## CONTENTS

1 First Steps
   1.1 pyfusion overview ........................................ 3
   1.2 Installing Pyfusion ........................................ 3
   1.3 Pyfusion Tutorial .......................................... 8

2 Reference
   2.1 Pyfusion Reference ........................................ 13
   2.2 Development ................................................ 28

3 Indices and tables ........................................ 31

Python Module Index ........................................ 33

Index ........................................................ 35
Release 0.5

Date  June 06, 2011
FIRST STEPS

1.1 pyfusion overview

1.2 Installing Pyfusion

   **Release** 0.5  
   **Date** June 06, 2011

Minimum Python version: 2.4. Python 3 not yet supported.

1.2.1 Installing pyfusion on Ubuntu Linux

   **Release** 0.5  
   **Date** June 06, 2011

This procedure has been tested for Ubuntu 10.04 LTS 64bit and assumes you have sudo privileges.

**Required**

Numpy provides optimised linear algebra libraries used by pyfusion:

`sudo apt-get install python-numpy`

**Recommended**

Ipython is an interactive python environment which is much more convenient to use than the default python shell:

`sudo apt-get install ipython`

Nose is used for testing python code in development:

`sudo apt-get install python-nose`
Installing pyfusion

At present, the recommended method of installing pyfusion is from the code repository. You’ll need a directory in your PYTHONPATH to install to, eg:

```
mkdir -p $HOME/code/python
echo "export PYTHONPATH="$PYTHONPATH:$HOME/code/python" >> $HOME/.bashrc
source $HOME/.bashrc
```

Install the git distributed version control system:

```
sudo apt-get install git-core
```

Make a clone of the pyfusion repository in your python path:

```
cd $HOME/code/python
git clone http://github.com/dpretty/pyfusion.git
```

Until version 1.0 of the code, we’ll be using the dev branch, so you need to check it out:

```
cd pyfusion
git checkout -b dev origin/dev
```

### 1.2.2 Installing pyfusion on Windows XP

**Release** 0.5  
**Date** June 06, 2011

This procedure uses native Windows installs, an alternative method is to use cygwin.

**Requirements**

Enthought Python Distribution contains python, numpy, matplotlib and other libraries used by pyfusion

**Recommended**

msysgit is a windows version of the Git distributed version control system, used to maintain pyfusion.

**Installing Pyfusion**

If you haven’t already got a local directory in your PYTHONPATH, add one, e.g: make a directory C:\Documents and Settings\user\code\python and, using My Computer -> Properties -> Advanced -> Environment Variables set PYTHONPATH to %PYTHONPATH%;C:\Documents and Settings\user\code\python.

If you have msysgit installed, use Start->All programs->Git->Git Bash:

```
cd code/python
git clone git://github.com/dpretty/pyfusion.git
cd pyfusion
git checkout -b dev origin/dev
```
Making a custom configuration file

Edit the file `C:\Documents and Settings\user\.pyfusion\pyfusion.cfg`

1.2.3 Installing pyfusion on CentOS 4.8

Release 0.5
Date June 06, 2011

Because CentOS 4.8 has an old version of python (version 2.3), which won’t work with some of the libraries required by pyfusion, we will install a newer version of python and some libraries in the user’s home directory.

Environment variables

First, set up some environment variables:

```bash
export PYTHONPATH=$PYTHONPATH:$HOME/code/python
mkdir -p code/python
export SOURCEDIR=$HOME/source
mkdir $SOURCEDIR
export LOCALDIR=$HOME/local
mkdir $LOCALDIR
export PATH=$LOCALDIR/bin:$PATH
```

The PYTHONPATH and PATH (and LOCALDIR if PATH refers to it) exports should also go in your `$HOME/.bashrc` file.

Python

Now install Python - the version number is the latest in December 2010, you can use a later one if it exists, but don’t use Python 3.x:

```bash
cd $SOURCEDIR
wget http://www.python.org/ftp/python/2.7.1/Python-2.7.1.tar.bz2
tar -xjf Python-2.7.1.tar.bz2
cd Python-2.7.1
./configure --prefix=$LOCALDIR
make
make install
```

Make sure the default python is now python2.7 (it should be this because $LOCALDIR/bin is first in your PATH environment variable):

```bash
> python
Python 2.7.1 (r271:86832, Dec 26 2010, 03:33:20)
[GCC 3.4.6 20060404 (Red Hat 3.4.6-11)] on linux2
Type “help”, “copyright”, “credits” or “license” for more information.
>>> 
```
**Setuptools**

We need setuptools so we can install pip:

cd $SOURCEDIR
wget http://pypi.python.org/packages/2.7/s/setuptools/setuptools-0.6c11-py2.7.egg
sh setuptools-0.6c11-py2.7.egg --prefix=$LOCALDIR

**pip**

We use pip to install other libraries:

cd $SOURCEDIR
wget http://pypi.python.org/packages/source/p/pip/pip-0.8.2.tar.gz
tar -xzf pip-0.8.2.tar.gz
cd pip-0.8.2
python setup.py install --prefix=$LOCALDIR

**numpy**

Either using sudo or log in as root to install the dependencies:

yum install blas lapack

Then use pip to install numpy:

pip -v install numpy

**sqlalchemy**

Install using pip:

pip -v install sqlalchemy

**ipython**

Optional, but very useful:

pip -v install ipython

**git**

Git is used for pyfusion revision control, and makes it easy for you to update pyfusion and help with development:

cd $SOURCEDIR
wget http://kernel.org/pub/software/scm/git/git-1.7.3.4.tar.bz2
tar -xjf git-1.7.3.4.tar.bz2
cd git-1.7.3.4
./configure --prefix=$LOCALDIR
make
make install
**Pyfusion**

We install with git:

cd $HOME/code/python
git clone git://github.com/dpretty/pyfusion.git
cd pyfusion
git checkout -b dev origin/dev

**Setting up mysql**

Because pyfusion uses sqlalchemy, you can choose from many different types of SQL servers, here we show how to set up MySQL.

As root (or sudo), install the required packages:

yum install mysql mysql-server mysql-devel

Still as root, start MySQL:

/etc/init.d/mysql start

and create a MySQL user for pyfusion:

mysql
> GRANT ALL PRIVILEGES ON *.* TO 'pyfusionuser'@'localhost' IDENTIFIED BY 'mypassword' WITH GRANT OPTION;

Now install (not as root) the python MySQL libraries:

pip -v install MySQL-python

and create a test database to use with pyfusion:

mysql -p
> create database pyfusion_test;

Now, edit $HOME/.pyfusion/pyfusion.cfg to tell pyfusion to use this database, if the directory doesn’t exist, make it:

mkdir $HOME/.pyfusion

and then in $HOME/.pyfusion/pyfusion.cfg:

[global]
database = mysql://pyfusionuser:mypassword@localhost/pyfusion_test

**Matplotlib**

You'll also want matplotlib installed to visualise the data. As root, install the dependencies (we'll use pyqt for the graphics backend):

yum install freetype-devel libpng-devel qt-devel

And as user:

cd $SOURCEDIR
# for some reason pip -v install matplotlib failed for me, so I downloaded the source file separately:
wget http://sourceforge.net/projects/matplotlib/files/matplotlib/matplotlib-1.0/matplotlib-1.0.0.tar.gz
pip -v install matplotlib-1.0.0

1.2. Installing Pyfusion
Now we setup the pyqt backend:

```bash
cd $SOURCEDIR
tar -xzf sip-4.12.tar.gz
cd sip-4.12
python configure.py
make
make install
cd $SOURCEDIR
wget http://www.riverbankcomputing.co.uk/static/Downloads/PyQt3/PyQt-x11-gpl-3.18.1.tar.gz
tar -xzf PyQt-x11-gpl-3.18.1.tar.gz
cd PyQt-x11-gpl-3.18.1
export QTDIR=/usr/lib/qt-3.3
python configure.py
make
make install
```

and configure matplotlib to use this backend:

```bash
mkdir $HOME/.matplotlib
cp $HOME/local/lib/python2.7/site-packages/matplotlib/mpl-data/matplotlibrc .matplotlib/.
```

and edit $HOME/.matplotlib/matplotlibrc to use the setting:

```bash
backend   : QtAgg
```

### 1.3 Pyfusion Tutorial

**Release** 0.5  
**Date** June 06, 2011

Pyfusion overview text here.

#### 1.3.1 Getting Data

**Using the Device class**

The recommended method of retrieving data with pyfusion is by creating an instance of `Device` class (to represent LHD, H-1, TJ-II, etc) and using the attached `getdata()` method:

```python
>>> import pyfusion
>>> h1 = pyfusion.getDevice('H1')
>>> mirnov_data = h1.acq.getdata(58133, 'H1_mirnov_array_1_coil_1')
```

The `getDevice()` method takes a single argument which corresponds to a Device entry in the pyfusion configuration:

```python
[Device:H1]
device_class = pyfusion.devices.H1.device.H1
acq_name = MDS_h1
```

getDevice() then returns an instance the specified subclass of `Device` (here, `pyfusion.devices.H1.device.H1` is a subclass of `Device`) initiated with the same argument, i.e. the following are synonyms:
>>> h1 = pyfusion.getDevice('H1')

and:

>>> from pyfusion.devices.H1.device import H1
>>> h1 = H1('H1')

Pre-loading of data acquisition system during Device instantiation.

Data acquisition in pyfusion is handled by two components, an Acquisition class and a DataFetcher class. The Acquisition class sets up a connection with a data acquisition system; a DataFetcher class uses this connection to fetch a requested data. When a new instance of a Device (sub)class is created, pyfusion will look to the configuration file to see if any data acquisition system is specified. In the example above, the H1 configuration contains:

```
acq_name = MDS_h1
```

which tells pyfusion to look for the data acquisition configuration:

```
[Acquisition:MDS_h1]
acq_class = pyfusion.acquisition.MDSPlus.acq.MDSPlusAcquisition
server = h1data.anu.edu.au
```

and attach a new instance of the specified `acq_class` to the device:

```python
>>> import pyfusion

>>> h1 = pyfusion.getDevice('H1')

>>> print h1.acquisition
<pyfusion.acquisition.MDSPlus.acq.MDSPlusAcquisition object at 0x96d460c>

>>> h1.acq == h1.acquisition
True
```

Where `h1.acq` is simply a shortcut to `h1.acquisition`. The connection to the data acquisition system is created when the Acquisition class is instantiated, in this example: `mdsconnect()` is called when `h1.acquisition` is created.

Data acquisition via getdata

In our original example:

```python
>>> mirnov_data = h1.acq.getdata(58133, 'H1_mirnov_array_1_coil_1')
```

the `getdata()` method is essentially a wrapper around the DataFetcher class which uses the configuration setting to determine which subclass of DataFetcher should be used for the specified diagnostic. Here, `h1.acq.getdata(58133, 'H1_mirnov_array_1_coil_1')` looks up the configuration section `[Diagnostic:H1_mirnov_array_1_coil_1]`:

```
[Diagnostic:H1_mirnov_array_1_coil_1]
data_fetcher = pyfusion.acquisition.H1.fetch.H1DataFetcher
mds_path = \h1data::top.operations.mirnov:a14_14:input_1
cords_cylindrical = 1.114, 0.7732, 0.355
coord_transform = H1_mirnov
```

An instance of the class specified by `data_fetcher` (a subclass of DataFetcher) is created with the parameters specified in the configuration. DataFetcher classes have a `fetch()` method, which returns the data as a pyfusion Data object. `getdata()` calls this `fetch()` method and returns the data object.
1.3.2 Working with Data

Filters

Filters in pyfusion are methods which take a data object, modify the data, and return another (or the same) data object. Generally, filters are attached to data classes as methods:

```python
data = h1.getData(.....)
data.reduce_time([0.02,0.03])
```

For example, the internal pyfusion code which defines the `reduce_time()` filter looks like this:

```python
@register("TimeseriesData", "DataSet")
def reduce_time(input_data, new_time_range):
    from pyfusion.data.base import DataSet
    if isinstance(input_data, DataSet):
        output_dataset = input_data.copy()
        output_dataset.clear()
        for data in input_data:
            try:
                output_dataset.add(data.reduce_time(new_time_range))
            except AttributeError:
                pyfusion.logger.warning("Data filter ‘reduce_time’ not applied to item in dataset")
        return output_dataset
    new_time_args = searchsorted(input_data.timebase, new_time_range)
    input_data.timebase = input_data.timebase[new_time_args[0]:new_time_args[1]]
    if input_data.signal.ndim == 1:
        input_data.signal = input_data.signal[new_time_args[0]:new_time_args[1]]
    else:
        input_data.signal = input_data.signal[:,new_time_args[0]:new_time_args[1]]
    return input_data
```

1.3.3 What have I done?? – Checking data history

Any pyfusion object can be considered as the result of a series of operations performed on some original data sourced from a data acquisition system. It is easy to lose track of which filter settings, etc have been applied to any data - pyfusion takes care of that for you. For example:

```python
>>> from numpy.random import rand
>>> from pyfusion.data.base import Coords
>>> from pyfusion.data.timeseries import TimeseriesData, generate_timebase, Signal

>>> # generate test timebase
>>> tb = generate_timebase(t0=-0.5, n_samples=1.e2, sample_freq=1.e2)
>>> # generate random signal
>>> tsd = TimeseriesData(timebase=tb,
...     signal=Signal(rand(len(tb))),
...     coords=[Coords()])

>>> # apply some filters
>>> tsd.subtract_mean()
>>> tsd.normalise(method='rms')

>>> # have a look at what we’ve done...
>>> print tsd.history
```
The data log is automatically provided for all filters which are added via the `@register()` decorator. Currently, no useful information is provided about how the original data was created (which shot, which channel, etc) – this will be added soon.

1.3.4 Example: Data mining with pyfusion

Here we show a method for finding classes of coherent fluctuations from many multi-channel timeseries signals (see Pretty and Blackwell paper).
2.1 Pyfusion Reference

Release 0.5
Date June 06, 2011

2.1.1 devices – Representation of fusion device

In general usage, a customised subclass will be used rather than direct instantiation of classes in this module. The getDevice() helper function is provided to simplify the use of device classes. If we have in our configuration file:

```
[Device:MyDevice]
dev_class = pyfusion.devices.H1.device.H1
param_1 = param_1_value
```

then getDevice() will return an instance of dev_class:

```
>>> import pyfusion
>>> my_device = pyfusion.getDevice("MyDevice")
>>> my_device
<pyfusion.devices.H1.device.H1 instance at 0xa1d26ec>
>>> my_device.param_1
'param_1_value'
```

Base Device Objects

The following classes are provided by the base submodule. In general, a customised subclass of Device is used rather than the base class itself.

```python
class devices.base.Device (config_name=None, **kwargs)
    Returns an instance of Device optionally initialised with an underlying instance of a specified subclass of BaseAcquisition.

    Parameters config_name -- configuration file section (i.e. [Device:config_name]) to load.

    Any configuration file option can be overridden by supplying an argument of the same name to Device. For example, given a configuration file:

    [Device:MyDevice]
    param_1 = param_1_value
    param_2 = param_2_value
```
the configuration will be loaded by specifying the config_name:

```python
>>> import pyfusion
>>> my_device = pyfusion.devices.base.Device(config_name="MyDevice")
>>> my_device.param_1
'param_1_value'
>>> my_device.param_2
'param_2_value'
```

```python
>>> my_other_device = pyfusion.devices.base.Device(config_name="MyDevice", param_1="some_other_value")
>>> my_other_device.param_1
'some_other_value'
```

The only configuration parameter which is directly handled by `Device` is “acq_name”, which specifies the name of the data acquisition system (i.e. [Acquisition:MyAcquisition] in the configuration file) to be attached to the `Device` instance. So, for a configuration file:

```
[Device:MyDevice]
acq_name = MyAcquisition
param_1 = param_1_value
param_2 = param_2_value

[Acquisition:MyAcquisition]
acq_class = pyfusion.acquisition.base.BaseAcquisition
```

we get:

```python
>>> import pyfusion
>>> my_device = pyfusion.devices.base.Device(config_name="MyDevice")
>>> my_device.acquisition
<pyfusion.acquisition.base.BaseAcquisition object at 0x87a3a4c>
>>> my_device.acq
<pyfusion.acquisition.base.BaseAcquisition object at 0x87a3a4c>
```

where `my_device.acquisition` (and synonym `my_device.acq`) is an instance of `BaseAcquisition`.

### H-1 Device Class

```python
class devices.H1.device.H1
    Trivial subclass of Device which doesn’t add anything new (yet).
```

#### 2.1.2 acquisition – data acquisition

The acquisition package is responsible for fetching data from an experimental database and returning pyfusion data objects. Base classes as well as datasystem-specific sub-packages are provided.

Two classes are involved in obtaining data. An acquisition class (subclass of `BaseAcquisition`) provides the basic interface to the data source, setting up any connections required. A fetcher class (subclass of `BaseDataFetcher`) is used to get data from a specified channel and shot number. In general usage, a fetcher class is not handled directly, but via the `getdata()` method. For example:

```python
>>> import pyfusion
>>> h1 = pyfusion.getDevice('H1')
>>> mirnov_data = h1.acq.getdata(58133, 'H1_mirnov_array_1coil_1')
```

Here, `h1` is an instance of `H1` (the subclass of `Device` specified in the [Device:H1] section in the configuration file). When instantiated, the device class checks the configuration file for a acquisition class specification, and attaches an instance of the specified acquisition class, here `h1.acq` (which is a synonym of...
h1.acquisition). The `getdata()` method checks for a configuration section (here it is a section named `[Diagnostic:H1_mirnov_array_1_coil_1]`) with information about the diagnostic including which data fetcher class to use. The data fetcher is then called to fetch and return the data.

### Base classes

```python
class pyfusion.acquisition.base.BaseAcquisition(config_name=None, **kwargs):
    Base class for datasystem specific acquisition classes.
    
    Parameters
    config_name -- name of acquisition as specified in configuration file.
    
    On instantiation, the pyfusion configuration is searched for a `[Acquisition:config_name]` section. The contents of the configuration section are loaded into the object namespace. For example, a configuration section:
    ```
    [Acquisition:my_custom_acq]
    acq_class = pyfusion.acquisition.base.BaseAcquisition
    server = my.dataserver.com
    ```
    will result in the following behaviour:
    ```
    >>> from pyfusion.acquisition.base import BaseAcquisition
    >>> my_acq = BaseAcquisition('my_custom_acq')
    >>> print(my_acq.server)
    my.dataserver.com
    ```
    The configuration entries can be overridden with keyword arguments:
    ```
    >>> my_other_acq = BaseAcquisition('my_custom_acq', server='your.data.net')
    >>> print(my_other_acq.server)
    your.data.net
    ```

    `getdata(shot, config_name=None, **kwargs)`
    Get the data and return prescribed subclass of BaseData.
    
    Parameters
    - shot -- shot number
    - config_name -- name of a fetcher class in the configuration file
    
    Returns
    an instance of a subclass of BaseData or BaseDataSet.
    
    This method needs to know which data fetcher class to use, if a config_name argument is supplied then the `[Diagnostic:config_name]` section must exist in the configuration file and contain a data_fetcher class specification, for example:
    ```
    [Diagnostic:H1_mirnov_array_1_coil_1]
    data_fetcher = pyfusion.acquisition.H1.fetch.H1DataFetcher
    mds_path = \h1data::top.operations.mirnov:a14_14:input_1
    coords_cylindrical = 1.114, 0.7732, 0.355
    coord_transform = H1_mirnov
    ```
    If a data_fetcher keyword argument is supplied, it overrides the configuration file specification.
    
    The fetcher class is instantiated, including any supplied keyword arguments, and the result of the fetch method of the fetcher class is returned.
```

```python
class pyfusion.acquisition.base.BaseDataFetcher(acq, shot, config_name=None, **kwargs):
    Base class providing interface for fetching data from an experimental database.
    
    Parameters
    ```
• **acq** – in instance of a subclass of `BaseAcquisition`
• **shot** – shot number
• **config_name** – name of a Diagnostic configuration section.

It is expected that subclasses of `BaseDataFetcher` will be called via the `getdata()` method, which calls the data fetcher’s `fetch()` method.

```python
def do_fetch():
    # Actually fetches the data, using the environment set up by setup()
    # Returns an instance of a subclass of BaseData or BaseDataSet
```

Although `BaseDataFetcher.do_fetch()` does not return any data object itself, it is expected that a `do_fetch()` method on a subclass of `BaseDataFetcher` will.

```python
def fetch():
    # Always use this to fetch the data, so that setup() and pulldown() are used to setup and pull down
    # the environmet used by do_fetch().
    # Returns the instance of a subclass of BaseData or BaseDataSet returned by do_fetch()
```

```python
def pulldown():
    # Called by fetch() after retrieving the data.
```

```python
def setup():
    # Called by fetch() before retrieving the data.
```

```python
class pyfusion.acquisition.base.MultiChannelFetcher(acq, shot, config_name=None, **kwargs):
```

Fetch data from a diagnostic with multiple timeseries channels.

This fetcher requires a multichannel configuration section such as:

```python
[Diagnostic:H1_mirnov_array_1]
data_fetcher = pyfusion.acquisition.base.MultiChannelFetcher
cchannel_1 = H1_mirnov_array_1_coil_1
cchannel_2 = H1_mirnov_array_1_coil_2
cchannel_3 = H1_mirnov_array_1_coil_3
cchannel_4 = H1_mirnov_array_1_coil_4
```

The channel names must be `channel_` followed by an integer, and the channel values must correspond to other configuration sections (for example `[Diagnostic:H1_mirnov_array_1_coil_1]`, `[Diagnostic:H1_mirnov_array_1_coil_3]`, etc) which each return a single channel instance of `TimeseriesData`.

```python
def fetch():
    # Fetch each channel and combine into a multichannel instance of TimeseriesData.
    # Return type TimeseriesData
```

```python
def ordered_channel_names():
    # Get an ordered list of the channel names in the diagnostic
    # Return type list
```

## Sub-packages for specific data sources

Custom subclasses `BaseAcquisition` and `BaseDataFetcher` classes are contained in dedicated sub-packages. Each sub-package has the structure:
MDSPlus

Interface for MDSplus data acquisition and storage.

This package depends on the MDSplus python package, available from http://www.mdsplus.org/binaries/python/

Pyfusion supports four modes for accessing MDSplus data:

1. local
2. thick client
3. thin client
4. HTTP via a H1DS MDSplus web service

The data access mode used is determined by the mds path and server variables in the configuration file (or supplied to the acquisition class via keyword arguments):

```
[Acquisition:my_data]
acq_class = pyfusion.acquisition.MDSPlus.acq.MDSPlusAcquisition
mydata_path = ...
server = my.mdsdataserver.net
```

The full MDSplus node path is stored in a diagnostic configuration section:

```
[Diagnostic:my_probe]
data_fetcher = pyfusion.acquisition.MDSPlus.fetch.MDSPlusDataFetcher
mds_node_path = \mydata::top.probe_signal
```

Local data access  The ‘local’ mode is used when a tree path definition refers to the local file system rather than an MDSplus server on the network. The mydata_path entry in the above example would look something like:

```
mydata_path = /path/to/my/data
```

Thick client access  The ‘thick client’ mode uses an MDSplus data server to retrieve the raw data files, but the client is responsible for evaluating expressions and decompressing the data. The server tree definitions are used, and the server for a given mds tree is specified by the tree path in the format:

```
mydata_path = my.mdsdataserver.net::
```

or, if a port other than the default (8000) is used:

```
mydata_path = my.mdsdataserver.net:port_number::
```

Thin client access  The ‘thin client’ mode maintains a connection to an MDSplus data server. Expressions are evaluated and data decompressed on the server, requiring greater amounts of data to be transferred over the network. Because the thin client mode uses the tree paths defined on the server, no path variable is required. Instead, the server entry is used:
server = my.mdsdataserver.net

or, if a port other than the default (8000) is used:
server = my.mdsdataserver.net:port_number

**HTTP web service access**  The HTTP web service mode uses standard HTTP queries via the H1DS RESTful API to access the MDSplus data. The server is responsible for evaluating the data and transmits quantisation-compressed data to the client over port 80. This is especially useful if the MDSplus data is behind a firewall. The `server` attribute will be used for web service access if it begins with `http://`, for example:

server = http://hlsvr.anu.edu.au/mdsplus/

The `server` attribute must be the URL component up to the MDSplus tree name. In this example, the URL for mds path `\h1data::top.operations.mirnov:a14_14:input_1` and shot 58063 corresponds to `http://hlsvr.anu.edu.au/mdsplus/h1data/58063/top/operations/mirnov/a14_14/input_1/`

**How Pyfusion chooses the access mode**  If an acquisition configuration section contains a `server` entry (which does not start with `http://`), then `MDSPlusAcquisition` will set up a connection to the mdsip server when it is instantiated. Additionally, any tree path definitions (local and thick client) are loaded into the runtime environment at this time. When a call to the data fetcher is made (via `getdata()`), the data fetcher uses the full node path (including tree name) from the configuration file. If a matching (tree name) `.path` variable is defined for the acquisition module, then the corresponding local or thick client mode will be used. If no tree path is defined then, if the `server` variable is defined, pyfusion will attempt to use either the web services mode (if `server` begins with `http://`) or the thin client mode (if `server` does not begin with `http://`).

**Classes**

```python
class pyfusion.acquisition.MDSPlus.acq.MDSPlusAcquisition(*args, **kwargs)
    Acquisition class for MDSplus data systems.
```

If a ‘server’ configuration parameter (not starting with ‘http’) is provided, a connection for thin client access will be set up. Also, any configuration parameters which end with ‘.path’ will be loaded into the environment.

```python
class pyfusion.acquisition.MDSPlus.fetch.MDSPlusDataFetcher(acq, shot, config_name=None, **kwargs)
    Determine which access mode should be used, and fetch the MDSplus data.
```

**H1**

The H1 data acquisition package.

This subpackage contains a subclass of the MDSplus data fetcher which gets additional H1 specific metadata.

**Classes**

```python
class pyfusion.acquisition.H1.fetch.H1DataFetcher(acq, shot, config_name=None, **kwargs)
    Subclass of MDSplus fetcher to get additional H1-specific metadata.
```

**LHD**

Data acquisition for LHD.
DSV

Acquisition module for data in a delimiter-separated value (DSV) format.

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>Name of data file, with (shot) substitution string, e.g. /data/(shot).dat -&gt; /data/12345.dat for shot 12345. (required)</td>
</tr>
<tr>
<td>delimiter</td>
<td>Delimiter character for values, e.g., , for comma separated value (CSV) format. (optional, default is whitespace)</td>
</tr>
</tbody>
</table>

This module provides support for reading data from a plain text file via numpy’s genfromtxt function. The only required configuration parameter is `filename`, which can include a shot number substitution string `(shot)`. An example, consider the following datafile for 2-channel timeseries signal for shot number 12345:

```
# timebase channel 1 channel 2
3.000000e+00 -1.201389e-01 3.177084e-01
3.000002e+00 6.437500e-01 -4.461806e-01
3.000004e+00 5.347222e-02 -1.684028e-01
3.000006e+00 1.923611e-01 -2.951390e-02
3.000008e+00 4.006945e-01 -5.156250e-01
3.000010e+00 -8.840278e-01 1.012153e+00
3.000012e+00 2.618056e-01 -2.031250e-01
3.000014e+00 -1.597222e-02 -1.336806e-01
3.000016e+00 -1.597222e-02 1.788194e-01
3.000018e+00 5.743055e-01 -7.586806e-01
```

If the datafile is saved at `/data/mirnov_data_12345.txt`, we could use the following configuration file:

```
[Acquisition:my_text_data]
acq_class = pyfusion.acquisition.DSV.acq.DSVAcquisition

[Diagnostic:mirnov_data]
data_fetcher = pyfusion.acquisition.DSV.fetch.DSVMultiChannelTimeseriesFetcher
filename = /data/mirnov_data_(shot).txt
```

And access the data with pyfusion:

```python
>>> import pyfusion as pf
>>> acq = pf.getAcquisition("my_text_data")
>>> data = acq.getdata(12345, "mirnov_data")
```

By default, pyfusion expects values to be delimited by whitespace characters. The delimiting character can also be set in the configuration file, for example, the following datafile and configuration give the same result as the above example:

```
# timebase, channel 1, channel 2
3.000000e+00, -1.201389e-01, 3.177084e-01
3.000002e+00, 6.437500e-01, -4.461806e-01
3.000004e+00, 5.347222e-02, -1.684028e-01
```

2.1. Pyfusion Reference
3.000006e+00, 1.923611e-01, -2.951390e-02
3.000008e+00, 4.006945e-01, -5.156250e-01
3.000010e+00, -8.840278e-01, 1.012153e+00
3.000012e+00, 2.618056e-01, -2.031250e-01
3.000014e+00, -1.597222e-02, -1.336806e-01
3.000016e+00, -1.597222e-02, 1.788194e-01
3.000018e+00, 5.743055e-01, -7.586806e-01

where the configuration is:

```
[Acquisition:my_text_data]
acq_class = pyfusion.acquisition.DSV.acq.DSVAcquisition
```

```
[Diagnostic:mirnov_data]
data_fetcher = pyfusion.acquisition.DSV.fetch.DSVMultiChannelTimeseriesFetcher
filename = /data/mirnov_data_(shot).txt
delimiter = ,
```

Note that whitespace is stripped from configuration file values - if you want to use whitespace delimited data, as in the first example, simply omit the delimiter setting in your configuration.

### Classes

#### pyfusion.acquisition.DSV.acq.DSVAcquisition

```
class pyfusion.acquisition.DSV.acq.DSVAcquisition(config_name=None, **kwargs)
```

Fetch DSV data from specified filename.

This data fetcher uses two configuration parameters, filename (required) and delimiter (optional).

The filename parameter can include a substitution string (shot) which will be replaced with the shot number.

By default, whitespace is used for the delimiter character (if the delimiter parameter is not provided.)

#### pyfusion.acquisition.DSV.fetch.DSVMultiChannelTimeseriesFetcher

```
class pyfusion.acquisition.DSV.fetch.DSVMultiChannelTimeseriesFetcher(acq, shot, config_name=None, **kwargs)
```

Acquisition module for generating fake timeseries data for testing purposes.

At present, only a single channel sine wave generator is provided. Available configuration parameters are:

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>t0</code></td>
<td>Starting time of signal timebase.</td>
</tr>
<tr>
<td><code>n_samples</code></td>
<td>Number of samples.</td>
</tr>
<tr>
<td><code>sample_freq</code></td>
<td>Sample frequency (Hz).</td>
</tr>
<tr>
<td><code>frequency</code></td>
<td>Frequency of test sine-wave signal (Hz).</td>
</tr>
<tr>
<td><code>amplitude</code></td>
<td>Amplitude of test sine-wave signal.</td>
</tr>
</tbody>
</table>

All parameters are required.

For example, with the following configuration:

```
[Acquisition:fake_acq]
acq_class = pyfusion.acquisition.FakeData.acq.FakeDataAcquisition
```

```
[Diagnostic:fake_data]
data_fetcher = pyfusion.acquisition.FakeData.fetch.SingleChannelSineFetcher
```
t0 = 0.0
n_samples = 1024
sample_freq = 1.e6
frequency = 2.e4
amplitude = 2.5

we can generate a 20 kHz sine wave:
>>> import pyfusion as pf
>>> shot = 12345
>>> acq = pf.getAcquisition("fake_acq")
>>> data = acq.getdata(shot, "fake_data")
>>> data.timebase
Timebase([ 0.00000000e+00, 1.00000000e-06, 2.00000000e-06, ..., 1.02100000e-03, 1.02200000e-03, 1.02300000e-03])
>>> data.signal
Signal([ 0. , 0.31333308, 0.62172472, ..., 1.20438419, 0.92031138, 0.62172472])

Classes

class pyfusion.acquisition.FakeData.acq.FakeDataAcquisition(config_name=None, **kwargs)

Acquisition class for generating fake data for testing purposes.

class pyfusion.acquisition.FakeData.fetch.SingleChannelSineFetcher(acq, shot, config_name=None, **kwargs)

Data fetcher for single channel sine wave.

2.1.3 Data

2.1.4 Configuration files

Overview

Pyfusion uses simple text files to store information such as data acquisition settings, diagnostic coordinates, SQL database configurations, etc. A pyfusion configuration file looks something like this:

[global]
database = sqlite:///memory:

[Device:H1]
dev_class = pyfusion.devices.H1.device.H1
acq_name = MDS_h1

[Acquisition:MDS_h1]
acq_class = pyfusion.acquisition.MDSPlus.acq.MDSPlusAcquisition
server = h1data.anu.edu.au

[CoordTransform:H1_mirnov]
magnetic = pyfusion.devices.H1.coords.MirnovKhMagneticCoordTransform

[Diagnostic:H1_mirnov_array_1_coil_1]
data_fetcher = pyfusion.acquisition.H1.fetch.H1DataFetcher
mds_path = h1data::top.operations.mirnov:a14_14:input_1

2.1. Pyfusion Reference
coords_cylindrical = 1.114, 0.7732, 0.355
coord_transform = H1_mirnov

[Diagnostic:H1_mirnov_array_1_coil_2]
data_fetcher = pyfusion.acquisition.H1.fetch.H1DataFetcher
mds_path = \h1data::top.operations.mirnov:a14_14:input_2
coords_cylindrical = 1.185, 0.7732, 0.289
coord_transform = H1_mirnov

[Diagnostic:H1_mirnov_array_1_coil_3]
data_fetcher = pyfusion.acquisition.H1.fetch.H1DataFetcher
mds_path = \h1data::top.operations.mirnov:a14_14:input_3
coords_cylindrical = 1.216, 0.7732, 0.227
coord_transform = H1_mirnov

[Diagnostic:H1_mirnov_array_1]
data_fetcher = pyfusion.acquisition.base.MultiChannelFetcher
channel_1 = H1_mirnov_array_1_coil_1
channel_2 = H1_mirnov_array_1_coil_2
channel_3 = H1_mirnov_array_1_coil_3

There are two types of sections in this file: there is one special section (global) and several component sections (e.g. Device:H1, Acquisition:MDS_h1, CoordTransform:H1_mirnov, etc.)

The sections in the configuration (except for [variabletypes]) file have the syntax [Component:name], where Component is one of: Acquisition, Device, Diagnostic. When instantiating a class, such as Device, Acquisition, Diagnostic, etc. which looks in the configuration file for settings, individual settings can be overridden using the corresponding keyword arguments. For example, Device('my_device') will use settings in the [Device:my_device] configuration section, and Device('my_device', database='sqlite://') will override the database configuration setting with sqlite:// (a temporary in-memory database).

The pyfusion configuration parser pyfusion.conf.PyfusionConfigParser is a simple subclass of the standard python configparser, for example, to see the configuration sections, type:

pyfusion.config.sections()

Loading config files

When pyfusion is imported, will load the default configuration file provided in the source code (that is in the pyfusion directory) followed by your custom configuration file, in $HOME/.pyfusion/pyfusion.cfg, if it exists. and finally files pointed to by the environment variable PYFUSION_CONFIG_FILE if they exist. This allows temporarily overriding config variables.

Additional config files can be loaded with pyfusion.read_config():

pyfusion.read_config(['another_config_filename_1', 'another_config_filename_2'])

The read_config argument can either be a single file-like object (any object which has a readlines() method) or a list of filenames, as shown above. If you do not supply any argument, read_config() will load the default configuration files (the same ones loaded when you import pyfusion).

To clear the loaded pyfusion configuration, use pyfusion.clear_config(). If you want to return the configuration to the default settings (the configuration you have when you import pyfusion), type:

pyfusion.clear_config()
pyfusion.read_config()
[variabletypes]

variabletypes is a section for defining the types (integer, float, boolean) of variables specified throughout the configuration file. By default, variables are assumed to be strings (text) - only variables of type integer, float or boolean should be listed here.

For example, if three variables (arguments) for the Diagnostic class are n_samples (integer), sample_freq (float) and normalise (boolean) the syntax is:

```python
Diagnostic__n_samples = int
Diagnostic__sample_freq = float
Diagnostic__normalise = bool
```

Note the double underscore (__) separating the class type and the variable name.

[Device:name]

database

Location of database in the SQLAlchemy database URL syntax.

e.g.: no example yet

acq_name

Name of Acquisition config setting ( [Acquisition:acq_name] ) to be used for this device.

e.g.: acq_name = test_fakedata

dev_class

Name of device class (subclass of pyfusion.devices.base.Device) to be used for this device. This is called when using the convenience function pyfusion.getDevice. For example, if the configuration file contains:

```ini
[Device:my_tjii_device]
dev_class = pyfusion.devices.TJII
```

then using:

```python
import pyfusion
my_dev = pyfusion.getDevice('my_tjii_device')
```

my_dev will be an instance of pyfusion.devices.TJII

[Acquisition:name]

acq_class

Location of acquisition class (subclass of pyfusion.acquisition.base.BaseAcquisition).
e.g.:
acq_class = pyfusion.acquisition.fakedata.FakeDataAcquisition

[Diagnostic:name]

data_fetcher

Location of class (subclass of pyfusion.acquisition.base.BaseDataFetcher) to fetch the data for the diagnostic.

tests.cfg

A separate configuration file “tests.cfg”, in the same ”.pyfusion” folder in your home directory, can be used during development to enable tests which are disabled by default.

An example of the syntax is:

[EnabledTests]
mdsplus = True
flucstrucs = True

etc...

Database

The database layer is handled by SQLAlchemy

Database URL

Database URLs are the same as for SQLAlchemy:

driver://username:password@host:port/database

Simplifying changes by substitution

The syntax %(sym)s will substitute the contents of sym. e.g. fetchr = pyfusion.acquisition.H1.fetch.H1LocalTimeseriesDataFetcherh1datafetcher data_fetcher = %(fetchr)s

This way only one edit needs to be made to change all diagnostics, if the definition is fetchr is in the special [DEFAULT] section

User Defined Sections

We will probably include a section Plots containing things like FT_Axis = [0, 0.08, 0, 300000] to provide defaults for the Frequency-Time axis etc. Note that such settings are highly device dependent and although they will be recognised in the code, they usually should not be given values in code distributions.

The User could put their own items in there or other sections to avoid

For more details, refer to http://www.sqlalchemy.org/docs/05/dbengine.html#create-engine-url-arguments
2.1.5 Coordinates

Coordinates can be defined in the configuration file, for example pyfusion.cfg contains:

```py
[CoordTransform:H1_mirnov]
magnetic = pyfusion.devices.H1.coords.MirnovKhMagneticCoordTransform
```

```py
[Diagnostic:H1_mirnov_array_1_coil_1]
data_fetcher = pyfusion.acquisition.H1.fetch.H1DataFetcher
mds_path = \h1data::top.operations.mirnov:a14_14:input_1
coords_cylindrical = 1.114, 0.7732, 0.355
coord_transform = H1_mirnov
```

which specifies coords_cylindrical as the canonical coordinate representation, followed by a reference to a magnetic coordinate transform which depends on magnetic configuration (and therefore varies from shot to shot).

When data is acquired, eg:

```py
import pyfusion as pf
h1 = pf.getDevice('H1')
data = h1.acq.getdata(58010, 'H1_mirnov_array_1_coil_1')
```

2.1.6 Object relational mapping

Overview

Object relational mapping (ORM) is a method of maintaining a connection between a relational database (e.g. MySQL) and object orientated programming languages (e.g. python). The types of data used with pyfusion are very well suited to being stored in relational databases, i.e. we deal with a large number of items which all share the same set of attributes. With an ORM, we get the benefits of both fast and efficient SQL querying of the data, and object orientated code.

In pyfusion, the ORM is activated by setting the database configuration variable in the [global] section of your configuration file. Pyfusion will not use ORM if database is set to None.

SQLAlchemy

SQLAlchemy is a python library which provides a comprehensive interface to relational databases, and includes ORM. The documentation is available here: [http://www.sqlalchemy.org/docs/](http://www.sqlalchemy.org/docs/)

The pyfusion ORM configuration

Pyfusion uses SQLAlchemy for its ORM. The standard method for configuring an ORM with SQLAlchemy is to explicitly construct Table objects and link them to Python classes with a mapping object. An alternative configuration is to use the SQLAlchemy declarative extension, which provides a base class which provides Table and mapper attributes to any class which inherits it. The two approaches represent different styles code rather than providing different functionality. Pyfusion uses the standard approach to keep ORM code separate from non-ORM (class definitions) code, allowing pyfusion to be used without ORM.

Module-wide configuration

Code which controls module-wide (i.e. all of pyfusion) ORM is located in pyfusion.orm. The important components here are the ORM engine, session and metadata. These are created by pyfusion.orm.setup_orm()
which is called after the configuration files are read during `import pyfusion`.

**Engine** The SQLAlchemy engine provides an abstraction of the relational database (beneath it could be MySQL, Postgres, SQLite, etc), and a pool of connections to the database. Starting a database connection is an expensive operation, to streamline database interaction, the engine keeps a pool of connections which it uses and recycles to avoid the overhead of creating database connections for each operation:

```python
pyfusion.orm_engine = create_engine(pyfusion.config.get('global', 'database'))
```

**Session** An instance of the SQLAlchemy `Session` class is used to manage interactions with the database, it can keep track of modifications to data instances and flush multiple changes to the database when required. We use `scoped_session` to provide a thread-local `Session` instance, which allows us to use the same session in different parts of pyfusion. The session configuration looks like:

```python
pyfusion.Session = scoped_session(sessionmaker(autocommit=False,
                                           autoflush=True,
                                           bind=pyfusion.orm_engine))
```

The `autocommit` and `autoflush` arguments prescribe how the session should organise transactions. A database transaction refers to a group of queries which should be treated as a single operation on the database, either all queries in a should be applied, or none of them should. Using `commit()` in an sqlalchemy session commits the current transaction, whereas `flush()` will write pending data to the database without closing the transaction. In `autocommit` mode SQLAlchemy automatically commits after each `flush()`, while this removes some flexibility in construction of transactions it can be useful for testing and debug purposes. Regardless of these settings, `commit()` will always call a `flush()` before committing the transaction. The `autoflush=True` argument specifies that `flush()` should be called before any individual query is issued.

**Class-level configuration**

The ORM code for classes in pyfusion follows the class code, and is active only if `pyfusion.USE_ORM` is `True`. The ORM class code contains a Table definition, a call to `pyfusion.metadata.create_all()` to create the table, and a mapping of the class to the table. For example, the `Device` class definition and ORM code appears as:

```python
class Device(object):
    def __init__(self, config_name, **kwargs):
        if pyfusion.config.pf_has_section('Device', config_name):
            self.__dict__.update(get_config_as_dict('Device', config_name))
            self.__dict__.update(kwargs)
            self.name = config_name

        ### attach acquisition
        if hasattr(self, 'acq_name'):
            acq_class_str = pyfusion.config.pf_get('Acquisition',
                                                    self.acq_name, 'acq_class')
            self.acquisition = import_from_str(acq_class_str)(self.acq_name)
            # shortcut
            self.acq = self.acquisition
        else:
            pyfusion.logging.warning("No acquisition class specified for device")

if pyfusion.USE_ORM:
    from sqlalchemy import Table, Column, Integer, String
```

26 Chapter 2. Reference
from sqlalchemy.orm import mapper
device_table = Table('devices', pyfusion.metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String, unique=True))

pyfusion.metadata.create_all()
mapper(Device, device_table)

Does pyfusion read from the config file or data database?

notes: e.g. when a device is created which has a config definition, it will be loaded from sql if it exists. if it doesn't exist it will be created. at present there is no checking to make sure that the sql version matches the params of the config. there is no automated way of changing the sql version if you change the config - this shouldn't be done anyway, as other data may have been created with the existing device, diagnostic etc and it we don't want to have processed data attached to an instance which is not responsible for its creation... etc...

2.1.7 Logging

2.1.8 Internals

Static internal values

The following are set when the pyfusion module is imported, and are not designed to be changed by the user.

PYFUSION_ROOT_DIR

A string containing the directory where the pyfusion module resides.

LOGGING_CONFIG_FILE

Filename of the configuration file used to set up pyfusion.logger.

VERSION

Pyfusion version, as returned by pyfusion.version.get_version().

DEFAULT_CONFIG_FILE

Location of file containing default pyfusion configuration.

USER_PYFUSION_DIR

Location of user's pyfusion directory. On linux this will be $HOME/.pyfusion
USER_CONFIG_FILE

Location of user’s pyfusion configuration file. On Linux this will be $HOME/.pyfusion/pyfusion.cfg

USER_ENV_CONFIG_FILE

This is set to the value of the PYFUSION_CONFIG_FILE environment variable, which supersedes USER_CONFIG_FILE and DEFAULT_CONFIG_FILE.

2.2 Development

2.2.1 Pyfusion Development

Design principles

Documentation

Documentation is maintained in the code repository. An online version is kept up to date here.

Test-driven design (TDD)

Distributed source code

Developing with pyfusion

Documentation

• use sphinx
• built docs not stored in repository

Tests

• use nosetests

• running nosetests pyfusion should be very fast. The idea behind regular testing is that the tests should be so fast that you don’t hesitate to run the test. Any test which requires significant computation or hard disk / network access should be disabled by default. Using $HOME/.pyfusion/tests.cfg you can enable any of these tests when you need them.

• selection of which tests are run is done with nosetests attributes, see nose docs for detail. For example, to run all tests except those which use SQL:

> nosetests -a '!sql' pyfusion

• There should not be conditional tests for pyfusion.USE_ORM within the test code as there can be confusion as to which configuration settings are present in the testing environment. Instead, use a separate class for the SQL code and provide it with the `sql` attribute.
The available attributes are:

<table>
<thead>
<tr>
<th>attr</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td>test requires sqlalchemy module</td>
</tr>
<tr>
<td>net</td>
<td>test requires internet access</td>
</tr>
<tr>
<td>lhd</td>
<td>test connects to LHD data acquisition system</td>
</tr>
<tr>
<td>tjii</td>
<td>test connects to TJII data acquisition system</td>
</tr>
<tr>
<td>h1</td>
<td>test connects to H-1 data acquisition system</td>
</tr>
<tr>
<td>plot</td>
<td>test requires matplotlib module</td>
</tr>
<tr>
<td>daq</td>
<td>test connects to a data system (superset of lhd, tjii and h1)</td>
</tr>
</tbody>
</table>

### 2.2.2 Pyfusion development roadmap

#### version 0.5
- full documentation of existing code

#### version 0.6
- signal amplitudes able to be extracted from fluestrucs
- metadata API
- clustering interface
- capability for efficient I/O (text file?) while doing (multi-process) pre-processing, and put back into sql asynchronously.
- allow separate configuration files for different devices, etc.

#### version 0.7
- re-implement HJ, TJII, W7-AS interfaces

#### version 1.0
- full feature compatibility with original version

### 2.2.3 Pyfusion tests
CHAPTER
THREE

INDICES AND TABLES

• genindex
• modindex
• search
Python Module Index

d
devices, 13
devices.base, 13
devices.H1.device, 14

p
pyfusion.acquisition, 14
pyfusion.acquisition.DSV, 19
pyfusion.acquisition.FakeData, 20
pyfusion.acquisition.H1, 18
pyfusion.acquisition.LHD, 18
pyfusion.acquisition.MDSPlus, 17
pyfusion.devices.tests, 29
pyfusion.test.tests, 29
INDEX

B
BaseAcquisition (class in pyfusion.acquisition.base), 15
BaseDataFetcher (class in pyfusion.acquisition.base), 15

D
Device (class in devices.base), 13
devices (module), 13
devices.base (module), 13
devices.H1.device (module), 14
do_fetch() (pyfusion.acquisition.base.BaseDataFetcher
method), 16
DSVAcquisition (class in pyfusion.acquisition.DSV.acq), 20
DSVMultiChannelTimeseriesFetcher (class in pyfusion.acquisition.DSV.fetch), 20

F
FakeDataAcquisition (class in pyfusion.acquisition.FakeData.acq), 21
fetch() (pyfusion.acquisition.base.BaseDataFetcher
method), 16
fetch() (pyfusion.acquisition.base.MultiChannelFetcher
method), 16

G
getdata() (pyfusion.acquisition.base.BaseAcquisition
method), 15

H
H1 (class in devices.H1.device), 14
H1DataFetcher (class in pyfusion.acquisition.H1.fetch), 18

M
MDSSPlusAcquisition (class in pyfusion.acquisition.MDSSPlus.acq), 18
MDSSPlusDataFetcher (class in pyfusion.acquisition.MDSSPlus.fetch), 18
MultiChannelFetcher (class in pyfusion.acquisition.base), 16

O
ordered_channel_names() (pyfusion.acquisition.base.MultiChannelFetcher
method), 16

P
pulldown() (pyfusion.acquisition.base.BaseDataFetcher
method), 16
pyfusion.acquisition (module), 14
pyfusion.acquisition.DSV (module), 19
pyfusion.acquisition.FakeData (module), 20
pyfusion.acquisition.H1 (module), 18
pyfusion.acquisition.LHD (module), 18
pyfusion.acquisition.MDSSPlus (module), 17
pyfusion.devices.tests (module), 29
pyfusion.test.tests (module), 29

S
setup() (pyfusion.acquisition.base.BaseDataFetcher
method), 16
SingleChannelSineFetcher (class in pyfusion.acquisition.FakeData.fetch), 21