', channel_modif, selected_config, None)
    config_description = attributes_dict.get_configured_element(
        'configuration_description', channel_modif, selected_config, '')

    # The next line of code will totally override response states in
    # attribute_dict IF there is a selected_config with response_stages
    response_stages_list = attributes_dict.get_configured_element(
        'response_stages', {}, selected_config, None)

    response_stages = ResponseStages(
        response_stages_list,
        channel_modif.get('response_modifications', {}),
        selected_config.get('response_modifications', {}),
        delay_correction)

    obj = cls(Equipment(ObsMetadata(attributes_dict.get('equipment',
                                                         None)),
                       channel_modif.get('equipment', {}),

```

(continues on next page)

(continued from previous page)

```
        selected_config.get('equipment', {})),
        sample_rate,
        delay_correction,
        response_stages,
        config_description)

    return obj
```

This seems like some code duplication, but that’s not my problem right now and this part is too far downstream for me to worry about: the “magic” part (that currently screws up for importing a new sensor, for example) is at the *Instrument* level when `ObsMetadata.get_configured_element()` is called.

My first guess it that `get_configured_element()` is substituting all the new information into the existing `sensor`, but since the new information has no `configuration_default` the old value for this (and other elements) is left.

The simplest (and best?) solution would be to first substitute the entire `InstrumentComponent` `attributes_dict` when “sensor:base”, “datalogger:base” or “preamplifier:base” is specified. This implies that I “change” the input file specification to use “base”, but since it doesn’t function right now anyway, this should not break anything.

10.5 File discovery

This is a stub.

This file should explain how file discovery is implemented.

10.6 ObsMetadata

`ObsMetadata` is a dict subclass that allows some advanced operations useful for **obsinfo**. It has the additional methods:

- `get_configured_element()`: substitutes elements with selected or modification values
- `get_information_file_format()`: determines if an information file is in JSON or YAML format
- `get_information_file_type()`: determines the information file type (‘network’, etc)
- `list_valid_types()`
- `read_json_yaml()`
- `read_json_yaml_ref()`
- `read_json_yaml_ref_datapath()`
- `validate()`
- `validate_date()`
- `validate_dates()`

All but the first method seem like they should be in another class. I’m guessing Luis but them together to make sure that the output of any read was an `ObsMetadata` object, but that should be just as easy to do with another class...

This file is a stub.

10.7 Logging

This is a stub.

This file should explain how information is logged (results of all error checks, possibility of adding debugging levels...). I assume we will use Python's intrinsic logging module (already started, but I think only for writing to screen). Could take example from ppol, where I think I do fairly careful, multi-level logging. Have to be careful to specify how to setup logging in each command-line program (I assume there will be a standard, automatic level and then each one can activate a more active debugging)

10.8 Testing

Testing is very important to minimize bugs, but also to show how routines should be called. Proper testing avoids the all-to-frequent case of a code modification breaking previously working parts, if there are enough tests covering these parts. In general, every method of a class should have a test and, if you discover a bug that is not covered by a test, you should add a test for it.

We implement testing using the unittest Unit testing framework (<https://docs.python.org/3/library/unittest.html>). Ideally, every submodule should have a tests directory with a test.py file. This allows you to concentrate on local tests when you are modifying code, and should keep the test.py codes from being too huge.

A good practice when you modify a function is to see if that function has a test code that will see what you are changing and, if not, to add it and run the test before and after your modifications.

To run all tests, go into the obsinfo top directory and run

The basic structure of the test.py files is:

..code-block:: python

```
#!/usr/bin/env python # -- coding: utf-8 -- """ Test network and station classes """ import warnings from
pathlib import Path import unittest import difflib

# Third party imports

# obsinfo modules from obsinfo.network import (Station, Network)

warnings.simplefilter("once") warnings.filterwarnings("ignore", category=DeprecationWarning) verbose
= False

class NetworkTest(unittest.TestCase): """ Class of test methods for network and station objects

    Attributes:

        testing_path (str): path to datafiles to be tested aside from the examples
        level (str): level to be printed test (boolean): determines if this is test mode print_output
        (boolean): determines if this is print mode. Both can
        coexist.

    """

    def setUp(self, test=True, print_output=False, level=None): """ Set up default values and paths
        """ self.testing_path = Path(__file__).parent.joinpath("data")

        self.level = level self.test = test self.print_output = print_output

    def assertTextFilesEqual(self, first, second, msg=None):

        with open(first) as f: str_a = f.read()
```

```
    with open(second) as f: str_b = f.read()
    if str_a != str_b: first_lines = str_a.splitlines(True) second_lines = str_b.splitlines(True)
        delta = difflib.unified_diff(
            first_lines, second_lines, fromfile=first, tofile=second)
        message = ''.join(delta) if msg:
            message += " : " + msg
        self.fail("Multi-line strings are unequal:n" + message)
    def test_A(self): """Test of one method""" ...
    def test_B(self): """Test of another method""" ...
    def test_C(self): """Test of yet another method""" ...
    def suite(): return unittest.makeSuite(NetworkTest, 'test')
    if __name__ == '__main__': unittest.main(defaultTest='suite')
```

TECHNICAL DOCUMENTATION

11.1 Code Documentation: obsinfo package

11.1.1 Subpackages

obsinfo.network package

Contains the main classes from Network to Station

obsinfo.network module

obsinfo.network.network module

Network :class:

class obsinfo.network.network.**Network**(*attributes_dict=None, station_only=False*)

Bases: object

Network obsinfo: Equivalent to obspy/StationXML Network

campaign_ref

Type str

fdsn_code

Type str

fdsn_name

Type str

start_date

Type str with date format

end_date

Type str with date format

description

Type str

restricted_status

Type str

operator

Type object of Operator

stations

Type list of objects of *Station*

comments
Type list of str

extras
Type list of str

obspsy_network (
class~`obspsy.core.inventory.network.Network`): obspsy equivalent of this class

convert_notes_and_extras_to_obspsy()
Convert info file notes and extras to XML comments

to_obspsy()
Convert network object to obspsy object

Returns
class~`obspsy.core.inventory.network.Network`): corresponding obspsy Network

Return type
(

obsinfo.network.Network class

class obsinfo.network.**Network**(*attributes_dict=None, station_only=False*)

Bases: object

Network obsinfo: Equivalent to obspsy/StationXML Network

campaign_ref

Type str

fdsn_code

Type str

fdsn_name

Type str

start_date

Type str with date format

end_date

Type str with date format

description

Type str

restricted_status

Type str

operator

Type object of Operator

stations

Type list of objects of *Station*

comments

Type list of str

extras

Type list of str

```

obspsy_network (
    class~`obspsy.core.inventory.network.Network`): obspsy equivalent of this class

convert_notes_and_extras_to_obspsy()
    Convert info file notes and extras to XML comments

to_obspsy()
    Convert network object to obspsy object
    Returns
        class~`obspsy.core.inventory.network.Network`: corresponding obspsy Network
    Return type
        (

```

obsinfo.network.Station class

```

class obsinfo.network.Station(label, attributes_dict, station_only=False, stations_operator=None)
    Bases: object

    Station. Equivalent to obspsy/StationXML Station

    Methods convert info files to an instance of this class and convert the object to an obspsy object.

label
site
    Type str
start_date
    station start date
    Type str with date format
end_date
    station end date
    Type str with date format
location_code
    Type str
restricted_status
    status of station
    Type str
locations
    Type list of Location
location
    default location code of channels, corresponding to location_code
    Type Location
instrumentation (:class:`.Instrumentation` or list of
    Instrumentation):
processing
    attributes for clock correction processing
    Type list of objects of Processing
comments
    Type list of str
extras

```

Type list of str

obspy_station
Equivalent obspy object
Type obspy.core.inventory.station.Station

convert_comments_in_obspy()
Convert info file notes and extras to XML comments

to_obspy()
Convert station object to obspy object
Returns corresponding obspy Station object
Return type (obspy.core.inventory.station.Station)

obsinfo.network.Station class

class obsinfo.network.**Network**(*attributes_dict=None, station_only=False*)
Bases: object
Network obsinfo: Equivalent to obspy/StationXML Network

campaign_ref
Type str

fdsn_code
Type str

fdsn_name
Type str

start_date
Type str with date format

end_date
Type str with date format

description
Type str

restricted_status
Type str

operator
Type object of Operator

stations
Type list of objects of [Station](#)

comments
Type list of str

extras
Type list of str

obspy_network (
class~`obspy.core.inventory.network.Network`): obspy equivalent of this class

convert_notes_and_extras_to_obspy()
Convert info file notes and extras to XML comments

to_obspy()
Convert network object to obspy object

Returns

class:~obspy.core.inventory.network.Network): corresponding obspy Network

Return type

(

obsinfo.network.station module

Station Class

class obsinfo.network.station.**Station**(*label, attributes_dict, station_only=False, stations_operator=None*)

Bases: object

Station. Equivalent to obspy/StationXML Station

Methods convert info files to an instance of this class and convert the object to an *obspy* object.

label

site

Type str

start_date

station start date

Type str with date format

end_date

station end date

Type str with date format

location_code

Type str

restricted_status

status of station

Type str

locations

Type list of *Location*

location

default location code of channels, corresponding to *location_code*

Type *Location*

instrumentation (:class:`.Instrumentation` or list of *Instrumentation*):

processing

attributes for clock correction processing

Type list of objects of *Processing*

comments

Type list of str

extras

Type list of str

obspy_station

Equivalent obspy object

Type obspy.core.inventory.station.Station

convert_comments_in_obspsy()

Convert info file notes and extras to XML comments

to_obspsy()

Convert station object to obspsy object

Returns corresponding obspsy Station object

Return type (`obspsy.core.inventory.station.Station`)

obsinfo.network.processing module

Processing Class, holds list of data processing steps

class `obsinfo.network.processing.Processing`(*attributes*)

Bases: `object`

No equivalent class in obspsy/StationXML

Saves a list of Processing steps as strings For now, just stores the list. Will be converted to StationXML comments

processing_list

list of processing steps with attributes, either `linear_drift` or `leapsecond`

Type `list`

obsinfo.instrumentation package**obsinfo.instrumentation.instrumentation module**

Instrumentation class

class `obsinfo.instrumentation.instrumentation.Instrumentation`(*attributes_dict_or_list*, *locations*,
start_date, *end_date*,
channel_modifs=`{}`,
serial_number=`None`)

Bases: `object`

One or more Instruments. Part of an obspsy/StationXML Station

Methods convert info files to an instance of this class. No equivalent obspsy/StationXML class

A more detailed description the class and its attributes is found in XXX

equipment

Type `Equipment`

channels

list of channels (`Channel`)

Type `list`

__init__(*attributes_dict_or_list*, *locations*, *start_date*, *end_date*, *channel_modifs*=`{}`, *serial_number*=`None`)

Constructor

attributes_dict may contain a `configuration_selection` for the instrumentation and the corresponding configs for the components: `datalogger`, `preamplifier` and `sensor`

Parameters

- **attributes_dict_or_list** (*dict/ObsMetadata* or *list of same*) – instrumentation(s) attributes
- **locations** (*list*) – of `Locations`

- **start_date** (*str*) – start date
- **end_date** (*str*) – end date
- **channel_modifs** (dict or ObsMetadata) – modification of attributes per channel specified in stations
- **serial_number** (*str*) – instrumentation (OBS) serial number. Configures nothing for now

locations, start_date and end_date are inherited from the corresponding attributes in a station to fill out StationXML fields. It is assumed an instrumentation default location, start date and end_date are the same as its station's.

obsinfo.instrumentation.channel module

Channel, Instrument and Operator classes

class obsinfo.instrumentation.channel.**Channel**(*label, attributes, locations, start_date, end_date, equipment, channel_default={}, channel_modifs={}*)

Bases: object

Corresponds to StationXML/obspy Channel plus channel code

das_channel

represents a channel with defaults incorporated

Type ObsMetadata

orientation_code

orientation code for this channel (part of its ID)

Type str

location_code

location code for this channel (part of its ID)

Type str

location

represents location corresponding to location_code

Type Location

start_date

inherited from Station

Type str

end_date

inherited from Station

Type str

instrument

a sensor, a datalogger and an optional preamplifier

Type Instrument

orientation

Type Location

comments

Type str

extras

Type str

obspy_channel

The obspy equivalent of this object

Type `obsipy.core.inventory.channel.Channel`

__init__(*label, attributes, locations, start_date, end_date, equipment, channel_default={}, channel_modifs={}*)

Constructor

Parameters

- **label** (*str*) – label to identify channel. Different from the channel id code and only used in the information file.
- **of** (*attributes (dict/ObsMetadata object or list)*) – dicts//ObsMetadata objects): channel attributes
- **locations** (list of Location) –
- **start_date** – (str): using date format
- **end_date** – (str): using date format
- **channel_default** (dict or ObsMetadata) – default attributes to complement attributes specified in attributes
- **channel_modifs** (dict or ObsMetadata) – modification of attributes per channel specified in stations

channel_code(*sample_rate*)

Return channel code for a given sample rate.

Validates instrument and orientation codes according to FDSN specifications (for instruments, just the length). Channel codes specified by user are indicative and are refined using actual sample rate.

Parameters **sample_rate** (*float*) – instrumentation sampling rate (sps)

property channel_id_code

Uniquely identify channel through orientation and location code

format: {orientation}-{location}

Returns channel code

complete_channel(*das_channel, channel_default*)

Complement fields defined for each das channel

Takes all the fields defined for each das channel and complementa them with the channel_default fields. If das_channel key exists, leave the corresponding value. If not, add channel_default key/value

Parameters

- **das_channel** (dict or object of ObsMetadata) – channel to be completed with default
- **channel_default** (dict or object of ObsMetadata) – default that will complement attributes not present in das_channel

convert_notes_and_extras_to_obsipy()

Convert notes and extras to comments.

In StationXML comments are found at the channel level and up.

get_orientation_code(*orientation_dict*)

Get the orientation code from a dict that may include azimuth and dip

get_selected_channel_modifs(*id_code, channel_modifs*)

Select a channel_modification by id_code and channel label.

Parameters

- **id_code** (*str*) – {orientation_code}-{location_code} for the given channel
- **channel_modifs** (*dict*) – channel modifications to apply, keys can be, in order of priority:
”<RNT>-“<LOC>” or “<RNT>”: Apply to orientation <RNT> (and location <LOC>) “*“-“<LOC>”: Apply to all orientations with location code <LOC>

“<RNT>”-“*”: Apply to all locations with orientation
code <RNT>

”*” or “-”: Apply to all `id_codes`

Returns the selected channel modification

property seed_code

This is equivalent to channel code

to_obspsy(*equipment*)

Create obspsy Channel object

Returns *obspsy.core.inventory.channel.Channel*

Return type (~class)

obsinfo.instrumentation.instrument_component module

InstrumentComponent class and subclasses Sensor, Preamplifier, Datalogger. Equipment class

class `obsinfo.instrumentation.instrument_component.Datalogger`(*equipment*, *sample_rate*,
delay_correction=None,
response_stages=None,
config_description=")

Bases: *obsinfo.instrumentation.instrument_component.InstrumentComponent*

Datalogger Instrument Component.

Obspy equivalent is Datalogger, but only contains elements of Equipment

__init__(*equipment*, *sample_rate*, *delay_correction*=None, *response_stages*=None, *config_description*=")

Constructor

Parameters

- **equipment** (Equipment) – equipment attributes
- **sample_rate** (*float*) – sample rate of given configuration. Checked against actual sample rate
- **delay_correction** (*float* or *None*) – the delay correction of the component. If a float, it is applied to the last stage and the other stage corrections are set to 0. If None, each stage’s correction is set equal to its delay
- **response_stages** (ResponseStages) – channel modifications inherited from station
- **config_description** (*str*) – the configuration description that was selected, to complement component description

classmethod dynamic_class_constructor(*attributes_dict*, *channel_modif*={}, *selected_config*={})

Create Datalogger instance from an *attributes_dict*

Parameters

- **attributes_dict** (dict or ObsMetadata) – component attributes
- **channel_modif** (dict or ObsMetadata) – channel modifications inherited from station
- **selected_config** (dict or ObsMetadata) – the configuration description that will override or complement default values

Returns (*Datalogger*)

class `obsinfo.instrumentation.instrument_component.InstrumentComponent`(*equipment*,
response_stages=None,
config_description=")

Bases: object

InstrumentComponent class. Is superclass of all component classes. No obspy/StationXML equivalent, because they only specify the whole sensor+amplifier+datalogger system

equipment

Type Equipment

response_stages

Type ResponseStages

obspy_equipment

Type obspy_Equipment

config_description

description of configuration to be added to equipment description

Type str

__init__(*equipment, response_stages=None, config_description=""*)

Constructor. Invoked by dynamic_class_constructor.

Parameters

- **equipment** (Equipment) – equipment features of component
- **response_stages** (ResponseStages) – Response of component as a list of stages
- **config_description** (str) – description of configuration to be added to equipment description

static dynamic_class_constructor(*component_type, attributes_dict, channel_modif={}, config_selector=""*)

Builds a modified, configured Instrument_component

Passes onto subclass (Sensor, Preamplifier, Datalogger) methods after error checking.

Parameters

- **component_type** (str) – type of component ('sensor', 'datalogger' or 'preamplifier')
- **attributes_dict** (dict or ObsMetadata) – Instrument attributes (with component_type keys)
- **channel_modif** (dict or ObsMetadata) – channel modification inherited from station (with component_type keys)
- **config_selector** (str) – configuration selector

Returns object of the adequate subclass

retrieve_configuration(*attributes_dict, configuration_selection=None*)

Completes the component configuration.

A configuration selector can be defined in the instrumentation class, otherwise a config default is used. They may be used to select a configuration in the list of configuration definitions. If a configuration matches the selector configuration its attributes will override or add to existing attributes.

Parameters

- **comp_type** (str) – Type of instrument component. Used for messaging purposes only
- **attributes_dict** (dict or ObsMetadata) – attributes of component, hopefully including configuration_default
- **configuration_selection** (str) – label of configuration to be selected

Returns selected configuration, if found. Otherwise, empty dictionary

class obsinfo.instrumentation.instrument_component.**Preamplifier**(*equipment, response_stages=None, config_description=""*)

Bases: [obsinfo.instrumentation.instrument_component.InstrumentComponent](#)

Preamplifier Instrument Component. No obspy equivalent

Attributes: None

__init__(*equipment*, *response_stages*=None, *config_description*=")

Constructor

Parameters

- **equipment** (*Equipment*) – Equipment information
- **response_stages** (*ResponseStages*) – channel modifications inherited from station
- **config_description** (*str*) – the configuration description that was selected, added to equipment description

classmethod dynamic_class_constructor(*attributes_dict*, *channel_modif*={}, *selected_config*={})

Create Preamplifier instance from an *attributes_dict*

Parameters

- **attributes_dict** (dict or *ObsMetadata*) – attributes of component
- **channel_modif** (dict or *ObsMetadata*) – channel modifications inherited from station
- **selected_config** (dict or *ObsMetadata*) – the configuration description that will override or complement default values

Returns (*Preamplifier*)

```
class obsinfo.instrumentation.instrument_component.Sensor(equipment, seed_band_base_code,
                                                         seed_instrument_code,
                                                         response_stages=None,
                                                         config_description=")
```

Bases: *obsinfo.instrumentation.instrument_component.InstrumentComponent*

Sensor Instrument Component. No obspy equivalent

seed_band_base_code

SEED base code ("B" or "S") indicating instrument band.

Type str

seed_instrument_code

SEED instrument code

Type str

__init__(*equipment*, *seed_band_base_code*, *seed_instrument_code*, *response_stages*=None, *config_description*=")

Constructor

Parameters

- **equipment** (*Equipment*) – Equipment information
- **seed_band_base_code** (*str* (len 1)) – SEED base code ("B" or "S") indicating instrument band. Must be modified by obsinfo to correspond to output sample rate. Actual SEED base code is determined by FDSN standard <<http://docs.fdsn.org/projects/source-identifiers/en/v1.0/channel-codes.html>>
- **code** (*seed_instrument*) – SEED instrument code, determined by FDSN standard <<http://docs.fdsn.org/projects/source-identifiers/en/v1.0/channel-codes.html>>
- **response_stages** (*ResponseStages*) – channel modifications inherited from station
- **config_description** (*str*) – the configuration description that was selected, to complement component description

classmethod dynamic_class_constructor(*attributes_dict*, *channel_modif*={}, *selected_config*={})

Create Sensor instance from an *attributes_dict*

Parameters

- **attributes_dict** (dict or ObsMetadata) – attributes of component
- **channel_modif** (dict or ObsMetadata) – channel modifications inherited from station
- **selected_config** (dict or ObsMetadata) – the configuration description that will override or complement default values

Returns (*Sensor*)

obsinfo.instrumentation.response_stages module

ResponseStages and Stage classes

```
class obsinfo.instrumentation.response_stages.ResponseStages(attribute_list, channel_modif={},  
                                                             selected_config={},  
                                                             delay_correction=None)
```

Bases: object

An ordered list of *Stage*, representing a total response.

stages

Type list of objects of *Stage*

obspsy_stages (list of objects of different obspsy classes according
to the type of stage)

```
__init__(attribute_list, channel_modif={}, selected_config={}, delay_correction=None)  
Constructor
```

Method can be invoked with *attribute_list* as a single element of class *Stage* or as a list of them.

Parameters

- **attribute_list** (*list of dicts or dict*) – list of information file dictionaries for stages
- **channel_modif** (dict or *ObsMetadata*) – channel modifications inherited from station
- **selected_config** (dict or *ObsMetadata*) – configuration description to override or complement default values
- **delay_correction** (*float*) – used only for datalogger: the delay correction for the entire instrument

to_obspsy()

Return list of obspsy stage classes

Each class will have a different obspsy class

Returns list of objects of obspsy classes

```
class obsinfo.instrumentation.response_stages.Stage(attributes_dict, channel_modif_list={},  
                                                    selected_config={}, correction=None,  
                                                    sequence_number=- 1)
```

Bases: object

Stage is a discrete portion of the response of the instrument

name

name of stage, if any

Type str

description

description of stage

Type str

input_units
validated in schema
Type str

input_units_description
Type str

output_units
validated in schema
Type str

output_units_description
Type str

gain
value of gain
Type float

gain_frequency
frequency at which gain is measured
Type float

filter
Type object of Filter

stage_sequence_number
sequence number in total response, assigned later

input_sample_rate
input sample rate in sps
Type float

delay
delay in seconds of stage. If not present, will be calculated from offset from digital stages
Type float

decimation_factor
decimation factor of stage
Type float

correction
delay correction. Calculated from instrument delay correction
Type float

polarity
whether stage changes polarity
Type str, either “+” or “-”

calibration_date
calibration date of stage
Type str in date format

instrument_sensitivity
Not used, set to None. Sensitivity is calculated for the whole response.
Type float

__init__(*attributes_dict*, *channel_modif_list*={}, *selected_config*={}, *correction*=None, *sequence_number*=-1)

Parameters

- **attributes_dict** (dict or ObsMetadata) – attributes of component

- **channel_modif** (dict or *ObsMetadata*) – channel modifications inherited from station
- **selected_config** (dict or *ObsMetadata*) – the configuration description that will override or complement default values
- **correction** (*float*) – used only for datalogger, it's the delay correction of the whole instrument
- **sequence_number** (*int*) – sequence number, starting at 1. First assigned within component, then for the whole instrument response

calculate_delay()

Calculates delay

Delay is a function of filter offset for digital filters if not specified in info file

get_selected_modifications(modif_dict, key_code)

Select which channel modifications specified at station level apply to a given stage, with the stage number (WITHIN an instrument component) as key code

Parameters

- **modif_dict** (dict or *ObsMetadata*) – channel modifications inherited from station
- **key_code** (*str*) – key to the selected configuration using special regex format, e.g.: “*”, “[1,2,4]” or “[1-3]”

property output_sample_rate

Output sample rate is not specified but calculated from input sample rate and decimation factor

to_obspsy()

Return equivalent *obspsy.core.inventory.response* classes stages:

Possible stage classes:

- *PolesZerosResponseStage*
- *FIRResponseStage*
- *CoefficientsTypeResponseStage*
- *ResponseListResponseStage*
- *Response*

Returns object of one the above classes

obsinfo.instrumentation.filter module

Filter class and subclasses

class `obsinfo.instrumentation.filter.AD_Conversion`(*filter_type*, *input_full_scale*, *output_full_scale*, *offset=0*)

Bases: `obsinfo.instrumentation.filter.Coefficients`

AD_Conversion Filter (Flat Coefficients filter)

input_full_scale

Type float

output_full_scale

Type float

__init__(*filter_type*, *input_full_scale*, *output_full_scale*, *offset=0*)

Constructor

Parameters

- **filter_type** (*str*) – type of filter, necessarily ‘ADConversion’
- **input_full_scale** (*float*) – A/D’s input full scale (volts)

- **output_full_scale** (*float*) – corresponding output full scale (counts)
- **offset** (*int*) – number of samples that the filter offsets an impulse

classmethod dynamic_class_constructor(*filter_type*, *attributes_dict*, *channel_modif*={},
selected_config={}, *stage_id*=- 1)

Create AD_Conversion instance from an *attributes_dict*

Parameters

- **filter_type** (*str*) – type of filter, necessarily ‘ADConversion’
- **attribute_list** (*dict or list of dicts*) – information file dictionaries for stages
- **channel_modif** (*dict or ObsMetadata*) – channel modifications inherited from station
- **selected_config** (*dict or ObsMetadata*) – configuration description to override or complement default values
- **stage_id** (*int*) – id of corresponding stage. Used for reporting only

Returns (ADConversion)

class obsinfo.instrumentation.filter.**Analog**(*filter_type*, *offset*=0, *normalization_frequency*=1.0)

Bases: [obsinfo.instrumentation.filter.PolesZeros](#)

Analog Filter (Flat PolesZeros filter)

__init__(*filter_type*, *offset*=0, *normalization_frequency*=1.0)

Constructor

Parameters

- **filter_type** (*str*) – necessarily ‘Analog’
- **offset** (*int*) –
- **normalization_frequency** (*float*) –

classmethod dynamic_class_constructor(*filter_type*, *attributes_dict*, *channel_modif*={},
selected_config={}, *stage_id*=- 1, *gain_frequency*=1.0)

Create Analog instance from an *attributes_dict*

Parameters

- **filter_type** (*str*) – type of filter, necessarily ‘Analog’
- **attribute_list** (*dict or list of dicts*) – information file dictionaries for stages
- **channel_modif** (*dict or ObsMetadata*) – channel modifications inherited from station
- **selected_config** (*dict or ObsMetadata*) – the configuration description that will override or complement default values
- **stage_id** (*int*) – id of corresponding stage. Used for reporting only
- **gain_frequency** (*float*) – frequency at which gain was specified.

Returns ([Analog](#))

class obsinfo.instrumentation.filter.**Coefficients**(*filter_type*, *transfer_function_type*,
numerator_coefficients, *denominator_coefficients*,
offset=0)

Bases: [obsinfo.instrumentation.filter.Filter](#)

Coefficients Filter Class

transfer_function_type

one of “ANALOG (RADIANS/SECOND)”, “ANALOG (HERTZ)”, or “DIGITAL”

Type str

numerator_coefficients

Type list of floats

denominator_coefficients

Type list of floats

__init__(*filter_type*, *transfer_function_type*, *numerator_coefficients*, *denominator_coefficients*, *offset*=0)
Constructor

Parameters

- **filter_type** (*str*) – type of filter, necessarily ‘Coefficients’
- **transfer_function_type** (*str*) – one of “ANALOG (RADIANS/SECOND)”, “ANALOG (HERTZ)”, or “DIGITAL”
- **numerator_coefficients** (*list*) – of floats
- **denominator_coefficients** (*list*) – of floats
- **offset** (*int*) – samples that an impulse is offset by the filter
- **filter_type** – type of filter, necessarily ‘Coefficients’
- **offset** –

classmethod dynamic_class_constructor(*filter_type*, *attributes_dict*, *channel_modif*={}, *selected_config*={}, *stage_id*=- 1)

Create Coefficients instance from an attributes_dict

Parameters

- **filter_type** (*str*) – type of filter, necessarily ‘Coefficients’
- **attribute_list** (*dict or list of dicts*) – information file dictionaries for stages
- **channel_modif** (*dict or ObsMetadata*) – channel modifications inherited from station
- **selected_config** (*dict or ObsMetadata*) – the configuration description that will override or complement default values
- **stage_id** (*int*) – id of corresponding stage. Used for reporting only

Returns (*Coefficients*)

class obsinfo.instrumentation.filter.Digital(*filter_type*, *offset*=0)

Bases: [obsinfo.instrumentation.filter.Coefficients](#)

Digital Filter (Flat Coefficients filter)

__init__(*filter_type*, *offset*=0)

Constructor

Parameters

- **filter_type** (*str*) – type of filter, necessarily ‘Digital’
- **offset** (*int*) –

classmethod dynamic_class_constructor(*filter_type*, *attributes_dict*, *channel_modif*={}, *selected_config*={}, *stage_id*=- 1)

Create Digital instance from an attributes_dict

Parameters

- **filter_type** (*str*) – type of filter, necessarily ‘Digital’
- **attribute_list** (*dict or list of dicts*) – information file dictionaries for stages
- **channel_modif** (*dict or ObsMetadata*) – channel modifications inherited from station
- **selected_config** (*dict or ObsMetadata*) – the configuration description that will override or complement default values
- **stage_id** (*int*) – id of corresponding stage. Used for reporting only

Returns (*Digital*)

class obsinfo.instrumentation.filter.FIR(*filter_type*, *symmetry*, *coefficients*, *coefficient_divisor*, *offset*=0, *stage_id*=- 1)

Bases: [obsinfo.instrumentation.filter.Filter](#)

FIR Filter

symmetry

filter symmetry, one of “EVEN”, “ODD”, “NONE”

Type str

coefficients

Type list of floats

coefficient_divisor

Type float

__init__(*filter_type, symmetry, coefficients, coefficient_divisor, offset=0, stage_id=-1*)

Constructor

Parameters

- **filter_type** (*str*) – type of filter, necessarily ‘FIR’
- **symmetry** (*str*) – filter symmetry, one of “EVEN”, “ODD”, “NONE”
- **coefficients** (*list*) – floats
- **coefficient_divisor** (*float*) –
- **offset** (*int*) – # of samples filter will offset an impulse
- **stage_id** (*int*) – id of corresponding stage. Used for reporting only

classmethod dynamic_class_constructor(*filter_type, attributes_dict, channel_modif={}, selected_config={}, stage_id=-1*)

Create FIR instance from an attributes_dict

Parameters

- **filter_type** (*str*) – type of filter, necessarily ‘FIR’
- **attribute_list** (*dict or list of dicts*) – information file dictionaries for stages
- **channel_modif** (*dict or ObsMetadata*) – channel modifications inherited from station
- **selected_config** (*dict or ObsMetadata*) – the configuration description that will override or complement default values
- **stage_id** (*int*) – id of corresponding stage. Used for reporting only

Returns (*FIR*)

class obsinfo.instrumentation.filter.**Filter**(*type='PolesZeros', offset=0*)

Bases: object

Filter is superclass of all filter classes

type

filter type

Type str

offset

offset is samples to skip at the beginning of a signal in digital filters

Type int

__init__(*type='PolesZeros', offset=0*)

Constructor

Parameters

- **type** (*str*) – filter type
- **offset** (*int*) – samples that an impulse is offset by a digital filter

static dynamic_class_constructor(*attributes_dict, channel_modif={}, selected_config={}, stage_id='-1', gain_frequency=1.0*)

Creates an appropriate Filter subclass from an attributes_dict

Parameters

- **attributes_dict** (*dict or list of dicts*) – information file dictionaries for stages

- **channel_modif** (dict or *ObsMetadata*) – channel modifications inherited from station
- **selected_config** (dict or *ObsMetadata*) – the configuration description that will override or complement default values
- **stage_id** (*int*) – id of corresponding stage. Used for reporting only
- **gain_frequency** (*float*) – frequency at which gain was specified. Used for PoleZeros Normalization factor/frequency

Returns object of the adequate filter subclass

Return type (*Filter*)

Raises (**TypeError**) – if filter type is not valid

```
class obsinfo.instrumentation.filter.PolesZeros(filter_type, transfer_function_type='LAPLACE
(RADIANS/SECOND)', poles=[], zeros=[],
normalization_frequency=1.0,
normalization_factor=None, offset=0, stage_id=- 1)
```

Bases: *obsinfo.instrumentation.filter.Filter*

PolesZeros filter

transfer_function_type

one of 'LAPLACE (RADIANS/SECOND)', 'LAPLACE (HERTZ)', 'DIGITAL (Z-TRANSFORM)'

Type str

poles

Type list of complex numbers

zeros

Type list of complex numbers

normalization_frequency

Type float

normalization_factor

Type float

```
__init__(filter_type, transfer_function_type='LAPLACE (RADIANS/SECOND)', poles=[], zeros=[],
normalization_frequency=1.0, normalization_factor=None, offset=0, stage_id=- 1)
```

Constructor

Parameters

- **filter_type** (*str*) –
- **transfer_function_type** (*str*) – one of 'LAPLACE (RADIANS/SECOND)', 'LAPLACE (HERTZ)', 'DIGITAL (Z-TRANSFORM)'
- **poles** (*list*) – complex numbers specified as [a, b]
- **zeros** (*list*) – complex numbers specified as [a, b]
- **normalization_frequency** (*float*) –
- **normalization_factor** (*float*) –
- **offset** (*int*) – makes no sense for an analog filter
- **stage_id** (*int*) – id of corresponding stage. Used for reporting only

calc_normalization_factor(*stage_id=- 1, debug=False*)

Calculate the normalization factor for a given set of poles-zeros

The norm factor A0 is calculated such that

$$A0 * \frac{\text{sequence_product_over_n}(s - \text{zero_n})}{\text{sequence_product_over_m}(s - \text{pole_m})} == 1$$

(continues on next page)

(continued from previous page)

```
for s_f = i*2pi*f if the transfer function is Laplace in radians
    i*f if the transfer function is Laplace in Hertz
```

There is no calculation for the digital z-transform

Returns normalization factor as a float or None if not Laplace

```
classmethod dynamic_class_constructor(filter_type, attributes_dict, channel_modif={},
                                       selected_config={}, stage_id=- 1, gain_frequency=1.0)
```

Create PolesZeros instance from an attributes_dict

Parameters

- **filter_type** (*str*) – type of filter, necessarily ‘PolesZeros’
- **attribute_list** (*dict or list of dicts*) – information file dictionaries for stages
- **channel_modif** (*dict or ObsMetadata*) – channel modifications inherited from station
- **selected_config** (*dict or ObsMetadata*) – the configuration description that will override or complement default values
- **stage_id** (*int*) – id of corresponding stage. Used for reporting only
- **gain_frequency** (*float*) – frequency at which gain was specified.

Returns (*PolesZeros*)

```
class obsinfo.instrumentation.filter.ResponseList(filter_type, response_list, offset=0)
```

Bases: *obsinfo.instrumentation.filter.Filter*

ResponseList Filter

response_list

list of responses instead of function coefficients

Type list of floats

```
__init__(filter_type, response_list, offset=0)
```

Constructor

Args” filter_type (*str*): type of filter, necessarily ‘ResponseList’ response_list (*list*): floats specifying response offset (*int*): samples that an impulse is offset by the filter

```
classmethod dynamic_class_constructor(filter_type, attributes_dict, channel_modif={},
                                       selected_config={}, stage_id=- 1)
```

Create Response List instance from an attributes_dict

Parameters

- **filter_type** (*str*) – type of filter, necessarily ‘ResponseList’
- **attribute_list** (*dict or list of dicts*) – information file dictionaries for stages
- **channel_modif** (*dict or ObsMetadata*) – channel modifications inherited from station
- **selected_config** (*dict or ObsMetadata*) – the configuration description that will override or complement default values
- **stage_id** (*int*) – id of corresponding stage. Used for reporting only

Returns (*ResponseList*)

obsinfo.instrumentation.equipment module

InstrumentComponent class and subclasses Sensor, Preamplifier, Datalogger. Equipment class

[illegible]

Bases: obspy.core.inventory.util.Equipment

Equipment.

Equivalent to :class: obspy.core.inventory.util.Equipment

type

Type str

channel_modif

Type str

selected_config

Type str

description

Type str

manufacturer

Type str

model

Type str

vendor

Type str

serial_number

Type str

installation_date

Type str in date format

removal_date

Type str in date format

calibration_dates

Type str in date format

resource_id

Type str

obspsy_equipment

Type class *obspy.core.inventory.equipmentEquipment*

```
__init__(attributes_dict, channel_modif={}, selected_config={})
```

Constructor

Parameters

- **attributes_dict** (dict or ObsMetadata) – attributes of component, hopefully including configuration_default
- **channel_modif** (dict or ObsMetadata) – channel modifications inherited from station
- **selected_config** (sdict or ObsMetadata) – configuration dict selected at instrument_component level

to_obspsy()

Convert an equipment (including the equipment description in components) to its obspy object

Returns (`obspy.core.inventory.util.Equipment`)

obsinfo.instrumentation.location module

Location classe

class `obsinfo.instrumentation.location.Location`(*attributes_dict*)

Bases: `object`

Location class.

latitude

station latitude (degrees N)

Type `float`

longitude

station longitude (degrees E)

Type `float`

elevation

station elevation (meters above sea level)

Type `float`

uncertainties_m

'lat', 'lon', 'elev' in METERS

Type `dict`

geology

site geology

Type `str`

vault

vault type

Type `str`

depth_m

depth of station beneath surface (meters)

Type `float`

localisation_method

method used to determine position

Type `str`

obspy_latitude

latitude as an *obspy* object

obspy_longitude

longitude as an *obspy* object

__init__(*attributes_dict*)

Create Location object and assign attributes from *attributes_dict*. Validate required location attributes exist
Convert to obspy longitude and latitude

Parameters *attributes_dict* (`dict` or object of *ObsMetadata*) – location information

static get_location_from_code(*locations, code, type, label*)

Obtain from the locations dictionary one location by code (key). Raise exception if not found

property obspy_elevation

Returns elevation as obspy `FloatWithUncertaintiesFixedUnit` object

property obspy_latitude

Returns obspy Latitude object

property obspy_longitude

Returns obspy Longitude object

obsinfo.instrumentation.orientation module

Orientation class

class obsinfo.instrumentation.orientation.**Orientation**(*attributes_dict, channel_modifs, polarity*)

Bases: object

Class for sensor orientations. No channel modifs. Cannot change orientation as it is part of the channel identifiers. Azimuth and dip can be changed Orientation is coded by *FDSN standard* <<http://docs.fdsn.org/projects/source-identifiers/en/v1.0/channel-codes.html>>

azimuth

azimuth, clockwise from north

Type degrees**azimuth_uncertainty****Type** degrees**dip**

dip, -90 to 90: positive=down, negative=up

Type degrees**type dip_uncertainty****Type** degrees**__init__**(*attributes_dict, channel_modifs, polarity*)

Constructor

Seismometer If a positive voltage corresponds to upward motion (typical vertical seismometer), dip = -90° (up). If positive voltage corresponds to downward motion (typical vertical geophone), dip = 90°

Hydrophone If a positive voltage corresponds to a positive (compressional) pressure, then dip = -90°. If a positive voltage corresponds to a decrease in pressure, then dip = 90°

Parameters **attributes_dict** (dict or *ObsMetadata*) – operator information

get_value_with_uncertainty(*info_list*)

Validate that info_list is a 2 member list.

Parameters **info_list** (*list of strings or floats*) – two numbers, a value and an uncertainty

Returns the two members

Return type (list)

Raises **ValueError** if not a list of two floats or strings that can be converted to floats

obsinfo.obsMetadata package

obsinfo.obsMetadata.obsmetadata module

obsinfo information file routines, contained in superclass ObsMetadata for generality

class obsinfo.obsMetadata.obsmetadata.**ObsMetadata**(*args, **kwargs)

Bases: dict

__init__(*args, **kwargs)

Constructor, simply create a dict with positional and keyword arguments

Parameters

- **args** (*list) – positional arguments
- **kwargs** (**dict) – keyword arguments

convert_to_obsmetadata()

Make all contained dictionaries objects of :class: *.ObsMetadata*

get_configured_element(key, channel_modification={}, selected_configuration={}, default=None)

Substitute an ObsMetadata element with selected or modification values

This is the heart of *obsinfo* flexibility, which permits changes in configuration or even in individual attributes via channel modifications.

Returns the value corresponding to key in the following priority:

- 1) channel_modification[key]
- 2) selected_configuration[key]
- 3) self[key] d) default

Parameters

- **key** (str) – key to values from the respective dictionaries
- **channel_modification** (dict or *ObsMetadata*) – modifications specified at station level
- **selected_configuration** (dict or *ObsMetadata*) – component-level configuration
- **default** (dict or *ObsMetadata*) – default value

Returns the highest priority value for ‘key’

Return type value (*ObsMetadata*)

get_information_file_format()

Determines if the information file is in JSON or YAML format.

Assumes that the filename is “.{*FORMAT*}”

Parameters **filename** (str) – filename to determine the type of

Returns format

Raises (**ValueError**) – on unknown format

get_information_file_type()

Determines the type of a file.

Assumes that the filename is “.{*TYPE*}.{*SOMETHING*}”

Parameters **filename** (str) – filename to determine the type of

Returns file type

Return type (str)

Raises **ValueError** –

static is_valid_type(type)

Returns true if input is a valid schema type

list_valid_types()

Returns a list of valid information file types

Returns list of valid information file types

static read_info_file(filename, datapath, remote=False, validate=True, format=None, verbose=False)

Reads an information file

Parameters

- **filename** (*str*) – filename
- **datapath** (*Datapath*) – stores list of directories to search info files
- **validate** (*bool*) – validate before reading
- **remote** (*bool*) – whether to use absolute/relative path locally or OBSINFO_DATAPATH
- **format** (*str*) – format type (“YAML” or “JSON”)

Returns JSON or YAML parsed info files

Return type (*ObsMetadata*)

read_json_yaml(format=None)

Reads a JSON or YAML file. Does NOT use jsonReference DEPRECATED.

DEPRECATED. Not being used by any obsinfo method or function. Kept for compatibility

Parameters

- **filename** (*str*) – filename
- **format** (*str*) – format type, either “YAML” or “JSON”

Returns JSON or YAML parsed information files

Return type (dict)

Raises

- (**JSONDecodeError**) – problem with JSON read
- (**FileNotFoundError**) – file not found
- (**IOError**) – File input/output error

read_json_yaml_ref(datapath, format=None)

Reads a JSON or YAML file using jsonReference

Like read_json_yaml_ref, but does not look for files in OBSINFO_DATAPATH \$ref within the data files without absolute or relative path will be still looked for in OBSINFO_DATAPATH

Parameters

- **filename** (*str*) – filename
- **datapath** (*Datapath*) – object to store list of
- **dummy**. (*directories to search info files. Used as a*) –
- **format** (*str*) – format type, either “YAML” or “JSON”

Returns JSON or YAML parsed information files

Return type (dict)

Raises

- (**JSONDecodeError**) – problem with JSON read
- (**FileNotFoundError**) – file not found
- (**IOError**) – File input/output error

read_json_yaml_ref_datapath(datapath, format=None)

Reads a JSON or YAML file using jsonReference using OBSINFO_DATAPATH

Parameters

- **filename** (*str*) – filename
- **datapath** (*Datapath*) – list of directories to search for info files
- **format** (*str*) – format type, either “YAML” or “JSON”

Returns JSON or YAML parsed information files

Return type (dict)

Raises

- (**JSONDecodeError**) – problem with JSON read

- **(FileNotFoundError)** – file not found
- **(IOError)** – File input/output error

validate(*schemas_path*, *filename*, *remote=False*, *format=None*, *file_type=None*, *verbose=False*, *schema_file=None*, *quiet=False*, *dp=None*)

Validates a YAML or JSON file against schema

Parameters

- **schemas_path** (*str*) – path to schema files
- **filename** (*str* or *path-like*) – name of file to validate
- **remote** (*bool*) – whether to search for filename in a remote repository
- **format** (*str*) – format of information file: “JSON” or “YAML”
- **file_type** (*str*) – type of info file: “network”, “station”, “instrumentation”, “datalogger”, “preamplifier”, “sensor”, “stage”, “filter”
- **verbose** (*bool*) – Print progression of validation steps with filenames
- **schema_file** (*str*) – name (without path) of schema file
- **quiet** (*bool*) – Deprecated. Conserved for compatibility. Cancels verbose.
- **dp** (*Datapath*) – datapath for information files. If None, defaults to values stored in .obsinfo

If file_type and/or format are not provided, tries to figure them out from the filename, which should end with “{FILETYPE}.{FORMAT}”

validate_date()

Reformats an individual date

Uses regular expressions to match known dates, either in UTC date format, in UTC date and time format or in regular dd-mm-yyyy format. If only two digits of year specified, assumes 21st century. The separator can be either “/” or “-”

Parameters **date** (*str*) – a date in a given format

Returns a reformatted date as string or None if no value

Return type (*str*)

Raises (**ValueError**) if date is unrecognizable –

validate_dates()

Converts dates to a standard format

Parameters **dates** (*list*) – dates as strings

Returns formatted dates as strings

Return type (*list*)

obsinfo.main package

obsinfo.main.makeStationXML module

Main functions for obsinfo-makeStationXML

Creates obsinfo objects starting with a network object in a hierarchy which strongly follows the hierarchy of StationXML files. Then converts the objects to a StationXML file using obspy.

obsinfo.main.makeStationXML.get_first_author(*info_dict*)

Get info file first author info and return it in a string.

Used to fill the author attribute in the Inventory object of obspy.

Parameters

- **info_dict** – The name to use.
- **state** (*bool*.) – Current state to be in.

Returns *str* – first author first name and last name and contact information

`obsinfo.main.makeStationXML.init_logging()`

Create or open a rotating logging file and add it to ObsinfoConfiguration

Returns object of Logger class

`obsinfo.main.makeStationXML.main(argv=None, dp=None)`

Entry point for obsinfo-makeStationXML.

- 1) Setups status variables from command line arguments.
- 2) Read yaml or json file invoking `read_info_file`, which returns a dictionary. Optionally validates dictionary schema.
- 3) Creates obsinfo objects starting from network object from the dictionary
- 4) Converts these to StationXML using obspsy libraries.

Manages all uncaught exceptions.

Parameters

- **argv** (*list*) – list of command-line arguments to pass to ArgumentParser. If None, will use `sys.argv`
- **dp** (*Datapath*) – Datapath object specifying where to look for files. If None, will use values specified in `.obsinfo`

`obsinfo.main.makeStationXML.retrieve_arguments(argv, datapath)`

Retrieve arguments from command line. Setup several status variables and get information file name

Parameters

- **argv** (*list*) – command line arguments. If None, uses `sys.argv`
- **datapath** (*Datapath*) – Object containing paths to find repository files, read from `.obsinfo`

Returns All status variables and the information file name.

Return type `args` (Namespace)

obsinfo.main.print module

obsinfo.main.setupObsinfo module

Application to configure variables used by obsinfo in file `.obsinfo` and to copy examples and templates out of the distribution directories

`obsinfo.main.setupObsinfo.build_datapath_list(args, remote_repository)`

Build datapath as a list of directories to search, similar to the `PATH` environment variable.

1. Add `working_directory` as first option if it exists
2. Next add local repository first if it exists (will be first if no working directory specified)
3. Finally add destination, where standard examples lie. This won't be used if `-x` is on
4. Final directory will be the remote one unless `-v` (invert) is specified

`obsinfo.main.setupObsinfo.retrieve_arguments()`

Parse command line arguments

Returns arguments, both status and directory names

Return type `parse_args` (dict)

`obsinfo.main.setupObsinfo.setup_obsinfo()`

Entry point to configure variables in file `.obsinfo`

and to copy examples and templates out of the distribution directories

`obsinfo.main.setupObsinfo.validate_arguments(args)`

Validate that directories in command line arguments exist. Raise an exception if not.

Parameters **args** (*dictionary of strings*) – directory with command line argument, but status and directories

:raises `OSError`, `ValueError`

obsinfo.main.validate module

Main functions for obsinfo-validate.

obsinfo strongly follows the hierarchy of StationXML files.

Module contains both class `ValidateObsinfo` and a main entry point for obsinfo-validate

class obsinfo.main.validate.ValidateObsinfo

Bases: object

Contains methods to validate each level of information files.

datapath

Store datapath list(s)

Type /.Datapath

verbose

Prints progress messages

Type bool

remote

Indicates file in command line argument is to be found using datapath

Type bool

debug

Print more detailed messages.

Type bool

assertTextFilesEqual(*first, second, msg=None*)

Compares two text files

Parameters

- **first** (*str*) – First file to compare
- **second** (*str*) – Second file to compare
- **msg** (*str*) – Message to print in case of failure

setUp(*verbose=True, remote=False, debug=False*)

Set up status variables according to .obsinfo and command line arguments.

Parameters

- **verbose** (*bool*) – Print several progress messages
- **remote** (*bool*) – Find file in command line argument using datapath
- **debug** (*bool*) – Print more detailed messages and enable traceback of exceptions

Returns: self

validate_all_components()

Validate all information files in each components (sensor, preamplifier, datalogger) subdirectory as given by datapath

validate_all_filters()

Validate all filter files in datapath/<component>/responses/filters/

validate_all_instrumentations()

Validate all instrumentation files in datapath/instrumentation/

validate_all_networks()

Validate all network files in datapath/network/

validate_all_stages()

Validate all stage files in datapath/<component>/responses/

validate_files(*files, filetype*)

Validate all files of a given type

Parameters

- **files** (Path) – Full paths of files (including wildcards)
- **filetype** (str) – information file type

validate_filters_in_directory(dir)

Validate all information files in filter directory.

Parameters **dir** (str) – directory where filter files reside

validate_single_file(info_file, filetype)

Validate a single obsinfo file.

Parameters

- **info_file** (str) – info file to validate. No assumptions made about path.
- **filetype** (str) – the information file type

Returns True if validated, False if not

Return type (bool)

validate_stages_in_directory(dir)

Validate all information files in stage directory as given by datapath.

Parameters **ldir** (str) – directory where stage files reside

obsinfo.main.validate.init_logging()

Create or open a rotating logging file and add it to ObsinfoConfiguration

Returns object of Logger class

obsinfo.main.validate.main()

Entry point for obsinfo-validate.

- 1) Setups status variables from command line arguments.
- 2) Validates file according to file type contained in name
- 3) Manages all uncaught exceptions

obsinfo.main.validate.retrieve_arguments()

Retrieve arguments from command line. Setup several status variables and get information file name

Returns dictionary object with all status variables and information file name.

obsinfo.misc package**obsinfo.misc.configuration module**

class obsinfo.misc.configuration.Singleton(cls)

Bases: object

Class to implement singleton pattern design in Python

Instance()

obsinfo.misc.const module

Exit values as constants as per UNIX BSD standard

obsinfo.misc.discoveryfiles module

class obsinfo.misc.discoveryfiles.**Datapath**(datapath=None)

Bases: object

Class to discover where information files are stored.

datapath_list

directories where information files will be searched, in sequence

Type list of str

infofiles_path

same as datapath_list, used by validate, kept for compatibility

Type list of str

validate_path

one unique path to validation schemas

Type str

static add_frag(path, frag)

Add the path and the frag to restore a partial or complete uri

Parameters

- **path** (str) – path portion of uri, possibly with other elements but without frag
- **frag** (str) – fragment portion of uri

Returns (str)L path with frag

build_datapath(file)

Create list of directories which may have data or schemas

- 1) If the file path is absolute, return the file itself.
- 2) If path starts by ./ or ../ complete to an absolute path using working directory
- 3) If the file has no prefix discover whether file exists in the datapath list. Use the first found file.

Parameters **file** (str or path) – filename of file to be found

Returns found file as string

Raises **FileNotFoundError** –

obsinfo.misc.printobs module

Functions to print obsinfo objects

class obsinfo.misc.printobs.**PrintObs**

Bases: object

Collection of methods to print obsinfo objects at different levels of depth. All methods are static.

None

static print_component(obj, level='all')

Prints component information and continues if level is not “component” or “channel”.

If level is not “channel” detailed equipment information is not printed.

Parameters **level** (str) – determines to which level the *obsinfo* object will be printed

static print_instrumentation(obj, level='all')

Prints instrumentation information and continues if level is not “instrumentation”.

If level is “response” or “all” response stages information will be printed (if it exists). Recall at this point all the response is gathered in `instrument.response_stages.sgates` If level is “all” or “filter” (i.e. not “response”) all information will be printed (only filter is left at this point...)

Parameters **level** (str) – determines to which level the *obsinfo* object will be printed

static print_network(*obj*, *level*='all')

Prints network information and continues if level is not “network”.

Parameters **level** (*str*) – determines to which level the *obsinfo* object will be printed

static print_station(*obj*, *level*='all')

Prints station information and continues if level is not “station”.

Parameters **level** (*str*) – determines to which level the *obsinfo* object will be printed

obsinfo.misc.remoteGitLab module

class obsinfo.misc.remoteGitLab.gitLabFile

Bases: object

Provide the methods to use the gitlab API to read a remote file and decode it

None

static get_gitlab_file(*uri*, *read_file*=True)

Get the file content pointed at by uri and decode it.

Uses b64 first to get the remote file and convert to a byte string, and then utf-8 to convert to regular string.

Parameters

- **uri** (*string* or *path-like*) – uri to read
- **(bool)** (*read_file*) – If true, reads the content. If not, simply checks if the file exists

Returns read content

Return type (*str*)

Raises: FileNotFoundError, ValueError

static isRemote(*file*)

Checks if scheme means file is remote.

Parameters **file** (*str*) – filename to be checked, with complete uri

Returns boolean. True if remote, False otherwise

obsinfo.misc.yamlref module

Module to read and parse YAML or JSON files, locally or remotely (gitlab only)

jsonref with YAML reading added.

copied directly from jsonref v0.2, with added routines `_yaml_load` and `_yaml_loads` replacing `json.load` and `json.loads` Added/modified lines are marked “# WCC”

class obsinfo.misc.yamlref.JsonLoader(*store*=(), *cache_results*=True)

Bases: object

Provides a callable which takes a URI, and returns the loaded JSON referred to by that URI. Uses `requests` if available for HTTP URIs, and falls back to `urllib`. By default it keeps a cache of previously loaded documents.

Attributes:

- **store:** pre-populated dictionary matching URIs to loaded JSON documents used as cache
- **cache_results** (boolean): if this is set to false, the internal cache of
- loaded JSON documents is not used

__init__(*store*=(), *cache_results*=True)

get_json_or_yaml(*uri*, ***kwargs*)

Open either a local file, if uri scheme is `file` or a remote one, calling a gitlab method which implements the gitlab API (version 4)

Parameters

- **uri** (*path-like object, string or byte string*) – The URI of the JSON or YAML document to load
- **kwargs** (*dict*) – Keyword arguments passed to `json.loads()`

Returns dictionary of parsed YAML or JSON formats

Raises `FileNotFoundError`, `IOError`, `OSError`

```
class obsinfo.misc.yamlref.JsonRef(refobj, base_uri="", loader=None, jsonschema=False,
                                   load_on_repr=True, _path=(), _store=None, datapath=None)
```

Bases: `proxytypes.LazyProxy`

A lazy loading proxy to the dereferenced data pointed to by a JSON Reference object.

Attributes:

- `__reference__`: dictionary object referenced to by a `$ref`
- `base_uri`: object of type `Path` which is used to build the full uri
- `loader`: a loader object (a callable) such as `JsonLoader`, to load a JSON or YAML file/string
- `jsonschema` = Flag to turn on JSON Schema mode
- **`load_on_repr` = If set to False, `repr()` call on a `JsonRef` object will not cause the reference to be loaded if it hasn't already. (defaults to True)**
- `path` = list of string keywords: keywords of different `$ref` in lists or dictionaries
- `store` = dictionary of cached objects used to prevent reading files over again
- `datapath` = object of `Datapath`, stores directories to search for files

```
__init__(refobj, base_uri="", loader=None, jsonschema=False, load_on_repr=True, _path=(), _store=None,
         datapath=None)
```

callback()

Callback from `proxytypes.LazyProxy`.

Resolves the pointer (part of the dictionary read from the info file) that is incorporated instead of `$ref`.
Updates `base_uri`

Returns

the fragment portion of the `base_doc`, which has already had its `$ref` replaced.

property full_uri

This method/property returns the full uri to reference a `$ref` object. It's the heart of how a `datapath` is used to either access a local or remote (gitlab) file. All schema files are supposed to be local, part of the obsinfo distribution

Returns updated full uri

Raises `ValueError`

```
classmethod replace_refs(obj, _recursive=False, **kwargs)
```

Returns a deep copy of `obj` with all contained JSON reference objects replaced with `JsonRef` instances.

Parameters

- **`obj`** (*JSONRef or collection*) – If a JSON reference object, a `JsonRef` instance will be created. If not, a deep copy of it will be created with all contained JSON reference objects replaced by `JsonRef` instances
- **`recursive`** (*bool*) – Process `$ref` recursively
- **`kwargs`** (*dict*) – Keyword arguments passed to `json.loads()`

Returns the information in `$ref` file

Return type `obj():class:JsonRef`

Raises `TypeError`, `ValueError` through `JsonRef` object creation –

kwargs include:

`base_uri` (`Path`): URI to resolve relative references against. Can be remote (`https://`) or local (`file://`) This is how `datapath` is implemented

`datapath` (`Datapath`): object to implement file discovery in a list of directories

`loader` (loader object such as `JsonLoader`): Callable that takes a URI and returns the parsed

JSON (defaults to global `jsonloader`, a *JsonLoader* instance)

jsonschema: Flag to turn on **JSON Schema mode**, which means the file is a schema file. This makes 'id' keyword to change the `base_uri` for references contained within the object, such as `$ref: '#/definitions'`

type jsonschema boolean

load_on_repr: If set to **False**, `repr()` call on a *JsonRef* object will not cause the reference to be loaded if it hasn't already. (defaults to **True**)

type load_on_repr boolean

resolve_pointer(*document, pointer*)

Resolve a json pointer *pointer* within the referenced document.

Parameters

- **document** – the referent document
- **pointer** (*str*) – a json pointer URI fragment to resolve within it

Returns part of document dictionary pointed at by pointer

exception `obsinfo.misc.yamlref.JsonRefError`(*message, reference, uri="", base_uri="", path=()*, *cause=None*)

Bases: Exception

Create exception for JSONRef

Attributes:

- **message** (*str*): message to print with exception
- **reference** (*str*): reference where exception occurred
- **uri** (*str* or path-like): uri of file being processed
- **base_uri**: (*str* or path-like): `base_uri` (complement) of file being processed
- **path** = list of string keywords: keywords of different `$ref` in lists or dictionaries
- **cause** (*str*): cause of exception

__init__(*message, reference, uri="", base_uri="", path=()*, *cause=None*)

`obsinfo.misc.yamlref.dump`(*obj, fp, **kwargs*)

Serialize *obj* as a JSON formatted stream to file-like *fp*

JsonRef objects will be dumped as the original reference object they were created from.

Parameters

- **obj** – Object to serialize
- **fp** (*File-like object*) –
- **kwargs** (*dict*) – Keyword arguments for `json.dumps()`

`obsinfo.misc.yamlref.dumps`(*obj, **kwargs*)

Serialize *obj*, which may contain *JsonRef* objects, to a JSON formatted string. *JsonRef* objects will be dumped as the original reference object they were created from.

Parameters

- **obj** – Object to serialize
- **kwargs** (*dict*) – Keyword arguments for `json.dumps()`

Returns dumped string

`obsinfo.misc.yamlref.load`(*fp, base_uri="", loader=None, jsonschema=False, load_on_repr=True, datapath=None, **kwargs*)

Drop in replacement for `json.load()`, where JSON references are proxied to their referent data.

The difference between `load` and `loads` is that the first uses a file-like object and the second a string.

Parameters

- **fp** (*File-like object*) – File-like object containing JSON document
- **base_uri** (object of type `Path`) – URI to resolve relative references against. Can be remote (<https://>) or local(<file://>) This is how `datapath` is implemented

- **datapath** (object of type `Datapath`) – object to implement file discovery in a list of directories
- **loader** (a loader object such as `JsonLoader`) – Callable that takes a URI and returns the parsed JSON (defaults to global `jsonloader`, a `JsonLoader` instance)
- **jsonschema** (*boolean*) – Flag to turn on `JSON Schema mode`, which means the file is a schema file. This makes ‘id’ keyword to change the `base_uri` for references contained within the object, such as \$ref: ‘#/definitions’
- **load_on_repr** (*boolean*) – If set to `False`, `repr()` call on a `class:JsonRef` object will not cause the reference to be loaded if it hasn’t already. (defaults to `True`)
- **kwargs** (*dict*) – This function takes any of the keyword arguments from `JsonRef.replace_refs()`. Any other keyword arguments will be passed to `_yaml_load()`

Returns dictionary of parsed YAML or JSON formats

`obsinfo.misc.yamlref.load_uri(uri, base_uri=None, datapath=None, loader=None, jsonschema=False, load_on_repr=True)`

Load JSON data from uri instead of file-like object or string. with JSON references proxied to their referent data. Not used in obsinfo.

Parameters

- **uri** (*string or path-like object*) – URI to fetch the JSON from
- **base_uri** (*Path*) – URI to resolve relative references against. Can be remote (<https://>) or local (<file://>) This is how `datapath` is implemented
- **datapath** (`Datapath`) – object to implement file discovery in a list of directories
- **loader** (loader object such as `JsonLoader`) – Callable that takes a URI and returns the parsed JSON (defaults to global `jsonloader`, a `JsonLoader` instance)
- **jsonschema** (*bool*) – Flag to turn on `JSON Schema mode`, which means the file is a schema file. This makes ‘id’ keyword to change the `base_uri` for references contained within the object, such as \$ref: ‘#/definitions’
- **load_on_repr** (*bool*) – If set to `False`, `repr()` call on a `class:JsonRef` object will not cause the reference to be loaded if it hasn’t already. (defaults to `True`)

Returns parsed YAML or JSON formats

Return type `newref` (`dict`)

`obsinfo.misc.yamlref.loads(s, base_uri="", loader=None, jsonschema=False, load_on_repr=True, datapath=None, recursive=True, **kwargs)`

Drop in replacement for `json.loads()`, where JSON references are proxied to their referent data.

The difference between `load` and `loads` is that the first uses a file-like object and the second a string.

Parameters

- **s** (*str*) – Input JSON document
- **base_uri** (*Path*) – URI to resolve relative references against. Can be remote (<https://>) or local (<file://>) This is how `datapath` is implemented
- **datapath** (`Datapath`) – object to implement file discovery in a list of directories
- **loader** (loader object such as `JsonLoader`) – Callable that takes a URI and returns the parsed JSON (defaults to global `jsonloader`, a `JsonLoader` instance)
- **jsonschema** (*bool*) – Flag to turn on `JSON Schema mode`, which means the file is a schema file. This makes ‘id’ keyword to change the `base_uri` for references contained within the object, such as \$ref: ‘#/definitions’
- **load_on_repr** (*bool*) – If set to `False`, `repr()` call on a `JsonRef` object will not cause the reference to be loaded if it hasn’t already. (defaults to `True`)
- **kwargs** (*dict*) – Any of the keyword arguments from `JsonRef.replace_refs()`. Any other keyword arguments will be passed to `_yaml_load()`

Returns decoded JSON or YAML

Return type `dic` (`dict`)

obsinfo.tests package

obsinfo.tests.remoteGithub module

`obsinfo.tests.remoteGithub.constructURL(user='404', repo_name='404', path_to_file='404', url='404')`

obsinfo.tests.test module

obsinfo.tests.testmain module

obsinfo.addons package

obsinfo.addons.LCHEAPO module

Write extraction script for LCHEAPO instruments (proprietary to miniseed)

`obsinfo.addons.LCHEAPO.process_script(network_code, station, station_dir, distrib_dir, input_dir='.',
output_dir='miniseed_basic', include_header=True)`

Writes script to transform raw OBS data to miniSEED

Parameters

- **network_code** (*str*) – FDSN network_code
- **(station)** – class: ~obsinfo.Station): the station to process
- **station_dir** (*str*) – the base directory for the station data
- **distrib_dir** (*str*) – directory where the lcheapo executables and property files are found
- **input_dir** (*str*) – directory beneath station_dir for LCHEAPO data
- **output_dir** (*str*) – directory beneath station_dir for basic miniseed]
- **include_header** (*bool*) – include the header that sets up paths (should be done once)

obsinfo.addons.SDPCHAIN module

Generate scripts needed to go from basic miniSEED to data center ready

`obsinfo.addons.SDPCHAIN.process_script(station, station_dir, distrib_dir='/opt/sdpchain',
input_dir='miniseed_basic', corrected_dir='miniseed_corrected',
extra_commands=None, include_header=True,
SDS_uncorr_dir='SDS_uncorrected',
SDS_corr_dir='SDS_corrected',
SDS_combined_dir='SDS_combined')`

Writes OBS data processing script using SDPCHAIN software

Parameters

- **station** – an obsinfo.station object
- **station_dir** – base directory for the station data
- **input_dir** – directory beneath station_dir for input (basic) miniseed data ['miniseed_basic']
- **corrected_dir** – directory beneath station_dir for output (corrected) miniseed data ['miniseed_corrected']
- **SDS_corr_dir** – directory beneath station_dir in which to write SDS structure of corrected data (ideally ../SOMETHING if ms2sds could write all to the same directory)
- **SDS_uncorr_dir** – directory beneath station_dir in which to write SDS structure of uncorrected data (ideally ../SOMETHING if ms2sds could write all to the same directory)

- **include_header** – whether or not to include the bash script header ('#!/bin/bash') at the top of the script [True]
- **distrib_dir** – Base directory of sdpchain distribution ['/opt/sdpchain']

The sequence of commands is:

1. optional proprietary format steps (proprietary format -> basic miniseed, separate)
2. optional extra_steps (any cleanup needed for the basic miniseed data, should either overwrite the existing data or remove the original files so that subsequent steps only see the cleaned data)
3. ms2sds on basic miniseed data
4. leap-second corrections, if necessary
5. msdrift (creates drift-corrected miniseed)

obsinfo.addons.LC2SDS module

Write a script to convert LCHEAPO data to SDS* using the lcheapo** python package

Includes clock drift and leap-second correction Script is a BASH shell script *SDS = SeisComp Data Structure **THIS PROGRAM DOES NOT CREATE DATA-CENTER QUALITY DATA:

- drift correction is calculated for each day, not each record
- does not set drift correction record header flags
- does not fill in record header time_correction field

`obsinfo.addons.LC2SDS.process_script(network_code, stations, station_data_path, input_dir='.', output_dir='.', include_header=True, no_drift_correct=False)`

Writes script to transform raw OBS data to SeisComp Data Structure

Parameters

- **network_code** (*str*) – FDSN network_code
- **stations** (list of *Station*) – the stations to process
- **station_data_path** (*str*) – the base directory beneath the station data dirs
- **input_dir** (*str*) – directory beneath station_dir for LCHEAPO data
- **output_dir** (*str*) – directory beneath station_dir for SDS directory
- **include_header** (*bool*) – include the header that sets up paths (should be done once)
- **no_drift_correct** (*bool*) – Do NOT drift correct

11.1.2 obsinfo.print_version module

`obsinfo.print_version.main()`

11.1.3 obsinfo.version module

CHANGELOG

- **0.110.11:**
 - Added obsinfo-makescripts-LC2SDS
- **0.110.12:**
 - Fixed bug in clock correction reading (leapsecond and drift)
 - Made obsinfo-makescripts-LC2SDS write to campaign directory by default
 - **Fixed some (not all) StationXML bugs:**
 - * Made clock correction Comments using JSON
 - * Removed *restrictedStatus='unknown'* (invalid) from StationXML
 - * Fixed bugs in Decimation filters and stage #s
 - * Standardized uncertainties in Poles, Zeros, Elevation, Dip, Azimuths
 - **Bugs that will probably have to wait for version 0.111:**
 - * instrumentation serial number and configurations not accounted for
- **0.110.13:**
 - Made all obsinfo-test cases work
 - Added “serial_number” to station level (temporary fix before v 0.111)
 - Updated JSON schema to draft07, allowing more precise/compact information on mutiple-choice errors (e.g., different types of filter)
- **0.110.14:**
 - Added StationXML test case and made it work
 - made `python -m unittest discover` work
- **0.110.15:**
 - corrected JSON validation schema for `orientation_code`
 - improved `azimuth.deg` and `dip.deg` schema definitions
 - Fixed a bug in reporting schema validation errors (in *obsmetadata.py*) introduced when shifting to draft07
 - Reduced a bug when a jsonref points to an inexistent pointer within a file
 - Uses Python 3.8-dependent syntax despite only requiring Python 3!
- **0.110.16:**

- Clean up information file validation, removing many redundancies in main/validate.py and hopefully fixing bug where schema files lacking “.schema.yaml” are searched for
- Requires Python 3.7 syntax and only uses 3.7-dependent syntax
- *post2*: streamlines *obsinfo-makescripts_LC2SDS*’s output
- **0.110.17:**
 - **Improvements to channel_modifications:**
 - * Make reading a new sensor, datalogger or preamplifier work
 - * Enable shortcuts for entering serial number
 - * Updated documentation
 - Added developer and information_file documentation in channel_modifications/
 - Moved tests up to united top-level directory
 - *offset* can be a non-integer
- **0.110.18:**
 - Added datacite information file
 - Added schema and validation of datacite, author, location_base, network_info and operator information files
 - Cleaned up some information file validation glitches

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

O

- [obsinfo.addons.LC2SDS](#), 227
- [obsinfo.addons.LCHEAPO](#), 226
- [obsinfo.addons.SDPCHAIN](#), 226
- [obsinfo.instrumentation.channel](#), 199
- [obsinfo.instrumentation.equipment](#), 212
- [obsinfo.instrumentation.filter](#), 206
- [obsinfo.instrumentation.instrument_component](#), 201
- [obsinfo.instrumentation.instrumentation](#), 198
- [obsinfo.instrumentation.location](#), 213
- [obsinfo.instrumentation.orientation](#), 214
- [obsinfo.instrumentation.response_stages](#), 204
- [obsinfo.main.makeStationXML](#), 217
- [obsinfo.main.setupObsinfo](#), 218
- [obsinfo.main.validate](#), 219
- [obsinfo.misc.configuration](#), 220
- [obsinfo.misc.const](#), 220
- [obsinfo.misc.discoveryfiles](#), 221
- [obsinfo.misc.printobs](#), 221
- [obsinfo.misc.remoteGitLab](#), 222
- [obsinfo.misc.yamlref](#), 222
- [obsinfo.network](#), 193
- [obsinfo.network.network](#), 193
- [obsinfo.network.processing](#), 198
- [obsinfo.network.station](#), 197
- [obsinfo.obsMetadata.obsmetadata](#), 215
- [obsinfo.print_version](#), 227
- [obsinfo.tests.remoteGithub](#), 226
- [obsinfo.version](#), 227

Symbols

- `__init__()` (*obsinfo.instrumentation.channel.Channel* method), 200
 - `__init__()` (*obsinfo.instrumentation.equipment.Equipment* method), 212
 - `__init__()` (*obsinfo.instrumentation.filter.AD_Conversion* method), 206
 - `__init__()` (*obsinfo.instrumentation.filter.Analog* method), 207
 - `__init__()` (*obsinfo.instrumentation.filter.Coefficients* method), 208
 - `__init__()` (*obsinfo.instrumentation.filter.Digital* method), 208
 - `__init__()` (*obsinfo.instrumentation.filter.FIR* method), 209
 - `__init__()` (*obsinfo.instrumentation.filter.Filter* method), 209
 - `__init__()` (*obsinfo.instrumentation.filter.PolesZeros* method), 210
 - `__init__()` (*obsinfo.instrumentation.filter.ResponseList* method), 211
 - `__init__()` (*obsinfo.instrumentation.instrument_component.InstrumentComponent* method), 201
 - `__init__()` (*obsinfo.instrumentation.instrument_component.InstrumentComponent* method), 202
 - `__init__()` (*obsinfo.instrumentation.instrument_component.Preamplifier* method), 203
 - `__init__()` (*obsinfo.instrumentation.instrument_component.Sensor* method), 203
 - `__init__()` (*obsinfo.instrumentation.instrumentation.Instrumentation* method), 198
 - `__init__()` (*obsinfo.instrumentation.location.Location* method), 213
 - `__init__()` (*obsinfo.instrumentation.orientation.Orientation* method), 214
 - `__init__()` (*obsinfo.instrumentation.response_stages.ResponseStages* method), 204
 - `__init__()` (*obsinfo.instrumentation.response_stages.Stage* method), 205
 - `__init__()` (*obsinfo.misc.yamlref.JsonLoader* method), 222
 - `__init__()` (*obsinfo.misc.yamlref.JsonRef* method), 223
 - `__init__()` (*obsinfo.misc.yamlref.JsonRefError* method), 224
 - `__init__()` (*obsinfo.obsMetadata.obsMetadata.ObsMetadata* method), 215
- ## A
- AD_Conversion* (class in *obsinfo.instrumentation.filter*), 206
 - add_frag()* (*obsinfo.misc.discoveryfiles.Datapath* static method), 221
 - Analog* (class in *obsinfo.instrumentation.filter*), 207
 - assertTextFilesEqual()* (*obsinfo.main.validate.ValidateObsinfo* method), 219
 - azimuth* (*obsinfo.instrumentation.orientation.Orientation* attribute), 214
 - azimuth_uncertainty* (*obsinfo.instrumentation.orientation.Orientation* attribute), 214
- ## B
- build_datapath()* (*obsinfo.misc.discoveryfiles.Datapath* method), 221
 - build_datapath_list()* (in module *obsinfo.main.setupObsinfo*), 218
- ## C
- calc_normalization_factor()* (*obsinfo.instrumentation.filter.PolesZeros* method), 210
 - calculate_delay()* (*obsinfo.instrumentation.response_stages.Stage* method), 206
 - calibration_date* (*obsinfo.instrumentation.response_stages.Stage* attribute), 205
 - calibration_dates* (*obsinfo.instrumentation.equipment.Equipment* attribute), 212
 - callback()* (*obsinfo.misc.yamlref.JsonRef* method), 223

- campaign_ref (*obsinfo.network.Network* attribute), 194, 196
- campaign_ref (*obsinfo.network.network.Network* attribute), 193
- Channel (class in *obsinfo.instrumentation.channel*), 199
- channel_code() (*obsinfo.instrumentation.channel.Channel* method), 200
- channel_id_code (*obsinfo.instrumentation.channel.Channel* property), 200
- channel_modif (*obsinfo.instrumentation.equipment.Equipment* attribute), 212
- channels (*obsinfo.instrumentation.instrumentation.Instrumentation* attribute), 198
- coefficient_divisor (*obsinfo.instrumentation.filter.FIR* attribute), 209
- Coefficients (class in *obsinfo.instrumentation.filter*), 207
- coefficients (*obsinfo.instrumentation.filter.FIR* attribute), 209
- comments (*obsinfo.instrumentation.channel.Channel* attribute), 199
- comments (*obsinfo.network.Network* attribute), 194, 196
- comments (*obsinfo.network.network.Network* attribute), 194
- comments (*obsinfo.network.Station* attribute), 195
- comments (*obsinfo.network.station.Station* attribute), 197
- complete_channel() (*obsinfo.instrumentation.channel.Channel* method), 200
- config_description (*obsinfo.instrumentation.instrument_component.InstrumentComponent* attribute), 202
- constructURL() (in module *obsinfo.tests.remoteGithub*), 226
- convert_comments_in_obspsy() (*obsinfo.network.Station* method), 196
- convert_comments_in_obspsy() (*obsinfo.network.station.Station* method), 197
- convert_notes_and_extras_to_obspsy() (*obsinfo.instrumentation.channel.Channel* method), 200
- convert_notes_and_extras_to_obspsy() (*obsinfo.network.Network* method), 195, 196
- convert_notes_and_extras_to_obspsy() (*obsinfo.network.network.Network* method), 194
- convert_to_obsmetadata() (*obsinfo.obsMetadata.obsmetadata.ObsMetadata* method), 215
- correction (*obsinfo.instrumentation.response_stages.Stage* attribute), 205
- ## D
- das_channel (*obsinfo.instrumentation.channel.Channel* attribute), 199
- Datalogger (class in *obsinfo.instrumentation.instrument_component*), 201
- Datapath (class in *obsinfo.misc.discoveryfiles*), 221
- datapath (*obsinfo.main.validate.ValidateObsinfo* attribute), 219
- datapath_list (*obsinfo.misc.discoveryfiles.Datapath* attribute), 221
- debug (*obsinfo.main.validate.ValidateObsinfo* attribute), 219
- decimation_factor (*obsinfo.instrumentation.response_stages.Stage* attribute), 205
- delay (*obsinfo.instrumentation.response_stages.Stage* attribute), 205
- denominator_coefficients (*obsinfo.instrumentation.filter.Coefficients* attribute), 207
- depth_m (*obsinfo.instrumentation.location.Location* attribute), 213
- description (*obsinfo.instrumentation.equipment.Equipment* attribute), 212
- description (*obsinfo.instrumentation.response_stages.Stage* attribute), 204
- description (*obsinfo.network.Network* attribute), 194, 196
- description (*obsinfo.network.network.Network* attribute), 193
- Digital (class in *obsinfo.instrumentation.filter*), 208
- dir (*obsinfo.instrumentation.orientation.Orientation* attribute), 214
- dump() (in module *obsinfo.misc.yamlref*), 224
- dumps() (in module *obsinfo.misc.yamlref*), 224
- dynamic_class_constructor() (*obsinfo.instrumentation.filter.AD_Conversion* class method), 207
- dynamic_class_constructor() (*obsinfo.instrumentation.filter.Analog* class method), 207
- dynamic_class_constructor() (*obsinfo.instrumentation.filter.Coefficients* class method), 208
- dynamic_class_constructor() (*obsinfo.instrumentation.filter.Digital* class method), 208
- dynamic_class_constructor() (*obsinfo.instrumentation.filter.Filter* static method), 209
- dynamic_class_constructor() (*obsinfo.instrumentation.filter.FIR* class method), 209

- `dynamic_class_constructor()` (*obsinfo.instrumentation.filter.PolesZeros* class method), 211
- `dynamic_class_constructor()` (*obsinfo.instrumentation.filter.ResponseList* class method), 211
- `dynamic_class_constructor()` (*obsinfo.instrumentation.instrument_component.Datalogger* class method), 201
- `dynamic_class_constructor()` (*obsinfo.instrumentation.instrument_component.InstrumentComponent* static method), 202
- `dynamic_class_constructor()` (*obsinfo.instrumentation.instrument_component.Preamplifier* class method), 203
- `dynamic_class_constructor()` (*obsinfo.instrumentation.instrument_component.Sensor* class method), 203
- ## E
- `elevation` (*obsinfo.instrumentation.location.Location* attribute), 213
- `end_date` (*obsinfo.instrumentation.channel.Channel* attribute), 199
- `end_date` (*obsinfo.network.Network* attribute), 194, 196
- `end_date` (*obsinfo.network.network.Network* attribute), 193
- `end_date` (*obsinfo.network.Station* attribute), 195
- `end_date` (*obsinfo.network.station.Station* attribute), 197
- `Equipment` (class in *obsinfo.instrumentation.equipment*), 212
- `equipment` (*obsinfo.instrumentation.instrument_component.InstrumentComponent* attribute), 202
- `equipment` (*obsinfo.instrumentation.instrumentation.Instrumentation* attribute), 198
- `extras` (*obsinfo.instrumentation.channel.Channel* attribute), 199
- `extras` (*obsinfo.network.Network* attribute), 194, 196
- `extras` (*obsinfo.network.network.Network* attribute), 194
- `extras` (*obsinfo.network.Station* attribute), 195
- `extras` (*obsinfo.network.station.Station* attribute), 197
- ## F
- `fdsn_code` (*obsinfo.network.Network* attribute), 194, 196
- `fdsn_code` (*obsinfo.network.network.Network* attribute), 193
- `fdsn_name` (*obsinfo.network.Network* attribute), 194, 196
- `fdsn_name` (*obsinfo.network.network.Network* attribute), 193
- `Filter` (class in *obsinfo.instrumentation.filter*), 209
- `filter` (*obsinfo.instrumentation.response_stages.Stage* attribute), 205
- `FIR` (class in *obsinfo.instrumentation.filter*), 208
- `full_uri` (*obsinfo.misc.yamlref.JsonRef* property), 223
- ## G
- `gain` (*obsinfo.instrumentation.response_stages.Stage* attribute), 205
- `gain_frequency` (*obsinfo.instrumentation.response_stages.Stage* attribute), 205
- `geology` (*obsinfo.instrumentation.location.Location* attribute), 213
- `get_configured_element()` (*obsinfo.obsMetadata.obsmetadata.ObsMetadata* method), 215
- `get_first_author()` (in module *obsinfo.main.makeStationXML*), 217
- `get_gitlab_file()` (*obsinfo.misc.remoteGitLab.gitLabFile* static method), 222
- `get_information_file_format()` (*obsinfo.obsMetadata.obsmetadata.ObsMetadata* method), 215
- `get_information_file_type()` (*obsinfo.obsMetadata.obsmetadata.ObsMetadata* method), 215
- `get_json_or_yaml()` (*obsinfo.misc.yamlref.JsonLoader* method), 222
- `get_location_from_code()` (*obsinfo.instrumentation.location.Location* static method), 213
- `get_orientation_code()` (*obsinfo.instrumentation.channel.Channel* method), 200
- `get_selected_channel_modifs()` (*obsinfo.instrumentation.channel.Channel* method), 200
- `get_selected_modifications()` (*obsinfo.instrumentation.response_stages.Stage* method), 206
- `get_value_with_uncertainty()` (*obsinfo.instrumentation.orientation.Orientation* method), 214
- `gitLabFile` (class in *obsinfo.misc.remoteGitLab*), 222
- ## I
- `infofiles_path` (*obsinfo.misc.discoveryfiles.Datapath* attribute), 221
- `init_logging()` (in module *obsinfo.main.makeStationXML*), 217
- `init_logging()` (in module *obsinfo.main.validate*), 220
- `input_full_scale` (*obsinfo.instrumentation.filter.AD_Conversion* attribute), 206
- `input_sample_rate` (*obsinfo.instrumentation.response_stages.Stage* attribute), 205

- `input_units` (*obsinfo.instrumentation.response_stages.Stage* attribute), 204
- `input_units_description` (*obsinfo.instrumentation.response_stages.Stage* attribute), 205
- `installation_date` (*obsinfo.instrumentation.equipment.Equipment* attribute), 212
- `Instance()` (*obsinfo.misc.configuration.Singleton* method), 220
- `instrument` (*obsinfo.instrumentation.channel.Channel* attribute), 199
- `instrument_sensitivity` (*obsinfo.instrumentation.response_stages.Stage* attribute), 205
- `Instrumentation` (class in *obsinfo.instrumentation.instrumentation*), 198
- `InstrumentComponent` (class in *obsinfo.instrumentation.instrument_component*), 201
- `is_valid_type()` (*obsinfo.obsMetadata.obsmetadata.ObsMetadata* static method), 215
- `isRemote()` (*obsinfo.misc.remoteGitLab.gitLabFile* static method), 222
- ## J
- `JsonLoader` (class in *obsinfo.misc.yamlref*), 222
- `JsonRef` (class in *obsinfo.misc.yamlref*), 223
- `JsonRefError`, 224
- ## L
- `label` (*obsinfo.network.Station* attribute), 195
- `label` (*obsinfo.network.station.Station* attribute), 197
- `latitude` (*obsinfo.instrumentation.location.Location* attribute), 213
- `list_valid_types()` (*obsinfo.obsMetadata.obsmetadata.ObsMetadata* method), 215
- `load()` (in module *obsinfo.misc.yamlref*), 224
- `load_uri()` (in module *obsinfo.misc.yamlref*), 225
- `loads()` (in module *obsinfo.misc.yamlref*), 225
- `localisation_method` (*obsinfo.instrumentation.location.Location* attribute), 213
- `Location` (class in *obsinfo.instrumentation.location*), 213
- `location` (*obsinfo.instrumentation.channel.Channel* attribute), 199
- `location` (*obsinfo.network.Station* attribute), 195
- `location` (*obsinfo.network.station.Station* attribute), 197
- `location_code` (*obsinfo.instrumentation.channel.Channel* attribute), 199
- `location_code` (*obsinfo.network.Station* attribute), 195
- `location_code` (*obsinfo.network.station.Station* attribute), 197
- `locations` (*obsinfo.network.Station* attribute), 195
- `locations` (*obsinfo.network.station.Station* attribute), 197
- `longitude` (*obsinfo.instrumentation.location.Location* attribute), 213
- ## M
- `main()` (in module *obsinfo.main.makeStationXML*), 218
- `main()` (in module *obsinfo.main.validate*), 220
- `main()` (in module *obsinfo.print_version*), 227
- `manufacturer` (*obsinfo.instrumentation.equipment.Equipment* attribute), 212
- `model` (*obsinfo.instrumentation.equipment.Equipment* attribute), 212
- module
- obsinfo.addons.LC2SDS*, 227
 - obsinfo.addons.LCHEAPO*, 226
 - obsinfo.addons.SDPCHAIN*, 226
 - obsinfo.instrumentation.channel*, 199
 - obsinfo.instrumentation.equipment*, 212
 - obsinfo.instrumentation.filter*, 206
 - obsinfo.instrumentation.instrument_component*, 201
 - obsinfo.instrumentation.instrumentation*, 198
 - obsinfo.instrumentation.location*, 213
 - obsinfo.instrumentation.orientation*, 214
 - obsinfo.instrumentation.response_stages*, 204
 - obsinfo.main.makeStationXML*, 217
 - obsinfo.main.setupObsinfo*, 218
 - obsinfo.main.validate*, 219
 - obsinfo.misc.configuration*, 220
 - obsinfo.misc.const*, 220
 - obsinfo.misc.discoveryfiles*, 221
 - obsinfo.misc.printobs*, 221
 - obsinfo.misc.remoteGitLab*, 222
 - obsinfo.misc.yamlref*, 222
 - obsinfo.network*, 193
 - obsinfo.network.network*, 193
 - obsinfo.network.processing*, 198
 - obsinfo.network.station*, 197
 - obsinfo.obsMetadata.obsmetadata*, 215
 - obsinfo.print_version*, 227
 - obsinfo.tests.remoteGithub*, 226
 - obsinfo.version*, 227
- ## N
- `name` (*obsinfo.instrumentation.response_stages.Stage* attribute), 204
- `Network` (class in *obsinfo.network*), 194, 196
- `Network` (class in *obsinfo.network.network*), 193

normalization_factor
 sinfo.instrumentation.filter.PolesZeros
 tribute), 210

normalization_frequency
 sinfo.instrumentation.filter.PolesZeros
 tribute), 210

numerator_coefficients
 sinfo.instrumentation.filter.Coefficients
 tribute), 207

O

obsinfo.addons.LC2SDS
 module, 227

obsinfo.addons.LCHEAPO
 module, 226

obsinfo.addons.SDPCHAIN
 module, 226

obsinfo.instrumentation.channel
 module, 199

obsinfo.instrumentation.equipment
 module, 212

obsinfo.instrumentation.filter
 module, 206

obsinfo.instrumentation.instrument_component
 module, 201

obsinfo.instrumentation.instrumentation
 module, 198

obsinfo.instrumentation.location
 module, 213

obsinfo.instrumentation.orientation
 module, 214

obsinfo.instrumentation.response_stages
 module, 204

obsinfo.main.makeStationXML
 module, 217

obsinfo.main.setupObsinfo
 module, 218

obsinfo.main.validate
 module, 219

obsinfo.misc.configuration
 module, 220

obsinfo.misc.const
 module, 220

obsinfo.misc.discoveryfiles
 module, 221

obsinfo.misc.printobs
 module, 221

obsinfo.misc.remoteGitLab
 module, 222

obsinfo.misc.yamlref
 module, 222

obsinfo.network
 module, 193

obsinfo.network.network

 module, 193

obsinfo.network.processing
 module, 198

obsinfo.network.station
 module, 197

obsinfo.obsMetadata.obsmetadata
 module, 215

obsinfo.print_version
 module, 227

obsinfo.tests.remoteGithub
 module, 226

obsinfo.version
 module, 227

ObsMetadata (class in *obsinfo.obsMetadata.obsmetadata*), 215

obspsy_channel (*obsinfo.instrumentation.channel.Channel* attribute), 199

obspsy_elevation (obsinfo.instrumentation.location.Location property), 213

obspsy_equipment (obsinfo.instrumentation.equipment.Equipment attribute), 212

obspsy_equipment (obsinfo.instrumentation.instrument_component.InstrumentComponent attribute), 202

obspsy_latitude (*obsinfo.instrumentation.location.Location* attribute), 213

obspsy_latitude (*obsinfo.instrumentation.location.Location* property), 213

obspsy_longitude (obsinfo.instrumentation.location.Location attribute), 213

obspsy_longitude (obsinfo.instrumentation.location.Location property), 214

obspsy_station (*obsinfo.network.Station* attribute), 196

obspsy_station (*obsinfo.network.station.Station* attribute), 197

offset (*obsinfo.instrumentation.filter.Filter* attribute), 209

operator (*obsinfo.network.Network* attribute), 194, 196

operator (*obsinfo.network.network.Network* attribute), 193

Orientation (class in *obsinfo.instrumentation.orientation*), 214

orientation (*obsinfo.instrumentation.channel.Channel* attribute), 199

orientation_code (obsinfo.instrumentation.channel.Channel attribute), 199

output_full_scale (obsinfo.instrumentation.filter.AD_Conversion attribute), 206

output_sample_rate (obsinfo.instrumentation.response_stages.Stage attribute), 206
 output_units (obsinfo.instrumentation.response_stages.Stage attribute), 205
 output_units_description (obsinfo.instrumentation.response_stages.Stage attribute), 205

P

polarity (obsinfo.instrumentation.response_stages.Stage attribute), 205
 poles (obsinfo.instrumentation.filter.PolesZeros attribute), 210
 PolesZeros (class in obsinfo.instrumentation.filter), 210
 Preamplifier (class in obsinfo.instrumentation.instrument_component), 202
 print_component() (obsinfo.misc.printobs.PrintObs static method), 221
 print_instrumentation() (obsinfo.misc.printobs.PrintObs static method), 221
 print_network() (obsinfo.misc.printobs.PrintObs static method), 222
 print_station() (obsinfo.misc.printobs.PrintObs static method), 222
 PrintObs (class in obsinfo.misc.printobs), 221
 process_script() (in module obsinfo.addons.LC2SDS), 227
 process_script() (in module obsinfo.addons.LCHEAPO), 226
 process_script() (in module obsinfo.addons.SDPCHAIN), 226
 Processing (class in obsinfo.network.processing), 198
 processing (obsinfo.network.Station attribute), 195
 processing (obsinfo.network.station.Station attribute), 197
 processing_list (obsinfo.network.processing.Processing attribute), 198

R

read_info_file() (obsinfo.obsMetadata.obsmetadata.ObsMetadata static method), 216
 read_json_yaml() (obsinfo.obsMetadata.obsmetadata.ObsMetadata method), 216
 read_json_yaml_ref() (obsinfo.obsMetadata.obsmetadata.ObsMetadata method), 216
 read_json_yaml_ref_datapath() (obsinfo.obsMetadata.obsmetadata.ObsMetadata method), 216
 remote (obsinfo.main.validate.ValidateObsinfo attribute), 219
 removal_date (obsinfo.instrumentation.equipment.Equipment attribute), 212
 replace_refs() (obsinfo.misc.yamlref.JsonRef class method), 223
 resolve_pointer() (obsinfo.misc.yamlref.JsonRef method), 224
 resource_id (obsinfo.instrumentation.equipment.Equipment attribute), 212
 response_list (obsinfo.instrumentation.filter.ResponseList attribute), 211
 response_stages (obsinfo.instrumentation.instrument_component.InstrumentComponent attribute), 202
 ResponseList (class in obsinfo.instrumentation.filter), 211
 ResponseStages (class in obsinfo.instrumentation.response_stages), 204
 restricted_status (obsinfo.network.Network attribute), 194, 196
 restricted_status (obsinfo.network.network.Network attribute), 193
 restricted_status (obsinfo.network.Station attribute), 195
 restricted_status (obsinfo.network.station.Station attribute), 197
 retrieve_arguments() (in module obsinfo.main.makeStationXML), 218
 retrieve_arguments() (in module obsinfo.main.setupObsinfo), 218
 retrieve_arguments() (in module obsinfo.main.validate), 220
 retrieve_configuration() (obsinfo.instrumentation.instrument_component.InstrumentComponent method), 202

S

seed_band_base_code (obsinfo.instrumentation.instrument_component.Sensor attribute), 203
 seed_code (obsinfo.instrumentation.channel.Channel property), 201
 seed_instrument_code (obsinfo.instrumentation.instrument_component.Sensor attribute), 203
 selected_config (obsinfo.instrumentation.equipment.Equipment attribute), 212
 Sensor (class in obsinfo.instrumentation.instrument_component), 203

serial_number (*obsinfo.instrumentation.equipment.Equipment* attribute), 210
 setUp() (*obsinfo.main.validate.ValidateObsinfo* method), 219
 setup_obsinfo() (in module *obsinfo.main.setupObsinfo*), 218
 Singleton (class in *obsinfo.misc.configuration*), 220
 site (*obsinfo.network.Station* attribute), 195
 site (*obsinfo.network.station.Station* attribute), 197
 Stage (class in *obsinfo.instrumentation.response_stages*), 204
 stage_sequence_number (*obsinfo.instrumentation.response_stages.Stage* attribute), 205
 stages (*obsinfo.instrumentation.response_stages.ResponseStages* attribute), 204
 start_date (*obsinfo.instrumentation.channel.Channel* attribute), 199
 start_date (*obsinfo.network.Network* attribute), 194, 196
 start_date (*obsinfo.network.network.Network* attribute), 193
 start_date (*obsinfo.network.Station* attribute), 195
 start_date (*obsinfo.network.station.Station* attribute), 197
 Station (class in *obsinfo.network*), 195
 Station (class in *obsinfo.network.station*), 197
 stations (*obsinfo.network.Network* attribute), 194, 196
 stations (*obsinfo.network.network.Network* attribute), 193
 symmetry (*obsinfo.instrumentation.filter.FIR* attribute), 208

T

to_obspsy() (*obsinfo.instrumentation.channel.Channel* method), 201
 to_obspsy() (*obsinfo.instrumentation.equipment.Equipment* method), 212
 to_obspsy() (*obsinfo.instrumentation.response_stages.ResponseStages* method), 204
 to_obspsy() (*obsinfo.instrumentation.response_stages.Stage* method), 206
 to_obspsy() (*obsinfo.network.Network* method), 195, 196
 to_obspsy() (*obsinfo.network.network.Network* method), 194
 to_obspsy() (*obsinfo.network.Station* method), 196
 to_obspsy() (*obsinfo.network.station.Station* method), 198
 transfer_function_type (*obsinfo.instrumentation.filter.Coefficients* attribute), 207
 transfer_function_type (*obsinfo.instrumentation.filter.PolesZeros* attribute), 210

U

uncertainties_m (*obsinfo.instrumentation.location.Location* attribute), 213

V

validate() (*obsinfo.obsMetadata.obsmetadata.ObsMetadata* method), 217
 validate_all_components() (*obsinfo.main.validate.ValidateObsinfo* method), 219
 validate_all_filters() (*obsinfo.main.validate.ValidateObsinfo* method), 219
 validate_all_instrumentations() (*obsinfo.main.validate.ValidateObsinfo* method), 219
 validate_all_networks() (*obsinfo.main.validate.ValidateObsinfo* method), 219
 validate_all_stages() (*obsinfo.main.validate.ValidateObsinfo* method), 219
 validate_arguments() (in module *obsinfo.main.setupObsinfo*), 218
 validate_date() (*obsinfo.obsMetadata.obsmetadata.ObsMetadata* method), 217
 validate_dates() (*obsinfo.obsMetadata.obsmetadata.ObsMetadata* method), 217
 validate_files() (*obsinfo.main.validate.ValidateObsinfo* method), 219
 validate_filters_in_directory() (*obsinfo.main.validate.ValidateObsinfo* method), 220
 validate_path (*obsinfo.misc.discoveryfiles.Datapath* attribute), 221
 validate_single_file() (*obsinfo.main.validate.ValidateObsinfo* method), 220
 validate_stages_in_directory() (*obsinfo.main.validate.ValidateObsinfo* method), 220
 ValidateObsinfo (class in *obsinfo.main.validate*), 219
 vault (*obsinfo.instrumentation.location.Location* attribute), 213

vendor (*obsinfo.instrumentation.equipment.Equipment*
attribute), [212](#)

verbose (*obsinfo.main.validate.ValidateObsinfo* *at-*
tribute), [219](#)

Z

zeros (*obsinfo.instrumentation.filter.PolesZeros* *at-*
tribute), [210](#)