pwkit

Release 1.0.0

Jun 18, 2023

Contents

This documentation has a lot of stubs.

CHAPTER 1

About the Software

`pwkit` is a collection of Peter Williams' miscellaneous Python tools. I'm packaging them so that other people can install them off of PyPI or Conda and run my code without having to go to too much work. That's the hope, at least.

1.1 Installation

The most recent stable version of `pwkit` is available on the Python package index, so you should be able to install this package simply by running `pip install pwkit`. The package is also available in the conda package manager by installing it from anaconda.org. If you are using packages from the conda-forge project, install with `conda install -c pkgw-forge pwkit`. Otherwise, use `conda install -c pkgw pwkit`.

If you want to download the source code and install `pwkit` manually, the package uses the standard Python setuptools, so running `python setup.py install` will do the trick.

Some `pwkit` functionality requires additional Python modules such as scipy; these issues should be very obvious as they manifest as `ImportErrors` triggered for the relevant modules. Bare minimum functionality requires:

- numpy >= 1.6
- six >= 1.9
- on Python 2.x only, pathlib >= 1.0

If you install `pwkit` through standard means, these modules should be automatically installed too if they weren't already available.

1.2 Citation

If you use pwkit in academic work, you should identify that you have done so and specify the version used. While pwkit does not (yet?) have an accompanying formal publication, in journals like ApJ you can "cite" the code directly via its record in the NASA Astrophysics Data System, which has identifier 2017ascl.soft04001W. This corresponds to record ascl:1704.001 in in the Astrophysics Source Code Library. By clicking on this link you can get the ADS-recommended BibTeX record for the reference.

If you are using aastex version 6 or higher, the appropriate code to include after your Acknowledgments section would be:

```
\software{..., pwkit \citep{2017ascl.soft04001W}, ...}
```

1.3 Authors

`pwkit` is authored by Peter K. G. Williams and collaborators. Despite this package being named after me, contributions are welcome and will be given full credit. I just don't want to have to make up a decent name for this package right now.

Contributions have come from (alphabetically by surname):

- Maïca Clavel
- Elisabeth Newton
- Denis Ryzhkov (I copied method_decorator)

1.4 Copyright and License

CHAPTER 2

Foundations

This documentation has a lot of stubs.

2.1 Core utilities (`pwkit`)

A toolkit for science and astronomy in Python.

The toplevel *`pwkit`* module includes a few basic abstractions that show up throughout the rest of the codebase. These include:

- *The Holder namespace object*
- *Utilities for exceptions*
- *Abstractions between Python versions 2 and 3*

2.1.1 The `Holder` namespace object

`Holder` is a "namespace object" that primarily exists so that you can fill it with named attributes however you want. It's essentially like a plain `dict`, but you can write the convenient form `myholder.xcoord` instead of `mydict['xcoord']`. It has useful methods like *`set()`* and *`to_pretty()`* also.

class pwkit.**Holder**(*_Holder__decorating=None*, ***kwargs*)

Create a new *`Holder`*. Any keyword arguments will be assigned as properties on the object itself, for instance, `o = Holder(foo=1)` yields an object such that `o.foo` is 1.

The *__decorating* keyword is used to implement the *`Holder`* decorator functionality, described below.

`get`(name[, defval])	Get an attribute on this *`Holder`*.
`set`(**kwargs)	For each keyword argument, sets an attribute on this *`Holder`* to its value.
`set_one`(name, value)	Set a single attribute on this object.

Continued on next page

Table 1 – continued from previous page

has(name)	Return whether the named attribute has been set on this object.
copy()	Return a shallow copy of this object.
to_dict()	Return a copy of this object converted to a `dict`.
to_pretty([format])	Return a string with a prettified version of this object's contents.

Iterating over a *Holder* yields its contents in the form of a sequence of `(name, value)` tuples. The stringification of a *Holder* returns its representation in a dict-like format. *Holder* objects implement `__contains__` so that boolean tests such as `"myprop" in myholder` act sensibly.

get (*name*, *defval=None*)
Get an attribute on this *Holder*.

Equivalent to `getattr(self, name, defval)`.

set (***kwargs*)
For each keyword argument, sets an attribute on this *Holder* to its value.

Equivalent to:

```
for key, value in kwargs.iteritems():
  setattr(self, key, value)
```

Returns *self*.

set_one (*name*, *value*)
Set a single attribute on this object.

Equivalent to `setattr(self, name, value)`. Returns *self*.

has (*name*)
Return whether the named attribute has been set on this object.

This can more naturally be expressed by writing `name in self`.

copy ()
Return a shallow copy of this object.

to_dict ()
Return a copy of this object converted to a `dict`.

to_pretty (*format='str'*)
Return a string with a prettified version of this object's contents.

The format is a multiline string where each line is of the form `key = value`. If the *format* argument is equal to `"str"`, each `value` is the stringification of the value; if it is `"repr"`, it is its `repr()`.

Calling `str()` on a *Holder* returns a slightly different pretty stringification that uses a textual representation similar to a Python `dict` literal.

`@pwkit.`**Holder**
The *Holder* class may also be used as a decorator on a class definition to transform its contents into a Holder instance. Writing:

```
@Holder
class mydata ():
    a = 1
    b = 'hello'
```

creates a Holder instance named `mydata` containing names `a` and `b`. This can be a convenient way to populate one-off data structures.

2.1.2 Utilities for exceptions

class pwkit.**PKError**(*fmt*, **args*)

A generic base class for exceptions.

All custom exceptions raised by *pwkit* modules should be subclasses of this class.

The constructor automatically applies old-fashioned `printf`-like (`%`-based) string formatting if more than one argument is given:

```
PKError('my format string says %r, %d', myobj, 12345)
# has text content equal to:
'my format string says %r, %d' % (myobj, 12345)
```

If only a single argument is given, the exception text is its stringification without applying `printf`-style formatting.

pwkit.**reraise_context**(*fmt*, **args*)

Reraise an exception with its message modified to specify additional context.

This function tries to help provide context when a piece of code encounters an exception while trying to get something done, and it wishes to propagate contextual information farther up the call stack. It only makes sense in Python 2, which does not provide Python 3's exception chaining functionality. Instead of that more sophisticated infrastructure, this function just modifies the textual message associated with the exception being raised.

If only a single argument is supplied, the exception text prepended with the stringification of that argument. If multiple arguments are supplied, the first argument is treated as an old-fashioned `printf`-type (`%`-based) format string, and the remaining arguments are the formatted values.

Example usage:

```
from pwkit import reraise_context
from pwkit.io import Path

filename = 'my-filename.txt'

try:
  f = Path(filename).open('rt')
  for line in f.readlines():
    # do stuff ...
except Exception as e:
  reraise_context('while reading "%r"', filename)
  # The exception is reraised and so control leaves this function.
```

If an exception with text `"bad value"` were to be raised inside the `try` block in the above example, its text would be modified to read `"while reading "my-filename.txt": bad value"`.

2.1.3 Abstractions between Python versions 2 and 3

The toplevel *pwkit* module imports the following variables from the `six` package that helps with Python 2/3 compatibility:

- `binary_type`

- text_type

pwkit.**unicode_to_str**(*s*)

A function for implementing the __str__ method of classes, the meaning of which differs between Python versions 2 and 3. In all cases, you should implement __unicode__ on your classes. Setting the __str__ property of a class to *unicode_to_str()* will cause it to Do The Right Thing™, which means returning the UTF-8 encoded version of its Unicode expression in Python 2, or returning the Unicode expression directly in Python 3:

```
import pwkit

class MyClass (object):
    def __unicode__ (self):
        return u'my value'

    __str__ = pwkit.unicode_to_str
```

2.2 Convenient file input and output (pwkit.io)

The pwkit package provides many tools to ease reading and writing data files. The most generic such tools are located in this module. The most important tool is the *Path* class for object-oriented navigation of the filesystem.

The functionality in this module can be grouped into these categories:

- *The Path object*
- *Functions helping with Unicode safety*
- *Other functions in pwkit.io* (generally being superseded by *Path*)

2.2.1 The Path object

class pwkit.io.**Path**

This is an extended version of the pathlib.Path class. (pathlib is built into Python 3.x and is available as a backport to Python 2.x.) It represents a path on the filesystem.

The methods and attributes on *Path* objects fall into several broad categories:

- *Manipulating and dissecting paths*
- *Filesystem interrogation*
- *Filesystem modifications*
- *Data input and output*

Constructors are:

Path(*part*, **more*)

Returns a new path equivalent to os.path.join (part, *more), except the arguments may be either strings or other *Path* instances.

classmethod cwd()

Returns a new path containing the absolute path of the current working directory.

classmethod create_tempfile(*want='handle'*, *resolution='try_unlink'*, *suffix=''*, ***kwargs*)

Returns a context manager managing the creation and destruction of a named temporary file. The operation of this function is exactly like that of the bound method *Path.make_tempfile()*, except that instead

of creating a temporary file with a name similar to an existing path, this function creates one with a name selected using the standard OS-dependent methods for choosing names of temporary files.

The `overwrite` resolution is not allowed here since there is no original path to overwrite.

Note that by default the returned context manager returns a file-like object and not an actual *Path* instance; use `want="path"` to get a *Path*.

Manipulating and dissecting paths

Child paths can be created by using the division operator, that is:

```
parent = Path ('directory')
child = parent / 'subdirectory'
```

Combining a relative path with an absolute path in this way will just yield the absolute path:

```
>>> foo = Path ('relative') / Path ('/a/absolute')
>>> print (foo)
<<< /a/absolute
```

Paths should be converted to text by calling `str()` or `unicode()` on them.

Instances of *Path* have the following attributes that help you create new paths or break them into their components:

anchor	The concatenation of the drive and root, or ‘’.
drive	The drive prefix (letter or UNC path), if any.
name	The final path component, if any.
parent	The logical parent of the path.
parents	A sequence of this path’s logical parents.
parts	An object providing sequence-like access to the components in the filesystem path.
stem	The final path component, minus its last suffix.
suffix	The final component’s last suffix, if any.
suffixes	A list of the final component’s suffixes, if any.

And they have the following related methods:

absolute()	Return an absolute version of this path.
as_uri()	Return the path as a ‘file’ URI.
expand([user, vars, glob, resolve])	Return a new *Path* with various expansions performed.
format(*args, **kwargs)	Return a new path formed by calling `str.format()` on the textualization of this path.
get_parent([mode])	Get the path of this path’s parent directory.
is_absolute()	True if the path is absolute (has both a root and, if applicable, a drive).
joinpath(*args)	Combine this path with one or several arguments, and return a new path representing either a subpath (if all arguments are relative paths) or a totally different path (if one of the arguments is anchored).
make_relative(other)	Return a new path that is the equivalent of this one relative to the path *other*.

Continued on next page

Table 3 – continued from previous page

relative_to(*other)	Return the relative path to another path identified by the passed arguments.
resolve([strict])	Make the path absolute, resolving all symlinks on the way and also normalizing it (for example turning slashes into backslashes under Windows).
with_name(name)	Return a new path with the file name changed.
with_suffix(suffix)	Return a new path with the file suffix changed.

Detailed descriptions of attributes

Path.**anchor**
The concatenation of *drive* and root.

Path.**drive**
The Windows or network drive of the path. The empty string on POSIX.

Path.**name**
The final path component. The *name* of /foo/ is "foo". The *name* of /foo/. is "foo" as well. The *name* of /foo/.. is "..".

Path.**parent**
This path's parent, in a textual sense: the *parent* of foo is ., but the parent of . is also .. The parent of /bar is /; the parent of / is also /.

See also:

Path.get_parent()

Path.**parents**
An immutable, indexable sequence of this path's parents. Here are some examples showing the semantics:

```
>>> list(Path("/foo/bar").parents)
<<< [Path("/foo"), Path("/")]
>>> list(Path("/foo/bar/").parents)
<<< [Path("/foo"), Path("/")]
>>> list(Path("/foo/bar/.").parents)
<<< [Path("/foo"), Path("/")]
>>> list(Path("/foo/./bar/.").parents)
<<< [Path("/foo"), Path("/")]
>>> list(Path("wib/wob").parents)
<<< [Path("wib"), Path(".")]
>>> list(Path("wib/../wob/.").parents)
<<< [Path("wib/.."), Path("wib"), Path(".")]
```

See also:

Path.get_parent()

Path.**parts**
A tuple of the path components. Examples:

```
>>> Path('/a/b').parts
<<< ('/', 'a', 'b')
>>> Path('a/b').parts
<<< ('a', 'b')
>>> Path('/a/b/').parts
<<< ('/', 'a', 'b')
```

(continues on next page)

(continued from previous page)

```
>>> Path('a/b/.').parts
<<< ('a', 'b')
>>> Path('/a/../b/./c').parts
<<< ('/', 'a', '..', 'b', 'c')
>>> Path('.').parts
<<< ()
>>> Path('').parts
<<< ()
```

`Path.`**`stem`**
: The *name* without its suffix. The stem of `"foo.tar.gz"` is `"foo.tar"`. The stem of `"noext"` is `"noext"`. It is an invariant that `name = stem + suffix`.

`Path.`**`suffix`**
: The suffix of the *name*, including the period. If there is no period, the empty string is returned:

```
>>> print (Path("foo.tar.gz").suffix)
<<< .gz
>>> print (Path("foo.dir/.").suffix)
<<< .dir
>>> print (repr (Path("noextension").suffix))
<<< ''
```

`Path.`**`suffixes`**
: A list of all suffixes on *name*, including the periods. The suffixes of `"foo.tar.gz"` are `[".tar", ".gz"]`. If *name* contains no periods, the empty list is returned.

Detailed descriptions of methods

`Path.`**`absolute`**`()`
: Return an absolute version of the path. Unlike *resolve()*, does not normalize the path or resolve symlinks.

`Path.`**`as_uri`**`()`
: Return the path stringified as a `file:///` URI.

`Path.`**`expand`**(*user=False*, *vars=False*, *glob=False*, *resolve=False*)
: Return a new *Path* with various expansions performed. All expansions are disabled by default but can be enabled by passing in true values in the keyword arguments.

  user [bool (default False)] Expand `~` and `~user` home-directory constructs. If a username is unmatched or `$HOME` is unset, no change is made. Calls `os.path.expanduser()`.

  vars [bool (default False)] Expand `$var` and `${var}` environment variable constructs. Unknown variables are not substituted. Calls `os.path.expandvars()`.

  glob [bool (default False)] Evaluate the path as a `glob` expression and use the matched path. If the glob does not match anything, do not change anything. If the glob matches more than one path, raise an `IOError`.

  resolve [bool (default False)] Call *resolve()* on the return value before returning it.

`Path.`**`format`**(**args*, ***kwargs*)
: Return a new path formed by calling `str.format()` on the textualization of this path.

`Path.`**`get_parent`**(*mode='naive'*)
: Get the path of this path's parent directory.

  Unlike the *parent* attribute, this function can correctly ascend into parent directories if *self* is `"."` or a sequence of `".."`. The precise way in which it handles these kinds of paths, however, depends on the *mode*

parameter:

"textual" Return the same thing as the *parent* attribute.

"resolved" As *textual*, but on the *resolve()*-d version of the path. This will always return the physical parent directory in the filesystem. The path pointed to by *self* must exist for this call to succeed.

"naive" As *textual*, but the parent of "." is "..", and the parent of a sequence of ".." is the same sequence with another "..". Note that this manipulation is still strictly textual, so results when called on paths like "foo/../bar/../other" will likely not be what you want. Furthermore, p.get_parent(mode="naive") never yields a path equal to p, so some kinds of loops will execute infinitely.

Path.**is_absolute**()
Returns whether the path is absolute.

Path.**joinpath**(**args*)
Combine this path with several new components. If one of the arguments is absolute, all previous components are discarded.

Path.**make_relative**(*other*)
Return a new path that is the equivalent of this one relative to the path *other*. Unlike *relative_to()*, this will not throw an error if *self* is not a sub-path of *other*; instead, it will use .. to build a relative path. This can result in invalid relative paths if *other* contains a directory symbolic link.

If *self* is an absolute path, it is returned unmodified.

Path.**relative_to**(**other*)
Return this path as made relative to another path identified by *other*. If this is not possible, raise ValueError.

Path.**resolve**()
Make this path absolute, resolving all symlinks and normalizing away ".." and "." components. The path must exist for this function to work.

Path.**with_name**(*name*)
Return a new path with the file name changed.

Path.**with_suffix**(*suffix*)
Return a new path with the file *suffix* changed, or a new suffix added if there was none before. *suffix* must start with a ".". The semantics of the *suffix* attribute are maintained, so:

```
>>> print (Path ('foo.tar.gz').with_suffix ('.new'))
<<< foo.tar.new
```

Filesystem interrogation

These methods probe the actual filesystem to test whether the given path, for example, is a directory; but they do not modify the filesystem.

exists()	Whether this path exists.
glob(pattern)	Iterate over this subtree and yield all existing files (of any kind, including directories) matching the given relative pattern.
is_block_device()	Whether this path is a block device.
is_char_device()	Whether this path is a character device.
is_dir()	Whether this path is a directory.
is_fifo()	Whether this path is a FIFO.

Continued on next page

Table 4 – continued from previous page

is_file()	Whether this path is a regular file (also True for symlinks pointing to regular files).
is_socket()	Whether this path is a socket.
is_symlink()	Whether this path is a symbolic link.
iterdir()	Iterate over the files in this directory.
match(path_pattern)	Return True if this path matches the given pattern.
readlink()	Assuming that this path is a symbolic link, read its contents and return them as another *Path* object.
rglob(pattern)	Recursively yield all existing files (of any kind, including directories) matching the given relative pattern, anywhere in this subtree.
scandir()	Iteratively scan this path, assuming it's a directory.
stat()	Return the result of the stat() system call on this path, like os.stat() does.

Detailed descriptions

Path.**exists**()
: Returns whether the path exists.

Path.**glob**(*pattern*)
: Assuming that the path is a directory, iterate over its contents and return sub-paths matching the given shell-style glob pattern.

Path.**is_block_device**()
: Returns whether the path resolves to a block device file.

Path.**is_char_device**()
: Returns whether the path resolves to a character device file.

Path.**is_dir**()
: Returns whether the path resolves to a directory.

Path.**is_fifo**()
: Returns whether the path resolves to a Unix FIFO.

Path.**is_file**()
: Returns whether the path resolves to a regular file.

Path.**is_socket**()
: Returns whether the path resolves to a Unix socket.

Path.**is_symlink**()
: Returns whether the path resolves to a symbolic link.

Path.**iterdir**()
: Iterate over the files in this directory. Does not yield any result for the special paths '.' and '..'.

  Assuming the path is a directory, generate a sequence of sub-paths corresponding to its contents.

Path.**match**(*pattern*)
: Test whether this path matches the given shell glob pattern.

Path.**readlink**()
: Assuming that this path is a symbolic link, read its contents and return them as another *Path* object. An "invalid argument" OSError will be raised if this path does not point to a symbolic link.

Path.**rglob**(*pattern*)
Recursively yield all files and directories matching the shell glob pattern *pattern* below this path.

Path.**scandir**()
Iteratively scan this path, assuming it's a directory. This requires and uses the `scandir` module.

scandir is different than *iterdir* because it generates *DirEntry* items rather than Path instances. DirEntry objects have their properties filled from the directory info itself, so querying them avoids syscalls that would be necessary with iterdir().

The generated values are `scandir.DirEntry` objects which have some information pre-filled. These objects have methods `inode()`, `is_dir()`, `is_file()`, `is_symlink()`, and `stat()`. They have attributes `name` (the basename of the entry) and `path` (its full path).

Path.**stat**()
Run `os.stat()` on the path and return the result.

Filesystem modifications

These functions actually modify the filesystem.

chmod(mode)	Change the permissions of the path, like os.chmod().
copy_to(dest[, preserve])	Copy this path — as a file — to another *dest*.
ensure_dir([mode, parents])	Ensure that this path exists as a directory.
ensure_parent([mode, parents])	Ensure that this path's *parent* directory exists.
make_tempfile([want, resolution, suffix])	Get a context manager that creates and cleans up a uniquely-named temporary file with a name similar to this path.
mkdir([mode, parents, exist_ok])	Create a new directory at this given path.
rellink_to(target[, force])	Make this path a symlink pointing to the given *target*, generating the proper relative path using `make_relative()`.
rename(target)	Rename this path to the given path.
rmdir()	Remove this directory.
rmtree([errors])	Recursively delete this directory and its contents.
symlink_to(target[, target_is_directory])	Make this path a symlink pointing to the given path.
touch([mode, exist_ok])	Create this file with the given access mode, if it doesn't exist.
unlink()	Remove this file or link.
try_unlink()	Try to unlink this path.

Detailed descriptions

Path.**chmod**(*mode*)
Change the mode of the named path. Remember to use octal `0o755` notation!

Path.**copy_to**(*dest*, *preserve='mode'*)
Copy this path — as a file — to another *dest*.

The *preserve* argument specifies which meta-properties of the file should be preserved:

none Only copy the file data.

mode Copy the data and the file mode (permissions, etc).

all Preserve as much as possible: mode, modification times, etc.

The destination *dest* may be a directory.

Returns the final destination path.

`Path.`**`ensure_dir`**(*mode=511*, *parents=False*)
Ensure that this path exists as a directory.

This function calls *`mkdir()`* on this path, but does not raise an exception if it already exists. It does raise an exception if this path exists but is not a directory. If the directory is created, *mode* is used to set the permissions of the resulting directory, with the important caveat that the current `os.umask()` is applied.

It returns a boolean indicating if the directory was actually created.

If *parents* is true, parent directories will be created in the same manner.

`Path.`**`ensure_parent`**(*mode=511*, *parents=False*)
Ensure that this path's *parent* directory exists.

Returns a boolean whether the parent directory was created. Will attempt to create superior parent directories if *parents* is true.

`Path.`**`make_tempfile`**(*want='handle'*, *resolution='try_unlink'*, *suffix=''*, ***kwargs*)
Get a context manager that creates and cleans up a uniquely-named temporary file with a name similar to this path.

This function returns a context manager that creates a secure temporary file with a path similar to *self*. In particular, if `str(self)` is something like `foo/bar`, the path of the temporary file will be something like `foo/bar.ame8_2`.

The object returned by the context manager depends on the *want* argument:

`"handle"` An open file-like object is returned. This is the object returned by `tempfile.NamedTemporaryFile`. Its name on the filesystem is accessible as a string as its *name* attribute, or (a customization here) as a *`Path`* instance as its *path* attribute.

`"path"` The temporary file is created as in `"handle"`, but is then immediately closed. A *`Path`* instance pointing to the path of the temporary file is instead returned.

If an exception occurs inside the context manager block, the temporary file is left lying around. Otherwise, what happens to it upon exit from the context manager depends on the *resolution* argument:

`"try_unlink"` Call *`try_unlink()`* on the temporary file — no exception is raised if the file did not exist.

`"unlink"` Call *`unlink()`* on the temporary file — an exception is raised if the file did not exist.

`"keep"` The temporary file is left lying around.

`"overwrite"` The temporary file is *`rename()`*-d to overwrite *self*.

For instance, when rewriting important files, it's typical to write the new data to a temporary file, and only rename the temporary file to the final destination at the end — that way, if a problem happens while writing the new data, the original file is left unmodified; otherwise you'd be stuck with a partially-written version of the file. This pattern can be accomplished with:

```
p = Path ('path/to/important/file')
with p.make_tempfile (resolution='overwrite', mode='wt') as h:
    print ('important stuff goes here', file=h)
```

The *suffix* argument is appended to the temporary file name after the random portion. It defaults to the empty string. If you want it to operate as a typical filename suffix, include a leading `"."`.

Other **kwargs** are passed to `tempfile.NamedTemporaryFile`.

`Path.`**`mkdir`**(*mode=0o777*, *parents=False*)
Create a directory at this path location. Creates parent directories if *parents* is true. Raises `OSError` if the path already exists, even if *parents* is true.

`Path.`**`rellink_to`**(*target*, *force=False*)
Make this path a symlink pointing to the given *target*, generating the proper relative path using *`make_relative()`*. This gives different behavior than *`symlink_to()`*. For instance, `Path ('a/b').symlink_to ('c')` results in `a/b` pointing to the path `c`, whereas *`rellink_to()`* results in it pointing to `../c`. This can result in broken relative paths if (continuing the example) `a` is a symbolic link to a directory.

If either *target* or *self* is absolute, the symlink will point at the absolute path to *target*. The intention is that if you're trying to link `/foo/bar` to `bee/boo`, it probably makes more sense for the link to point to `/path/to/.../bee/boo` rather than `../../../../bee/boo`.

If *force* is true, *`try_unlink()`* will be called on *self* before the link is made, forcing its re-creation.

`Path.`**`rename`**(*target*)
Rename this path to *target*.

`Path.`**`rmdir`**()
Delete this path, if it is an empty directory.

`Path.`**`rmtree`**(*errors='warn'*)
Recursively delete this directory and its contents. The *errors* keyword specifies how errors are handled:

"warn" (the default) Print a warning to standard error.

"ignore" Ignore errors.

`Path.`**`symlink_to`**(*target*, *target_is_directory=False*)
Make this path a symlink pointing to the given target.

`Path.`**`touch`**(*mode=0o666*, *exist_ok=True*)
Create a file at this path with the given mode, if needed.

`Path.`**`unlink`**()
Unlink this file or symbolic link.

`Path.`**`try_unlink`**()
Try to unlink this path. If it doesn't exist, no error is returned. Returns a boolean indicating whether the path was really unlinked.

Data input and output

`open`([mode, buffering, encoding, errors, ...])	Open the file pointed by this path and return a file object, as the built-in open() function does.
`try_open`([null_if_noexist])	Call *`Path.open()`* on this path (passing *kwargs*) and return the result.
`as_hdf_store`([mode])	Return the path as an opened `pandas.HDFStore` object.
`read_astropy_ascii`(**kwargs)	Open as an ASCII table, returning a `astropy.table.Table` object.
`read_fits`(**kwargs)	Open as a FITS file, returning a `astropy.io.fits.HDUList` object.

Continued on next page

Table 6 – continued from previous page

read_fits_bintable([hdu, drop_nonscalar_ok])	Open as a FITS file, read in a binary table, and return it as a `pandas.DataFrame`, converted with `pkwit.numutil.fits_recarray_to_data_frame()`.
read_hdf(key, **kwargs)	Open as an HDF5 file using `pandas` and return the item stored under the key *key*.
read_inifile([noexistok, typed])	Open assuming an "ini-file" format and return a generator yielding data records using either *pwkit.inifile.read_stream()* (if *typed* is false) or `pwkit.tinifile.read_stream()` (if it's true).
read_json([mode])	Use the `json` module to read in this file as a JSON-formatted data structure.
read_lines([mode, noexistok])	Generate a sequence of lines from the file pointed to by this path, by opening as a regular file and iterating over it.
read_numpy(**kwargs)	Read this path into a `numpy.ndarray` using `numpy.load()`.
read_numpy_text([dfcols])	Read this path into a `numpy.ndarray` as a text file using `numpy.loadtxt()`.
read_pandas([format])	Read using `pandas`.
read_pickle()	Open the file, unpickle one object from it using `pickle`, and return it.
read_pickles()	Generate a sequence of objects by opening the path and unpickling items until EOF is reached.
read_tabfile(**kwargs)	Read this path as a table of typed measurements via *pwkit.tabfile.read()*.
read_text([encoding, errors, newline])	Read this path as one large chunk of text.
read_toml([encoding, errors, newline])	Read this path as a TOML document.
read_yaml([encoding, errors, newline])	Read this path as a YAML document.
write_pickle(obj)	Dump *obj* to this path using `cPickle`.
write_pickles(objs)	*objs* must be iterable.
write_yaml(data[, encoding, errors, newline])	Read *data* to this path as a YAML document.

Detailed descriptions

`Path.`**`open`**(*mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*)

Open the file pointed at by the path and return a `file` object. This delegates to the modern `io.open()` function, not the global builtin *open()*.

`Path.`**`try_open`**(*null_if_noexist=False*, ***kwargs*)

Call *Path.open()* on this path (passing *kwargs*) and return the result. If the file doesn't exist, the behavior depends on *null_if_noexist*. If it is false (the default), `None` is returned. Otherwise, `os.devnull` is opened and returned.

`Path.`**`as_hdf_store`**(*mode='r'*, ***kwargs*)

Return the path as an opened `pandas.HDFStore` object. Note that the `HDFStore` constructor unconditionally prints messages to standard output when opening and closing files, so use of this function will pollute your program's standard output. The *kwargs* are forwarded to the `HDFStore` constructor.

`Path.`**`read_astropy_ascii`**(***kwargs*)

Open as an ASCII table, returning a `astropy.table.Table` object. Keyword arguments are passed to `astropy.io.ascii.open()`; valid ones likely include:

- `names = <list>` (column names)

- `format` ('basic', 'cds', 'csv', 'ipac', ...)
- `guess = True` (guess table format)
- `delimiter` (column delimiter)
- `comment = <regex>`
- `header_start = <int>` (line number of header, ignoring blank and comment lines)
- `data_start = <int>`
- `data_end = <int>`
- `converters = <dict>`
- `include_names = <list>` (names of columns to include)
- `exclude_names = <list>` (names of columns to exclude; applied after include)
- `fill_values = <dict>` (filler values)

`Path.`**`read_fits`**`(`***kwargs*`)`

Open as a FITS file, returning a `astropy.io.fits.HDUList` object. Keyword arguments are passed to `astropy.io.fits.open()`; valid ones likely include:

- `mode = 'readonly'` (or "update", "append", "denywrite", "ostream")
- `memmap = None`
- `save_backup = False`
- `cache = True`
- `uint = False`
- `ignore_missing_end = False`
- `checksum = False`
- `disable_image_compression = False`
- `do_not_scale_image_data = False`
- `ignore_blank = False`
- `scale_back = False`

`Path.`**`read_fits_bintable`**`(`*hdu=1*, *drop_nonscalar_ok=True*, ***kwargs*`)`

Open as a FITS file, read in a binary table, and return it as a `pandas.DataFrame`, converted with `pkwit.numutil.fits_recarray_to_data_frame()`. The *hdu* argument specifies which HDU to read, with its default 1 indicating the first FITS extension. The *drop_nonscalar_ok* argument specifies if non-scalar table values (which are inexpressible in `pandas.DataFrame`s) should be silently ignored (``True`) or cause a `ValueError` to be raised (`False`). Other **kwargs** are passed to `astropy.io.fits.open()`, (see *`Path.read_fits()`*) although the open mode is hardcoded to be `"readonly"`.

`Path.`**`read_hdf`**`(`*key*, ***kwargs*`)`

Open as an HDF5 file using `pandas` and return the item stored under the key *key*. *kwargs* are passed to `pandas.read_hdf()`.

`Path.`**`read_inifile`**`(`*noexistok=False*, *typed=False*`)`

Open assuming an "ini-file" format and return a generator yielding data records using either *`pwkit.inifile.read_stream()`* (if *typed* is false) or `pwkit.tinifile.read_stream()` (if it's true). The latter version is designed to work with numerical data using the *`pwkit.msmt`* subsystem. If *noexistok* is true, a nonexistent file will result in no items being generated rather than an `IOError` being raised.

Path.**read_json**(*mode='rt'*, ***kwargs*)
Use the json module to read in this file as a JSON-formatted data structure. Keyword arguments are passed to json.load(). Returns the read-in data structure.

Path.**read_lines**(*mode='rt'*, *noexistok=False*, ***kwargs*)
Generate a sequence of lines from the file pointed to by this path, by opening as a regular file and iterating over it. The lines therefore contain their newline characters. If *noexistok*, a nonexistent file will result in an empty sequence rather than an exception. *kwargs* are passed to *Path.open()*.

Path.**read_numpy**(***kwargs*)
Read this path into a numpy.ndarray using numpy.load(). *kwargs* are passed to numpy.load(); they likely are:

mmap_mode [None, 'r+', 'r', 'w+', 'c'] Load the array using memory-mapping

allow_pickle [bool = True] Whether Pickle-format data are allowed; potential security hazard.

fix_imports [bool = True] Try to fix Python 2->3 import renames when loading Pickle-format data.

encoding ['ASCII', 'latin1', 'bytes'] The encoding to use when reading Python 2 strings in Pickle-format data.

Path.**read_numpy_text**(*dfcols=None*, ***kwargs*)
Read this path into a numpy.ndarray as a text file using numpy.loadtxt(). In normal conditions the returned array is two-dimensional, with the first axis spanning the rows in the file and the second axis columns (but see the *unpack* and *dfcols* keywords).

If *dfcols* is not None, the return value is a pandas.DataFrame constructed from the array. *dfcols* should be an iterable of column names, one for each of the columns returned by the numpy.loadtxt() call. For convenience, if *dfcols* is a single string, it will by turned into an iterable by a call to str.split().

The remaining *kwargs* are passed to numpy.loadtxt(); they likely are:

dtype [data type] The data type of the resulting array.

comments [str] If specific, a character indicating the start of a comment.

delimiter [str] The string that separates values. If unspecified, any span of whitespace works.

converters [dict] A dictionary mapping zero-based column *number* to a function that will turn the cell text into a number.

skiprows [int (default=0)] Skip this many lines at the top of the file

usecols [sequence] Which columns keep, by number, starting at zero.

unpack [bool (default=False)] If true, the return value is transposed to be of shape (cols, rows).

ndmin [int (default=0)] The returned array will have at least this many dimensions; otherwise mono-dimensional axes will be squeezed.

Path.**read_pandas**(*format='table'*, ***kwargs*)
Read using pandas. The function pandas.read_FORMAT is called where FORMAT is set from the argument *format*. *kwargs* are passed to this function. Supported formats likely include clipboard, csv, excel, fwf, gbq, html, json, msgpack, pickle, sql, sql_query, sql_table, stata, table. Note that hdf is not supported because it requires a non-keyword argument; see *Path.read_hdf()*.

Path.**read_pickle**()
Open the file, unpickle one object from it using pickle, and return it.

Path.**read_pickles**()
Generate a sequence of objects by opening the path and unpickling items until EOF is reached.

Path.**read_tabfile**(***kwargs*)
Read this path as a table of typed measurements via *pwkit.tabfile.read()*. Returns a generator for a sequence of *pwkit.Holder* objects, one for each row in the table, with attributes for each of the columns.

tabwidth [int (default=8)] The tab width to assume. Defaults to 8 and should not be changed unless absolutely necessary.

mode [str (default='rt')] The file open mode, passed to io.open().

noexistok [bool (default=False)] If true, a nonexistent file will result in no items being generated, as opposed to an IOError.

kwargs [keywords] Additional arguments are passed to io.open().

Path.**read_text**(*encoding=None*, *errors=None*, *newline=None*)
Read this path as one large chunk of text.

This function reads in the entire file as one big piece of text and returns it. The *encoding*, *errors*, and *newline* keywords are passed to *open()*.

This is not a good way to read files unless you know for sure that they are small.

Path.**read_toml**(*encoding=None*, *errors=None*, *newline=None*, ***kwargs*)
Read this path as a TOML document.

The TOML parsing is done with the pytoml module. The *encoding*, *errors*, and *newline* keywords are passed to *open()*. The remaining *kwargs* are passed to toml.load().

Returns the decoded data structure.

Path.**read_yaml**(*encoding=None*, *errors=None*, *newline=None*, ***kwargs*)
Read this path as a YAML document.

The YAML parsing is done with the yaml module. The *encoding*, *errors*, and *newline* keywords are passed to *open()*. The remaining *kwargs* are passed to yaml.load().

Returns the decoded data structure.

Path.**write_pickle**(*obj*)
Dump *obj* to this path using cPickle.

Path.**write_pickles**(*objs*)
objs must be iterable. Write each of its values to this path in sequence using cPickle.

Path.**write_yaml**(*data*, *encoding=None*, *errors=None*, *newline=None*, ***kwargs*)
Read *data* to this path as a YAML document.

The *encoding*, *errors*, and *newline* keywords are passed to *open()*. The remaining *kwargs* are passed to yaml.dump().

2.2.2 Functions helping with Unicode safety

get_stdout_bytes()	Get a reference to the standard output stream that accepts bytes, not unicode characters.
get_stderr_bytes()	Get a reference to the standard error stream that accepts bytes, not unicode characters.

pwkit.io.**get_stdout_bytes**()
Get a reference to the standard output stream that accepts bytes, not unicode characters.

Returns: a file-like object hooked up to the process' standard output.

Usually, you want to write text to a process's standard output stream ("stdout"), so you want `sys.stdout` to be a stream that accepts Unicode. The function *`pwkit.cli.unicode_stdio()`* sets this up in Python 2, which has an imperfect hack to allow Unicode output to work most of the time. However, there are other times when you really *do* want to write arbitrary binary data to stdout. Depending on whether you're using Python 2 or Python 3, or whether *`pwkit.cli.unicode_stdio()`* has been called, the right way to get access to the underlying byte-based stream is different. This function encapsulates these checks and works across all of these cases.

`pwkit.io.`**`get_stderr_bytes`**`()`
Get a reference to the standard error stream that accepts bytes, not unicode characters.

Returns: a file-like object hooked up to the process' standard error.

Usually, you want to write text to a process's standard error stream ("stderr"), so you want `sys.stderr` to be a stream that accepts Unicode. The function *`pwkit.cli.unicode_stdio()`* sets this up in Python 2, which has an imperfect hack to allow Unicode output to work most of the time. However, there are other times when you really *do* want to write arbitrary binary data to stderr. Depending on whether you're using Python 2 or Python 3, or whether *`pwkit.cli.unicode_stdio()`* has been called, the right way to get access to the underlying byte-based stream is different. This function encapsulates these checks and works across all of these cases.

2.2.3 Other functions in `pwkit.io`

These are generally superseded by operations on *`Path`*.

`pwkit.io.`**`try_open`**`(*args, **kwargs)`
Placeholder.

`pwkit.io.`**`words`**`(linegen)`
Placeholder.

`pwkit.io.`**`pathwords`**`(path, mode='rt', noexistok=False, **kwargs)`
Placeholder.

`pwkit.io.`**`pathlines`**`(path, mode='rt', noexistok=False, **kwargs)`
Placeholder.

`pwkit.io.`**`make_path_func`**`(*baseparts)`
Placeholder.

`pwkit.io.`**`djoin`**`(*args)`
Placeholder.

`pwkit.io.`**`rellink`**`(source, dest)`
Placeholder.

`pwkit.io.`**`ensure_dir`**`(path, parents=False)`
Placeholder.

`pwkit.io.`**`ensure_symlink`**`(src, dst)`
Placeholder.

2.3 Numerical utilities (`pwkit.numutil`)

The `numpy` and `scipy` packages provide a whole host of routines, but there are still some that are missing. The *`pwkit.numutil`* module provides several useful additions.

The functionality in this module can be grouped into these categories:

2.3.1 Making functions that auto-broadcast their arguments

`@pwkit.numutil.`**`broadcastize`**(*n_arr*, *ret_spec=0*, *force_float=True*)

Wrap a function to automatically broadcast `numpy.ndarray` arguments.

It's often desirable to write numerical utility functions in a way that's compatible with vectorized processing. It can be tedious to do this, however, since the function arguments need to turned into arrays and checked for compatible shape, and scalar values need to be special cased.

The `@broadcastize` decorator takes care of these matters. The decorated function can be implemented in vectorized form under the assumption that all array arguments have been broadcast to the same shape. The broadcasting of inputs and (potentially) de-vectorizing of the return values are done automatically. For instance, if you decorate a function `foo(x,y)` with `@numutil.broadcastize(2)`, you can implement it assuming that both *x* and *y* are `numpy.ndarray` objects that have at least one dimension and are both of the same shape. If the function is called with only scalar arguments, *x* and *y* will have shape `(1,)` and the function's return value will be turned back into a scalar before reaching the caller.

The *n_arr* argument specifies the number of array arguments that the function takes. These are required to be at the beginning of its argument list.

The *ret_spec* argument specifies the structure of the function's return value.

- `0` indicates that the value has the same shape as the (broadcasted) vector arguments. If the arguments are all scalar, the return value will be scalar too.
- `1` indicates that the value is an array of higher rank than the input arguments. For instance, if the input has shape `(3,)`, the output might have shape `(4,4,3)`; in general, if the input has shape `s`, the output will have shape `t + s` for some tuple `t`. If the arguments are all scalar, the output will have a shape of just `t`. The `numpy.asarray()` function is called on such arguments, so (for instance) you can return a list of arrays `[a, b]` and it will be converted into a `numpy.ndarray`.
- `None` indicates that the value is completely independ of the inputs. It is returned as-is.
- A tuple `t` indicates that the return value is also a tuple. The elements of the *ret_spec* tuple should contain the values listed above, and each element of the return value will be handled accordingly.

The default *ret_spec* is `0`, i.e. the return value is expected to be an array of the same shape as the argument(s).

If *force_float* is true (the default), the input arrays will be converted to floating-point types if necessary (with `numpy.asfarray()`) before being passed to the function.

Example:

```
@numutil.broadcastize (2, ret_spec=(0, 1, None)):
def myfunction (x, y, extra_arg):
    print ('a random non-vector argument is:', extra_arg)
    z = x + y
    z[np.where (y)] *= 2
    higher_vector = [x, y, z]
    return z, higher_vector, 'hello'
```

2.3.2 Convenience functions for statistics

rms(x)	Return the square root of the mean of the squares of x.
weighted_mean(values, uncerts, **kwargs)	
weighted_mean_df(df, **kwargs)	The same as *weighted_mean()*, except the argument is expected to be a two-column pandas.DataFrame whose first column gives the data values and second column gives their uncertainties.
weighted_variance(x, weights)	Return the variance of a weighted sample.

pwkit.numutil.**rms**(*x*)
Return the square root of the mean of the squares of x.

pwkit.numutil.**weighted_mean**(*values*, *uncerts*, ***kwargs*)

pwkit.numutil.**weighted_mean_df**(*df*, ***kwargs*)
The same as *weighted_mean()*, except the argument is expected to be a two-column pandas.DataFrame whose first column gives the data values and second column gives their uncertainties. Returns (weighted_mean, uncertainty_in_mean).

pwkit.numutil.**weighted_variance**(*x*, *weights*)
Return the variance of a weighted sample.

The weighted sample mean is calculated and subtracted off, so the returned variance is upweighted by n / (n - 1). If the sample mean is known to be zero, you should just compute np.average(x**2, weights=weights).

2.3.3 Convenience functions for pandas.DataFrame objects

reduce_data_frame(df, chunk_slicers[, ...])	"Reduce" a DataFrame by collapsing rows in grouped chunks.
reduce_data_frame_evenly_with_gaps(df, ...)	"Reduce" a DataFrame by collapsing rows in grouped chunks, grouping based on gaps in one of the columns.
slice_around_gaps(values, maxgap)	Given an ordered array of values, generate a set of slices that traverse all of the values.
slice_evenly_with_gaps(values, target_len, ...)	Given an ordered array of values, generate a set of slices that traverse all of the values.
dfsmooth(window, df, ucol[, k])	Smooth a pandas.DataFrame according to a window, weighting based on uncertainties.
smooth_data_frame_with_gaps(window, df, ...)	Smooth a pandas.DataFrame according to a window, weighting based on uncertainties, and breaking the smoothing process at gaps in a time axis.
fits_recarray_to_data_frame(recarray[, ...])	Convert a FITS data table, stored as a Numpy record array, into a Pandas DataFrame object.
data_frame_to_astropy_table(dataframe)	This is a backport of the Astropy method astropy.table.table.Table.from_pandas().
usmooth(window, uncerts, *data, **kwargs)	Smooth data series according to a window, weighting based on uncertainties.
page_data_frame(df[, pager_argv])	Render a DataFrame as text and send it to a terminal pager program (e.g.

pwkit.numutil.**reduce_data_frame**(*df*, *chunk_slicers*, *avg_cols=()*, *uavg_cols=()*, *minmax_cols=()*, *nchunk_colname='nchunk'*, *uncert_prefix='u'*, *min_points_per_chunk=3*)

"Reduce" a DataFrame by collapsing rows in grouped chunks. Returns another DataFrame with similar columns but fewer rows.

Arguments:

df The input `pandas.DataFrame`.

chunk_slicers An iterable that returns values that are used to slice *df* with its `pandas.DataFrame.iloc()` indexer. An example value might be the generator returned from *slice_evenly_with_gaps()*.

avg_cols An iterable of names of columns that are to be reduced by taking the mean.

uavg_cols An iterable of names of columns that are to be reduced by taking a weighted mean.

minmax_cols An iterable of names of columns that are to be reduced by reporting minimum and maximum values.

nchunk_colname The name of a column to create reporting the number of rows contributing to each chunk.

uncert_prefix The column name prefix for locating uncertainty estimates. By default, the uncertainty on the column `"temp"` is given in the column `"utemp"`.

min_points_per_chunk Require at least this many rows in each chunk. Smaller chunks are discarded.

Returns a new `pandas.DataFrame`.

pwkit.numutil.**reduce_data_frame_evenly_with_gaps**(*df*, *valcol*, *target_len*, *maxgap*, ***kwargs*)

"Reduce" a DataFrame by collapsing rows in grouped chunks, grouping based on gaps in one of the columns.

This function combines *reduce_data_frame()* with *slice_evenly_with_gaps()*.

pwkit.numutil.**slice_around_gaps**(*values*, *maxgap*)

Given an ordered array of values, generate a set of slices that traverse all of the values. Within each slice, no gap between adjacent values is larger than *maxgap*. In other words, these slices break the array into chunks separated by gaps of size larger than maxgap.

pwkit.numutil.**slice_evenly_with_gaps**(*values*, *target_len*, *maxgap*)

Given an ordered array of values, generate a set of slices that traverse all of the values. Each slice contains about *target_len* items. However, no slice contains a gap larger than *maxgap*, so a slice may contain only a single item (if it is surrounded on both sides by a large gap). If a non-gapped run of values does not divide evenly into *target_len*, the algorithm errs on the side of making the slices contain more than *target_len* items, rather than fewer. It also attempts to keep the slice size uniform within each non-gapped run.

pwkit.numutil.**dfsmooth**(*window*, *df*, *ucol*, *k=None*)

Smooth a `pandas.DataFrame` according to a window, weighting based on uncertainties.

Arguments are:

window The smoothing window.

df The `pandas.DataFrame`.

ucol The name of the column in *df* that contains the uncertainties to weight by.

k = None If specified, only every *k*-th point of the results will be kept. If k is None (the default), it is set to `window.size`, i.e. correlated points will be discarded.

Returns: a smoothed data frame.

The returned data frame has a default integer index.

Example:

```
sdata = numutil.dfsmooth(np.hamming(7), data, 'u_temp')
```

pwkit.numutil.**fits_recarray_to_data_frame**(*recarray*, *drop_nonscalar_ok=True*)

Convert a FITS data table, stored as a Numpy record array, into a Pandas DataFrame object. By default, non-scalar columns are discarded, but if *drop_nonscalar_ok* is False then a `ValueError` is raised. Column names are lower-cased. Example:

```
from pwkit import io, numutil
hdu_list = io.Path('my-table.fits').read_fits()
# assuming the first FITS extension is a binary table:
df = numutil.fits_recarray_to_data_frame(hdu_list[1].data)
```

FITS data are big-endian, whereas nowadays almost everything is little-endian. This seems to be an issue for Pandas DataFrames, where `df[['col1', 'col2']]` triggers an assertion for me if the underlying data are not native-byte-ordered. This function normalizes the read-in data to native endianness to avoid this.

See also *pwkit.io.Path.read_fits_bintable()*.

pwkit.numutil.**data_frame_to_astropy_table**(*dataframe*)

This is a backport of the Astropy method `astropy.table.table.Table.from_pandas()`. It converts a Pandas `pandas.DataFrame` object to an Astropy `astropy.table.Table`.

pwkit.numutil.**usmooth**(*window*, *uncerts*, **data*, ***kwargs*)

Smooth data series according to a window, weighting based on uncertainties.

Arguments:

window The smoothing window.

uncerts An array of uncertainties used to weight the smoothing.

data One or more data series, of the same size as *uncerts*.

k = None If specified, only every *k*-th point of the results will be kept. If k is None (the default), it is set to `window.size`, i.e. correlated points will be discarded.

Returns: `(s_uncerts, s_data[0], s_data[1], ...)`, the smoothed uncertainties and data series.

Example:

```
u, x, y = numutil.usmooth(np.hamming(7), u, x, y)
```

pwkit.numutil.**page_data_frame**(*df, pager_argv=['less'], **kwargs*)

Render a DataFrame as text and send it to a terminal pager program (e.g. *less*), so that one can browse a full table conveniently.

df The DataFrame to view

pager_argv: default `['less']` A list of strings passed to `subprocess.Popen` that launches the pager program

kwargs Additional keywords are passed to `pandas.DataFrame.to_string()`.

Returns `None`. Execution blocks until the pager subprocess exits.

2.3.4 Parallelized versions of simple math algorithms

parallel_newton(func, x0[, fprime, ...])	A parallelized version of `scipy.optimize.newton()`.
parallel_quad(func, a, b[, par_args, ...])	A parallelized version of `scipy.integrate.quad()`.

pwkit.numutil.**parallel_newton**(*func*, *x0*, *fprime=None*, *par_args=()*, *simple_args=()*, *tol=1.48e-08*, *maxiter=50*, *parallel=True*, ***kwargs*)

A parallelized version of `scipy.optimize.newton()`.

Arguments:

func The function to search for zeros, called as `f(x, [*par_args...], [*simple_args...])`.

x0 The initial point for the zero search.

fprime (Optional) The first derivative of *func*, called the same way.

par_args Tuple of additional parallelized arguments.

simple_args Tuple of additional arguments passed identically to every invocation.

tol The allowable error of the zero value.

maxiter Maximum number of iterations.

parallel Controls parallelization; default uses all available cores. See *pwkit.parallel.make_parallel_helper()*.

kwargs Passed to `scipy.optimize.newton()`.

Returns: an array of locations of zeros.

Finds zeros in parallel. The values *x0*, *tol*, *maxiter*, and the items of *par_args* should all be numeric, and may be N-dimensional Numpy arrays. They are all broadcast to a common shape, and one zero-finding run is performed for each element in the resulting array. The return value is an array of zero locations having the same shape as the common broadcast of the parameters named above.

The *simple_args* are passed to each function identically for each integration. They do not need to be Pickle-able.

Example:

```
>>> parallel_newton(lambda x, a: x - 2 * a, 2,
                    par_args=(np.arange(6),))
<<< array([  0.,   2.,   4.,   6.,   8.,  10.])
>>> parallel_newton(lambda x: np.sin(x), np.arange(6))
<<< array([  0.00000000e+00,   3.65526589e-26,   3.14159265e+00,
             3.14159265e+00,   3.14159265e+00,   6.28318531e+00])
```

pwkit.numutil.**parallel_quad**(*func*, *a*, *b*, *par_args=()*, *simple_args=()*, *parallel=True*, ***kwargs*)

A parallelized version of `scipy.integrate.quad()`.

Arguments are:

func The function to integrate, called as `f(x, [*par_args...], [*simple_args...])`.

a The lower limit(s) of integration.

b The upper limits(s) of integration.

par_args Tuple of additional parallelized arguments.

simple_args Tuple of additional arguments passed identically to every invocation.

parallel Controls parallelization; default uses all available cores. See *pwkit.parallel.make_parallel_helper()*.

kwargs Passed to `scipy.integrate.quad()`. Don't set *full_output* to True.

Returns: integrals and errors; see below.

Computes many integrals in parallel. The values *a*, *b*, and the items of *par_args* should all be numeric, and may be N-dimensional Numpy arrays. They are all broadcast to a common shape, and one integral is performed for each element in the resulting array. If this common shape is (X,Y,Z), the return value has shape (2,X,Y,Z), where the subarray [0,...] contains the computed integrals and the subarray [1,...] contains the absolute error estimates. If *a*, *b*, and the items in *par_args* are all scalars, the return value has shape (2,).

The *simple_args* are passed to each integrand function identically for each integration. They do not need to be Pickle-able.

Example:

```
>>> parallel_quad(lambda x, u, v, q: u * x + v,
                  0, # a
                  [3, 4], # b
                  (np.arange(6).reshape((3,2)), np.arange(3).reshape((3,1))), #␣
↪par_args
                  ('hello',),)
```

Computes six integrals and returns an array of shape `(2,3,2)`. The functions that are evaluated are:

```
[[ 0*x + 0, 1*x + 0 ],
 [ 2*x + 1, 3*x + 1 ],
 [ 4*x + 2, 5*x + 2 ]]
```

and the bounds of the integrals are:

```
[[ (0, 3), (0, 4) ],
 [ (0, 3), (0, 4) ],
 [ (0, 3), (0, 4) ]]
```

In all cases the unused fourth parameter *q* is `'hello'`.

2.3.5 Tophat and step functions

unit_tophat_ee(x)	Tophat function on the unit interval, left-exclusive and right-exclusive.
unit_tophat_ei(x)	Tophat function on the unit interval, left-exclusive and right-inclusive.
unit_tophat_ie(x)	Tophat function on the unit interval, left-inclusive and right-exclusive.
unit_tophat_ii(x)	Tophat function on the unit interval, left-inclusive and right-inclusive.
make_tophat_ee(lower, upper)	Return a ufunc-like tophat function on the defined range, left-exclusive and right-exclusive.
make_tophat_ei(lower, upper)	Return a ufunc-like tophat function on the defined range, left-exclusive and right-inclusive.
make_tophat_ie(lower, upper)	Return a ufunc-like tophat function on the defined range, left-inclusive and right-exclusive.

Continued on next page

Table 11 – continued from previous page

make_tophat_ii(lower, upper)	Return a ufunc-like tophat function on the defined range, left-inclusive and right-inclusive.
make_step_lcont(transition)	Return a ufunc-like step function that is left-continuous.
make_step_rcont(transition)	Return a ufunc-like step function that is right-continuous.

pwkit.numutil.**unit_tophat_ee**(*x*)
Tophat function on the unit interval, left-exclusive and right-exclusive. Returns 1 if 0 < x < 1, 0 otherwise.

pwkit.numutil.**unit_tophat_ei**(*x*)
Tophat function on the unit interval, left-exclusive and right-inclusive. Returns 1 if 0 < x <= 1, 0 otherwise.

pwkit.numutil.**unit_tophat_ie**(*x*)
Tophat function on the unit interval, left-inclusive and right-exclusive. Returns 1 if 0 <= x < 1, 0 otherwise.

pwkit.numutil.**unit_tophat_ii**(*x*)
Tophat function on the unit interval, left-inclusive and right-inclusive. Returns 1 if 0 <= x <= 1, 0 otherwise.

pwkit.numutil.**make_tophat_ee**(*lower*, *upper*)
Return a ufunc-like tophat function on the defined range, left-exclusive and right-exclusive. Returns 1 if lower < x < upper, 0 otherwise.

pwkit.numutil.**make_tophat_ei**(*lower*, *upper*)
Return a ufunc-like tophat function on the defined range, left-exclusive and right-inclusive. Returns 1 if lower < x <= upper, 0 otherwise.

pwkit.numutil.**make_tophat_ie**(*lower*, *upper*)
Return a ufunc-like tophat function on the defined range, left-inclusive and right-exclusive. Returns 1 if lower <= x < upper, 0 otherwise.

pwkit.numutil.**make_tophat_ii**(*lower*, *upper*)
Return a ufunc-like tophat function on the defined range, left-inclusive and right-inclusive. Returns 1 if lower <= x <= upper, 0 otherwise.

pwkit.numutil.**make_step_lcont**(*transition*)
Return a ufunc-like step function that is left-continuous. Returns 1 if x > transition, 0 otherwise.

pwkit.numutil.**make_step_rcont**(*transition*)
Return a ufunc-like step function that is right-continuous. Returns 1 if x >= transition, 0 otherwise.

2.4 Framework for easy parallelized processing (`pwkit.parallel`)

A framework making it easy to write functions that can perform computations in parallel.

Use this framework if you are writing a function that you would like to perform some of its work in parallel, using multiple CPUs at once. First, you must design the parallel part of the function's operation to be implementable in terms of the standard library `map()` function. Then, give your function an optional `parallel=True` keyword argument and use the *make_parallel_helper()* function from this module like so:

```
from pwkit.parallel import make_parallel_helper

def my_parallelizable_function(arg1, arg1, parallel=True):
    # Get a "parallel helper" object that can provide us with a parallelized
    # "map" function. The caller specifies how the parallelization is done;
    # we don't have to know the details.
    phelp = make_parallel_helper(parallel)
```

(continues on next page)

(continued from previous page)

```
    ...

    # When used as a context manager, the helper provides a function that
    # acts like the standard library function "map", except it may
    # parallelize its operation.
    with phelp.get_map() as map:
        results1 = map(my_subfunc1, subargs1)
        ...
        results2 = map(my_subfunc2, subargs2)

    ... do stuff with results1 and results2 ...
```

Passing `parallel=True` to a function defined this way will cause it to parallelize `map` calls across all cores. Passing `parallel=0.5` will cause it to use about half your machine. Passing `parallel=False` will cause it to use serial processing. The helper must be used as a context manager (via the `with` statement) because the parallel computation may involve creating and destroying heavyweight resources (namely, child processes).

Along with standard *ParallelHelper.get_map()*, *ParallelHelper* instances support a "partially-Pickling" *map*-like function *ParallelHelper.get_ppmap()* that works around Pickle-related limitations in the `multiprocessing` library.

2.4.1 Main Interface

The most important parts of this module are the *make_parallel_helper()* function and the interface defined by the abstract *ParallelHelper* class.

make_parallel_helper(parallel_arg, **kwargs)	Return a *ParallelHelper* object that can be used for easy parallelization of computations.
ParallelHelper	Object that helps genericize the setup needed for parallel computations.

pwkit.parallel.**make_parallel_helper**(*parallel_arg*, ***kwargs*)

Return a *ParallelHelper* object that can be used for easy parallelization of computations. *parallel_arg* is an object that lets the caller easily specify the kind of parallelization they are interested in. Allowed values are:

False Serial processing only.

True Parallel processing using all available cores.

1 Equivalent to `False`.

Other positive integer Parallel processing using the specified number of cores.

x, 0 < x < 1 Parallel processing using about `x * N` cores, where N is the total number of cores in the system. Note that the meanings of `0.99` and `1` as arguments are very different.

***ParallelHelper* instance** Returns the instance.

The `**kwargs` are passed on to the appropriate *ParallelHelper* constructor, if the caller wants to do something tricky.

Expected usage is:

```
from pwkit.parallel import make_parallel_helper

def sub_operation(arg):
```

(continues on next page)

(continued from previous page)

```
    ... do some computation ...
    return result

def my_parallelizable_function(arg1, arg2, parallel=True):
    phelp = make_parallel_helper(parallel)

    with phelp.get_map() as map:
        op_results = map(sub_operation, args)

    ... reduce "op_results" in some way ...
    return final_result
```

This means that `my_parallelizable_function` doesn't have to worry about all of the various fancy things the caller might want to do in terms of special parallel magic.

Note that `sub_operation` above must be defined in a stand-alone fashion because of the way Python's `multiprocessing` module works. This can be worked around somewhat with the special *`ParallelHelper.get_ppmap()`* variant. This returns a "partially-Pickling" map operation — with a different calling signature — that allows un-Pickle-able values to be used. See the documentation for *`serial_ppmap()`* for usage information.

class pwkit.parallel.**ParallelHelper**

Object that helps genericize the setup needed for parallel computations. Each method returns a context manager that wraps up any resource allocation and deallocation that may need to occur to make the parallelization happen under the hood.

`ParallelHelper` objects should be obtained by calling *`make_parallel_helper()`*, not direct construction, unless you have special needs. See the documentation of that function for an example of the general usage pattern.

Once you have a *`ParallelHelper`* instance, usage should be something like:

```
with phelp.get_map() as map:
    results_arr = map(my_function, my_args)
```

The partially-Pickling map works around a limitation in the multiprocessing library. This library spawns subprocesses and executes parallel tasks by sending them to the subprocesses, which means that the data describing the task must be pickle-able. There are hacks so that you can pass functions defined in the global namespace but they're pretty much useless in production code. The "partially-Pickling map" works around this by using a different method that allows some arguments to the map operation to avoid being pickled. (Instead, they are directly inherited by `os.fork()`-ed subprocesses.) See the docs for *`serial_ppmap()`* for usage information.

get_map()

Get a *context manager* that yields a function with the same call signature as the standard library function `map()`. Its results are the same, but it may evaluate the mapped function in parallel across multiple threads or processes — the calling function should not have to particularly care about the details. Example usage is:

```
with phelp.get_map() as map:
    results_arr = map(my_function, my_args)
```

The passed function and its arguments must be Pickle-able. The alternate method *`get_ppmap()`* relaxes this restriction somewhat.

get_ppmap()

Get a *context manager* that yields a "partially-pickling map function". It can be used to perform a paral-

lelized `map()` operation with some un-pickle-able arguments.

The yielded function has the signature of *`serial_ppmap()`*. Its behavior is functionally equivalent to the following code, except that the calls to `func` may happen in parallel:

```
def ppmap(func, fixed_arg, var_arg_iter):
    return [func(i, fixed_arg, x) for i, x in enumerate(var_arg_iter)]
```

The arguments to the `ppmap` function are:

func A callable taking three arguments and returning a Pickle-able value.

fixed_arg Any value, even one that is not pickle-able.

var_arg_iter An iterable that generates Pickle-able values.

The arguments to your `func` function, which actually does the interesting computations, are:

index The 0-based index number of the item being processed; often this can be ignored.

fixed_arg The same *fixed_arg* that was passed to `ppmap`.

var_arg The *index*'th item in the *var_arg_iter* iterable passed to `ppmap`.

This variant of the standard `map()` function exists to allow the parallel-processing system to work around `pickle`-related limitations in the `multiprocessing` library.

2.4.2 Implementation Details

Some of these classes and functions may be useful for other modules, but in generally you need only concern yourself with the *`make_parallel_helper()`* function and *`ParallelHelper`* base class.

`SerialHelper`([chunksize])	A *`ParallelHelper`* that actually does serial processing.
`serial_ppmap`(func, fixed_arg, var_arg_iter)	A serial implementation of the "partially-pickling map" function returned by the *`ParallelHelper.get_ppmap()`* interface.
`MultiprocessingPoolHelper`([chunksize])	A *`ParallelHelper`* that parallelizes computations using Python's `multiprocessing.Pool` with a configurable number of processes.
`multiprocessing_ppmap_worker`(in_queue, ...)	Worker for the `multiprocessing` ppmap implementation.
`InterruptiblePool`([processes, initializer, ...])	A modified version of *multiprocessing.pool.Pool* that has better behavior with regard to KeyboardInterrupts in the *map* method.
`VacuousContextManager`(value)	A context manager that just returns a static value and doesn't do anything clever with exceptions.

class pwkit.parallel.**SerialHelper**(*chunksize=None*)

A *`ParallelHelper`* that actually does serial processing.

pwkit.parallel.**serial_ppmap**(*func*, *fixed_arg*, *var_arg_iter*)

A serial implementation of the "partially-pickling map" function returned by the *`ParallelHelper.get_ppmap()`* interface. Its arguments are:

func A callable taking three arguments and returning a Pickle-able value.

fixed_arg Any value, even one that is not pickle-able.

var_arg_iter An iterable that generates Pickle-able values.

The functionality is:

```
def serial_ppmap(func, fixed_arg, var_arg_iter):
    return [func(i, fixed_arg, x) for i, x in enumerate(var_arg_iter)]
```

Therefore the arguments to your `func` function, which actually does the interesting computations, are:

index The 0-based index number of the item being processed; often this can be ignored.

fixed_arg The same *fixed_arg* that was passed to `ppmap`.

var_arg The *index*'th item in the *var_arg_iter* iterable passed to `ppmap`.

class pwkit.parallel.**MultiprocessingPoolHelper**(*chunksize=None*, ***pool_kwargs*)
A *ParallelHelper* that parallelizes computations using Python's `multiprocessing.Pool` with a configurable number of processes. Actually, we use a wrapped version of `multiprocessing.Pool` that handles `KeyboardInterrupt` exceptions more helpfully.

pwkit.parallel.**multiprocessing_ppmap_worker**(*in_queue*, *out_queue*, *func*, *fixed_arg*)
Worker for the `multiprocessing` ppmap implementation. Strongly derived from code posted on StackExchange by "klaus se": http://stackoverflow.com/a/16071616/3760486.

class pwkit.parallel.**InterruptiblePool**(*processes=None*, *initializer=None*, *initargs=()*, ***kwargs*)
A modified version of *multiprocessing.pool.Pool* that has better behavior with regard to KeyboardInterrupts in the *map* method. Parameters:

processes The number of worker processes to use; defaults to the number of CPUs.

initializer Either None, or a callable that will be invoked by each worker process when it starts.

initargs Arguments for *initializer*.

kwargs Extra arguments. Python 2.7 supports a *maxtasksperchild* parameter.

Python's multiprocessing.Pool class doesn't interact well with KeyboardInterrupt signals, as documented in places such as:

- http://stackoverflow.com/questions/1408356/
- http://stackoverflow.com/questions/11312525/
- http://noswap.com/blog/python-multiprocessing-keyboardinterrupt

Various workarounds have been shared. Here, we adapt the one proposed in the last link above, by John Reese, and shared as

- https://github.com/jreese/multiprocessing-keyboardinterrupt/

This version is a drop-in replacement for multiprocessing.Pool ... as long as the map() method is the only one that needs to be interrupt-friendly.

class pwkit.parallel.**VacuousContextManager**(*value*)
A context manager that just returns a static value and doesn't do anything clever with exceptions.

2.5 Quick enumerations of constant values (`pwkit.simpleenum`)

The *pwkit.simpleenum* module contains a single decorator function for creating "enumerations", by which we mean a group of named, un-modifiable values. For example:

```
from pwkit.simpleenum import enumeration

@enumeration
class Constants (object):
  period_days = 2.771
  period_hours = period_days * 24
  n_iters = 300
  # etc

def myfunction ():
  print ('the period is', Constants.period_hours, 'hours')
```

The `class` declaration syntax is handy here because it lets you define new values in relation to old values. In the above example, you cannot change any of the properties of `Constants` once it is constructed.

Important: If you populate an enumeration with a mutable data type, however, we're unable to prevent you from modifying it. For instance, if you do this:

```
@enumeration
class Dangerous (object):
  mutable = [1, 2]
  immutable = (1, 2)
```

You can then do something like write `Dangerous.mutable.append (3)` and modify the value stored in the enumeration. If you're concerned about this, make sure to populate the enumeration with immutable classes such as `tuple`, `frozenset`, `int`, and so on.

---

pwkit.simpleenum.**enumeration**(*cls*)

A very simple decorator for creating enumerations. Unlike Python 3.4 enumerations, this just gives a way to use a class declaration to create an immutable object containing only the values specified in the class.

If the attribute `__pickle_compat__` is set to True in the decorated class, the resulting enumeration value will be callable such that `EnumClass(x) = x`. This is needed to unpickle enumeration values that were previously implemented using `enum.Enum`.

CHAPTER 3

Scientific Algorithms

This documentation has a lot of stubs.

3.1 Basic astronomical calculations (`pwkit.astutil`)

This module provides functions and constants for doing a variety of basic calculations and conversions that come up in astronomy.

This topics covered in this module are:

- *Useful Constants*
- *Sexagesimal Notation*
- *Working with Angles*
- *Simple Operations on 2D Gaussians*
- *Basic Astrometry*
- *Miscellaneous Astronomical Computations*

Angles are **always** measured in radians, whereas some other astronomical codebases prefer degrees.

3.1.1 Useful Constants

The following useful constants are provided:

`pi` Mathematical π.

`twopi` Mathematical 2π.

`halfpi` Mathematical $\pi/2$.

`R2A` A constant for converting radians to arcseconds by multiplication:

```
arcsec = radians * astutil.R2A
```

Equal to `3600 * 180 / pi` or about 206265.

A2R A constant for converting arcseconds to radians by multiplication:

```
radians = arcsec * astutil.A2R
```

R2D Analogous to `R2A`: a constant for converting radians to degrees

D2R Analogous to `A2R`: a constant for converting degrees to radians

R2H Analogous to `R2A`: a constant for converting radians to hours

H2R Analogous to `A2R`: a constant for converting hours to radians

F2S A constant for converting a Gaussian FWHM (full width at half maximum) to a standard deviation (σ) value by multiplication:

```
sigma = fwhm * astutil.F2S
```

Equal to `(8 * ln(2))**-0.5` or about 0.425.

S2F A constant for converting a Gaussian standard deviation (σ) value to a FWHM (full width at half maximum) by multiplication.

J2000 The astronomical J2000.0 epoch as a MJD (modified Julian Date). Precisely equal to 51544.5.

3.1.2 Sexagesimal Notation

fmthours(radians[, norm, precision, seps])	Format an angle as sexagesimal hours in a string.
fmtdeglon(radians[, norm, precision, seps])	Format a longitudinal angle as sexagesimal degrees in a string.
fmtdeglat(radians[, norm, precision, seps])	Format a latitudinal angle as sexagesimal degrees in a string.
fmtradec(rarad, decrad[, precision, raseps, ...])	Format equatorial coordinates in a single sexagesimal string.
parsehours(hrstr)	Parse a string formatted as sexagesimal hours into an angle.
parsedeglat(latstr)	Parse a latitude formatted as sexagesimal degrees into an angle.
parsedeglon(lonstr)	Parse a longitude formatted as sexagesimal degrees into an angle.

pwkit.astutil.**fmthours**(*radians*, *norm='wrap'*, *precision=3*, *seps='::'*)

Format an angle as sexagesimal hours in a string.

Arguments are:

radians The angle, in radians.

norm (default "wrap") The normalization mode, used for angles outside of the standard range of 0 to 2π. If "none", the value is formatted ignoring any potential problems. If "wrap", it is wrapped to lie within the standard range. If "raise", a `ValueError` is raised.

precision (default 3) The number of decimal places in the "seconds" place to use in the formatted string.

seps (default "::") A two- or three-item iterable, used to separate the hours, minutes, and seconds components.

If a third element is present, it appears after the seconds component. Specifying "hms" yields something like "12h34m56s"; specifying `['', '']` yields something like "123456".

Returns a string.

`pwkit.astutil.`**`fmtdeglon`**(*radians*, *norm='wrap'*, *precision=2*, *seps='::'*)
Format a longitudinal angle as sexagesimal degrees in a string.

Arguments are:

radians The angle, in radians.

norm (default "wrap") The normalization mode, used for angles outside of the standard range of 0 to 2π. If "none", the value is formatted ignoring any potential problems. If "wrap", it is wrapped to lie within the standard range. If "raise", a `ValueError` is raised.

precision (default 2) The number of decimal places in the "arcseconds" place to use in the formatted string.

seps (default "::") A two- or three-item iterable, used to separate the degrees, arcminutes, and arcseconds components. If a third element is present, it appears after the arcseconds component. Specifying "dms" yields something like "12d34m56s"; specifying `['', '']` yields something like "123456".

Returns a string.

`pwkit.astutil.`**`fmtdeglat`**(*radians*, *norm='raise'*, *precision=2*, *seps='::'*)
Format a latitudinal angle as sexagesimal degrees in a string.

Arguments are:

radians The angle, in radians.

norm (default "raise") The normalization mode, used for angles outside of the standard range of $-\pi/2$ to $\pi/2$. If "none", the value is formatted ignoring any potential problems. If "wrap", it is wrapped to lie within the standard range. If "raise", a `ValueError` is raised.

precision (default 2) The number of decimal places in the "arcseconds" place to use in the formatted string.

seps (default "::") A two- or three-item iterable, used to separate the degrees, arcminutes, and arcseconds components. If a third element is present, it appears after the arcseconds component. Specifying "dms" yields something like "+12d34m56s"; specifying `['', '']` yields something like "123456".

Returns a string. The return value always includes a plus or minus sign. Note that the default of *norm* is different than in *`fmthours()`* and *`fmtdeglon()`* since it's not so clear what a "latitude" of 110 degrees (e.g.) means.

`pwkit.astutil.`**`fmtradec`**(*rarad*, *decrad*, *precision=2*, *raseps='::'*, *decseps='::'*, *intersep=' '*)
Format equatorial coordinates in a single sexagesimal string.

Returns a string of the RA/lon coordinate, formatted as sexagesimal hours, then *intersep*, then the Dec/lat coordinate, formatted as degrees. This yields something like "12:34:56.78 -01:23:45.6". Arguments are:

rarad The right ascension coordinate, in radians. More generically, this is the longitudinal coordinate; note that the ordering in this function differs than the other spherical functions, which generally prefer coordinates in "lat, lon" order.

decrad The declination coordinate, in radians. More generically, this is the latitudinal coordinate.

precision (default 2) The number of decimal places in the "arcseconds" place of the latitudinal (declination) coordinate. The longitudinal (right ascension) coordinate gets one additional place, since hours are bigger than degrees.

raseps (default "::") A two- or three-item iterable, used to separate the hours, minutes, and seconds components of the RA/lon coordinate. If a third element is present, it appears after the seconds component. Specifying "hms" yields something like "12h34m56s"; specifying `['', '']` yields something like "123456".

decseps (default "::") A two- or three-item iterable, used to separate the degrees, arcminutes, and arcseconds components of the Dec/lat coordinate.

intersep (default " ") The string separating the RA/lon and Dec/lat coordinates

pwkit.astutil.**parsehours**(*hrstr*)
Parse a string formatted as sexagesimal hours into an angle.

This function converts a textual representation of an angle, measured in hours, into a floating point value measured in radians. The format of *hrstr* is very limited: it may not have leading or trailing whitespace, and the components of the sexagesimal representation must be separated by colons. The input must therefore resemble something like `"12:34:56.78"`. A `ValueError` will be raised if the input does not resemble this template. Hours greater than 24 are not allowed, but negative values are.

pwkit.astutil.**parsedeglat**(*latstr*)
Parse a latitude formatted as sexagesimal degrees into an angle.

This function converts a textual representation of a latitude, measured in degrees, into a floating point value measured in radians. The format of *latstr* is very limited: it may not have leading or trailing whitespace, and the components of the sexagesimal representation must be separated by colons. The input must therefore resemble something like `"-00:12:34.5"`. A `ValueError` will be raised if the input does not resemble this template. Latitudes greater than 90 or less than -90 degrees are not allowed.

pwkit.astutil.**parsedeglon**(*lonstr*)
Parse a longitude formatted as sexagesimal degrees into an angle.

This function converts a textual representation of a longitude, measured in degrees, into a floating point value measured in radians. The format of *lonstr* is very limited: it may not have leading or trailing whitespace, and the components of the sexagesimal representation must be separated by colons. The input must therefore resemble something like `"270:12:34.5"`. A `ValueError` will be raised if the input does not resemble this template. Values of any sign and magnitude are allowed, and they are not normalized (e.g. to lie within the range [0, 2π]).

3.1.3 Working with Angles

angcen(a)	
orientcen(a)	
sphdist(lat1, lon1, lat2, lon2)	Calculate the distance between two locations on a sphere.
sphbear(lat1, lon1, lat2, lon2[, tol])	Calculate the bearing between two locations on a sphere.
sphofs(lat1, lon1, r, pa[, tol, rmax])	Offset from one location on the sphere to another.
parang(hourangle, declination, latitude)	Calculate the parallactic angle of a sky position.

pwkit.astutil.**angcen**(*a*)
"Center" an angle *a* to be between -π and +π.

This is done by adding or subtracting multiples of 2π as necessary. Both *a* and the return value are in radians. The argument may be a vector.

pwkit.astutil.**orientcen**(*a*)
"Center" an orientation *a* to be between -π/2 and +π/2.

This is done by adding or subtract multiples of π as necessary. Both *a* and the return value are in radians. The argument may be a vector.

An "orientation" is different than an angle because values that differ by just π, not 2π, are considered equivalent. Orientations can come up in the discussion of linear polarization, for example.

pwkit.astutil.**sphdist**(*lat1*, *lon1*, *lat2*, *lon2*)
Calculate the distance between two locations on a sphere.

lat1 The latitude of the first location.

lon1 The longitude of the first location.

lat2 The latitude of the second location.

lon2 The longitude of the second location.

Returns the separation in radians. All arguments are in radians as well. The arguments may be vectors.

Note that the ordering of the arguments maps to the nonstandard ordering `(Dec, RA)` in equatorial coordinates. In a spherical projection it maps to `(Y, X)` which may also be unexpected.

The distance is computed with the "specialized Vincenty formula". Faster but more error-prone formulae are possible; see Wikipedia on Great-circle Distance.

pwkit.astutil.**sphbear**(*lat1*, *lon1*, *lat2*, *lon2*, *tol=1e-15*)
Calculate the bearing between two locations on a sphere.

lat1 The latitude of the first location.

lon1 The longitude of the first location.

lat2 The latitude of the second location.

lon2 The longitude of the second location.

tol Tolerance for checking proximity to poles and rounding to zero.

The bearing (AKA the position angle, PA) is the orientation of point 2 with regards to point 1 relative to the longitudinal axis. Returns the bearing in radians. All arguments are in radians as well. The arguments may be vectors.

Note that the ordering of the arguments maps to the nonstandard ordering `(Dec, RA)` in equatorial coordinates. In a spherical projection it maps to `(Y, X)` which may also be unexpected.

The sign convention is astronomical: bearings range from -π to π, with negative values if point 2 is in the western hemisphere with regards to point 1, positive if it is in the eastern. (That is, "east from north".) If point 1 is very near the pole, the bearing is undefined and the result is NaN.

The *tol* argument is used for checking proximity to the poles and for rounding the bearing to precisely zero if it's extremely small.

Derived from `bear()` in angles.py from Prasanth Nair. His version is BSD licensed. This one is sufficiently different that I think it counts as a separate implementation.

pwkit.astutil.**sphofs**(*lat1*, *lon1*, *r*, *pa*, *tol=0.01*, *rmax=None*)
Offset from one location on the sphere to another.

This function is given a start location, expressed as a latitude and longitude, a distance to offset, and a direction to offset (expressed as a bearing, AKA position angle). It uses these to compute a final location. This function mirrors *sphdist()* and *sphbear()* such that:

```
# If:
r = sphdist (lat1, lon1, lat2a, lon2a)
pa = sphbear (lat1, lon1, lat2a, lon2a)
lat2b, lon2b = sphofs (lat1, lon1, r, pa)
# Then lat2b = lat2a and lon2b = lon2a
```

Arguments are:

lat1 The latitude of the start location.

lon1 The longitude of the start location.

r The distance to offset by.

pa The position angle ("PA" or bearing) to offset towards.

tol The tolerance for the accuracy of the calculation.

rmax The maximum allowed offset distance.

Returns a pair `(lat2, lon2)`. All arguments and the return values are measured in radians. The arguments may be vectors. The PA sign convention is astronomical, measuring orientation east from north.

Note that the ordering of the arguments and return values maps to the nonstandard ordering `(Dec, RA)` in equatorial coordinates. In a spherical projection it maps to `(Y, X)` which may also be unexpected.

The offset is computed naively as:

```
lat2 = lat1 + r * cos (pa)
lon2 = lon1 + r * sin (pa) / cos (lat2)
```

This will fail for large offsets. Error checking can be done in two ways. If *tol* is not None, *sphdist()* is used to calculate the actual distance between the two locations, and if the magnitude of the fractional difference between that and *r* is larger than *tol*, `ValueError` is raised. This will add an overhead to the computation that may be significant if you're going to be calling this function a lot.

Additionally, if *rmax* is not None, magnitudes of *r* greater than *rmax* are rejected. For reference, an *r* of 0.2 (~11 deg) gives a maximum fractional distance error of ~3%.

pwkit.astutil.**parang**(*hourangle*, *declination*, *latitude*)
Calculate the parallactic angle of a sky position.

This computes the parallactic angle of a sky position expressed in terms of an hour angle and declination. Arguments:

hourangle The hour angle of the location on the sky.

declination The declination of the location on the sky.

latitude The latitude of the observatory.

Inputs and outputs are all in radians. Implementation adapted from GBTIDL `parangle.pro`.

3.1.4 Simple Operations on 2D Gaussians

pwkit.astutil.**gaussian_convolve**(*maj1*, *min1*, *pa1*, *maj2*, *min2*, *pa2*)
Convolve two Gaussians analytically.

Given the shapes of two 2-dimensional Gaussians, this function returns the shape of their convolution.

Arguments:

maj1 Major axis of input Gaussian 1.

min1 Minor axis of input Gaussian 1.

pa1 Orientation angle of input Gaussian 1, in radians.

maj2 Major axis of input Gaussian 2.

min2 Minor axis of input Gaussian 2.

pa2 Orientation angle of input Gaussian 2, in radians.

The return value is `(maj3, min3, pa3)`, with the same format as the input arguments. The axes can be measured in any units, so long as they're consistent.

Implementation copied from MIRIAD's `gaufac`.

`pwkit.astutil.`**`gaussian_deconvolve`**(*smaj*, *smin*, *spa*, *bmaj*, *bmin*, *bpa*)
Deconvolve two Gaussians analytically.

Given the shapes of 2-dimensional "source" and "beam" Gaussians, this returns a deconvolved "result" Gaussian such that the convolution of "beam" and "result" is "source".

Arguments:

smaj Major axis of source Gaussian.

smin Minor axis of source Gaussian.

spa Orientation angle of source Gaussian, in radians.

bmaj Major axis of beam Gaussian.

bmin Minor axis of beam Gaussian.

bpa Orientation angle of beam Gaussian, in radians.

The return value is `(rmaj, rmin, rpa, status)`. The first three values have the same format as the input arguments. The *status* result is one of "ok", "pointlike", or "fail". A "pointlike" status indicates that the source and beam shapes are difficult to distinguish; a "fail" status indicates that the two shapes seem to be mutually incompatible (e.g., source and beam are very narrow and orthogonal).

The axes can be measured in any units, so long as they're consistent.

Ideally if:

```
rmaj, rmin, rpa, status = gaussian_deconvolve (smaj, smin, spa, bmaj, bmin, bpa)
```

then:

```
smaj, smin, spa = gaussian_convolve (rmaj, rmin, rpa, bmaj, bmin, bpa)
```

Implementation derived from MIRIAD's `gaudfac`. This function currently doesn't do a great job of dealing with pointlike sources, i.e. ones where "source" and "beam" are nearly indistinguishable.

3.1.5 Basic Astrometry

The *`AstrometryInfo`* class can be used to perform basic astrometric calculations that are nonetheless fairly accurate.

`class` `pwkit.astutil.`**`AstrometryInfo`**(*simbadident=None*, ***kwargs*)
Holds astrometric data and their uncertainties, and can predict positions with uncertainties.

The attributes encoding the astrometric data are as follows. Values of `None` will be treated as unknown. Most of this information can be automatically filled in from the *`fill_from_simbad()`* function, if you trust Simbad.

`ra`	The J2000 right ascension of the object, measured in radians.
`dec`	The J2000 declination of the object, measured in radians.
`pos_u_maj`	Major axis of the error ellipse for the object position, in radians.

Continued on next page

Table 3 – continued from previous page

pos_u_min	Minor axis of the error ellipse for the object position, in radians.
pos_u_pa	Position angle (really orientation) of the error ellipse for the object position, east from north, in radians.
pos_epoch	The epoch of position, that is, the date when the position was measured, in MJD[TT].
promo_ra	The proper motion in right ascension, in milliarcsec per year.
promo_dec	The object's proper motion in declination, in milliarcsec per year.
promo_u_maj	Major axis of the error ellipse for the object's proper motion, in milliarcsec per year.
promo_u_min	Minor axis of the error ellipse for the object's proper motion, in milliarcsec per year.
promo_u_pa	Position angle (really orientation) of the error ellipse for the object proper motion, east from north, in radians.
parallax	The object's parallax, in milliarcsec.
u_parallax	Uncertainty in the object's parallax, in milliarcsec.
vradial	The object's radial velocity, in km/s.
u_vradial	The uncertainty in the object's radial velocity, in km/s.

Methods are:

verify([complain])	Validate that the attributes are self-consistent.
predict(mjd[, complain, n])	Predict the object position at a given MJD.
print_prediction(ptup[, precision])	Print a summary of a predicted position.
predict_without_uncertainties(mjd[, complain])	Predict the object position at a given MJD.
fill_from_simbad(ident[, debug])	Fill in astrometric information using the Simbad web service.
fill_from_allwise(ident[, catalog_ident])	Fill in astrometric information from the AllWISE catalog using Astroquery.

The stringification of an *AstrometryInfo* class formats its fields in a human-readable, multiline format that uses Unicode characters.

AstrometryInfo.**ra = None**
The J2000 right ascension of the object, measured in radians.

AstrometryInfo.**dec = None**
The J2000 declination of the object, measured in radians.

AstrometryInfo.**pos_u_maj = None**
Major axis of the error ellipse for the object position, in radians.

AstrometryInfo.**pos_u_min = None**
Minor axis of the error ellipse for the object position, in radians.

AstrometryInfo.**pos_u_pa = None**
Position angle (really orientation) of the error ellipse for the object position, east from north, in radians.

AstrometryInfo.**pos_epoch = None**
The epoch of position, that is, the date when the position was measured, in MJD[TT].

AstrometryInfo.**promo_ra = None**
The proper motion in right ascension, in milliarcsec per year. XXX: cos(dec) confusion!

AstrometryInfo.**promo_dec = None**
The object's proper motion in declination, in milliarcsec per year.

AstrometryInfo.**promo_u_maj = None**
Major axis of the error ellipse for the object's proper motion, in milliarcsec per year.

AstrometryInfo.**promo_u_min = None**
Minor axis of the error ellipse for the object's proper motion, in milliarcsec per year.

AstrometryInfo.**promo_u_pa = None**
Position angle (really orientation) of the error ellipse for the object proper motion, east from north, in radians.

AstrometryInfo.**parallax = None**
The object's parallax, in milliarcsec.

AstrometryInfo.**u_parallax = None**
Uncertainty in the object's parallax, in milliarcsec.

AstrometryInfo.**vradial = None**
The object's radial velocity, in km/s. NOTE: not relevant in our usage.

AstrometryInfo.**u_vradial = None**
The uncertainty in the object's radial velocity, in km/s. NOTE: not relevant in our usage.

AstrometryInfo.**verify**(*complain=True*)
Validate that the attributes are self-consistent.

This function does some basic checks of the object attributes to ensure that astrometric calculations can legally be performed. If the *complain* keyword is true, messages may be printed to `sys.stderr` if non-fatal issues are encountered.

Returns *self*.

AstrometryInfo.**predict**(*mjd*, *complain=True*, *n=20000*)
Predict the object position at a given MJD.

The return value is a tuple `(ra, dec, major, minor, pa)`, all in radians. These are the predicted position of the object and its uncertainty at *mjd*. If *complain* is True, print out warnings for incomplete information. *n* is the number of Monte Carlo samples to draw for computing the positional uncertainty.

The uncertainty ellipse parameters are sigmas, not FWHM. These may be converted with the `S2F` constant.

This function relies on the external `skyfield` package.

AstrometryInfo.**print_prediction**(*ptup*, *precision=2*)
Print a summary of a predicted position.

The argument *ptup* is a tuple returned by *`predict()`*. It is printed to `sys.stdout` in a reasonable format that uses Unicode characters.

AstrometryInfo.**predict_without_uncertainties**(*mjd*, *complain=True*)
Predict the object position at a given MJD.

The return value is a tuple `(ra, dec)`, in radians, giving the predicted position of the object at *mjd*. Unlike *`predict()`*, the astrometric uncertainties are ignored. This function is therefore deterministic but potentially misleading.

If *complain* is True, print out warnings for incomplete information.

This function relies on the external `skyfield` package.

AstrometryInfo.**fill_from_simbad**(*ident*, *debug=False*)
Fill in astrometric information using the Simbad web service.

This uses the CDS Simbad web service to look up astrometric information for the source name *ident* and fills in attributes appropriately. Values from Simbad are not always reliable.

Returns *self*.

AstrometryInfo.**fill_from_allwise**(*ident*, *catalog_ident='II/328/allwise'*)
Fill in astrometric information from the AllWISE catalog using Astroquery.

This uses the `astroquery` module to query the AllWISE (2013wise.rept....1C) source catalog through the Vizier (2000A&AS..143...23O) web service. It then fills in the instance with the relevant information. Arguments are:

ident The AllWISE catalog identifier of the form `"J112254.70+255021.9"`.

catalog_ident The Vizier designation of the catalog to query. The default is "II/328/allwise", the current version of the AllWISE catalog.

Raises *PKError* if something unexpected happens that doesn't itself result in an exception within `astroquery`.

You should probably prefer *fill_from_simbad()* for objects that are known to the CDS Simbad service, but not all objects in the AllWISE catalog are so known.

If you use this function, you should acknowledge AllWISE and Vizier.

Returns *self*.

A few helper functions may also be of interest:

load_skyfield_data()	Load data files used in Skyfield.
get_2mass_epoch(tmra, tmdec[, debug])	Given a 2MASS position, look up the epoch when it was observed.
get_simbad_astrometry_info(ident[, items, debug])	Fetch astrometric information from the Simbad web service.

pwkit.astutil.**load_skyfield_data**()
Load data files used in Skyfield. This will download files from the internet if they haven't been downloaded before.

Skyfield downloads files to the current directory by default, which is not ideal. Here we abuse astropy and use its cache directory to cache the data files per-user. If we start downloading files in other places in pwkit we should maybe make this system more generic. And the dep on astropy is not at all necessary.

Skyfield will print out a progress bar as it downloads things.

Returns `(planets, ts)`, the standard Skyfield ephemeris and timescale data files.

pwkit.astutil.**get_2mass_epoch**(*tmra*, *tmdec*, *debug=False*)
Given a 2MASS position, look up the epoch when it was observed.

This function uses the CDS Vizier web service to look up information in the 2MASS point source database. Arguments are:

tmra The source's J2000 right ascension, in radians.

tmdec The source's J2000 declination, in radians.

debug If True, the web server's response will be printed to `sys.stdout`.

The return value is an MJD. If the lookup fails, a message will be printed to `sys.stderr` (unconditionally!) and the `J2000` epoch will be returned.

`pwkit.astutil.`**`get_simbad_astrometry_info`**(*ident*, *items=['COO(d;A)', 'COO(d;D)', 'COO(E)', 'COO(B)', 'PM(A)', 'PM(D)', 'PM(E)', 'PLX(V)', 'PLX(E)', 'RV(V)', 'RV(E)']*, *debug=False*)

Fetch astrometric information from the Simbad web service.

Given the name of a source as known to the CDS Simbad service, this function looks up its positional information and returns it in a dictionary. In most cases you should use an *`AstrometryInfo`* object and its *`fill_from_simbad()`* method instead of this function.

Arguments:

ident The Simbad name of the source to look up.

items An iterable of data items to look up. The default fetches position, proper motion, parallax, and radial velocity information. Each item name resembles the string `COO(d;A)` or `PLX(E)`. The allowed formats are defined on this CDS page.

debug If true, the response from the webserver will be printed.

The return value is a dictionary with a key corresponding to the textual result returned for each requested item.

3.1.6 Miscellaneous Astronomical Computations

These functions don't fit under the other rubrics very well.

`abs2app`(abs_mag, dist_pc)	Convert an absolute magnitude to an apparent magnitude, given a source's (luminosity) distance in parsecs.
`app2abs`(app_mag, dist_pc)	Convert an apparent magnitude to an absolute magnitude, given a source's (luminosity) distance in parsecs.

`pwkit.astutil.`**`abs2app`**(*abs_mag*, *dist_pc*)

Convert an absolute magnitude to an apparent magnitude, given a source's (luminosity) distance in parsecs.

Arguments:

abs_mag Absolute magnitude.

dist_pc Distance, in parsecs.

Returns the apparent magnitude. The arguments may be vectors.

`pwkit.astutil.`**`app2abs`**(*app_mag*, *dist_pc*)

Convert an apparent magnitude to an absolute magnitude, given a source's (luminosity) distance in parsecs.

Arguments:

app_mag Apparent magnitude.

dist_pc Distance, in parsecs.

Returns the absolute magnitude. The arguments may be vectors.

3.2 File-format-agnostic loading of astronomical images (pwkit.astimage)

pwkit.astimage – generic loading of (radio) astronomical images

Use *open (path, mode)* to open an astronomical image, regardless of its file format.

The emphasis of this module is on getting 90%-good-enough semantics and a really, genuinely, uniform interface. This can be tough to achieve.

exception pwkit.astimage.**UnsupportedError**(*fmt*, **args*)

class pwkit.astimage.**AstroImage**(*path*, *mode*)

An astronomical image.

path The filesystem path of the image.

mode Its access mode: 'r' for read, 'rw' for read/write.

shape The data shape, like numpy.ndarray.shape.

bmaj If not None, the restoring beam FWHM major axis in radians.

bmin If not None, the restoring beam FWHM minor axis in radians.

bpa If not None, the restoring beam position angle (east from celestial north) in radians.

units Lower-case string describing image units (e.g., jy/beam, jy/pixel). Not standardized between formats.

pclat Latitude (usually dec) of the pointing center in radians.

pclon Longitude (usually RA) of the pointing center in radians.

charfreq Characteristic observing frequency of the image in GHz.

mjd Mean MJD of the observations.

axdescs If not None, list of strings describing the axis types. Not standardized.

size The number of pixels in the image (=shape.prod ()).

Methods:

close Close the image.

read Read all of the data.

write Rewrite all of the data.

toworld Convert pixel coordinates to world coordinates.

topixel Convert world coordinates to pixel coordinates.

simple Convert to a 2D lat/lon image.

subimage Extract a sub-cube of the image.

save_copy Save a copy of the image.

save_as_fits Save a copy of the image in FITS format.

delete Delete the on-disk image.

axdescs = None

If not None, list of strings describing the axis types; no standard format.

bmaj = None

If not None, the restoring beam FWHM major axis in radians

bmin = None
If not None, the restoring beam FWHM minor axis in radians

bpa = None
If not None, the restoring beam position angle (east from celestial north) in radians

charfreq = None
Characteristic observing frequency of the image in GHz

mjd = None
Mean MJD of the observations

pclat = None
Latitude of the pointing center in radians

pclon = None
Longitude of the pointing center in radians

shape = None
An integer ndarray of the image shape

subimage(*pixofs*, *shape*)
Extract a sub-cube of this image.

Both *pixofs* and *shape* should be integer arrays with as many elements as this image has axes. Thinking of this operation as taking a Python slice of an N-dimensional cube, the i'th axis of the sub-image is slices from *pixofs[i]* to *pixofs[i] + shape[i]*.

units = None
Lower-case string describing image units (e.g., jy/beam, jy/pixel)

class pwkit.astimage.**MIRIADImage**(*path*, *mode*)
A MIRIAD format image. Requires the *mirtask* module from miriad-python.

class pwkit.astimage.**PyrapImage**(*path*, *mode*)
A CASA-format image loaded with the 'pyrap' Python module.

class pwkit.astimage.**FITSImage**(*path*, *mode*)
A FITS format image.

class pwkit.astimage.**SimpleImage**(*parent*)
A 2D, latitude/longitude image, referenced to a parent image.

3.3 The Bayesian Blocks algorithm (pwkit.bblocks)

pwkit.bblocks - Bayesian Blocks analysis, with a few extensions.

Bayesian Blocks analysis for the "time tagged" case described by Scargle+ 2013. Inspired by the bayesian_blocks implementation by Jake Vanderplas in the AstroML package, but that turned out to have some limitations.

We have iterative determination of the best number of blocks (using an ad-hoc routine described in Scargle+ 2013) and bootstrap-based determination of uncertainties on the block heights (ditto).

Functions are:

bin_bblock() Bayesian Blocks analysis with counts and bins.

tt_bblock() BB analysis of time-tagged events.

bs_tt_bblock() Like *tt_bblock()* with bootstrap-based uncertainty assessment. NOTE: the uncertainties are not very reliable!

pwkit.bblocks.**bin_bblock**(*widths*, *counts*, *p0=0.05*)
Fundamental Bayesian Blocks algorithm. Arguments:

widths - Array of consecutive cell widths. counts - Array of numbers of counts in each cell. p0=0.05 - Probability of preferring solutions with additional bins.

Returns a Holder with:

blockstarts - Start times of output blocks. counts - Number of events in each output block. finalp0 - Final value of p0, after iteration to minimize *nblocks*. nblocks - Number of output blocks. ncells - Number of input cells/bins. origp0 - Original value of p0. rates - Event rate associated with each block. widths - Width of each output block.

pwkit.bblocks.**tt_bblock**(*tstarts*, *tstops*, *times*, *p0=0.05*, *intersect_with_bins=False*)
Bayesian Blocks for time-tagged events. Arguments:

tstarts Array of input bin start times.

tstops Array of input bin stop times.

times Array of event arrival times.

***p0* = 0.05** Probability of preferring solutions with additional bins.

***intersect_with_bins* = False** If true, intersect bblock bins with input bins; can result in more bins than bblocks wants; they will have the same rate values.

Returns a Holder with:

counts Number of events in each output block.

finalp0 Final value of p0, after iteration to minimize *nblocks*.

ledges Times of left edges of output blocks.

midpoints Times of midpoints of output blocks.

nblocks Number of output blocks.

ncells Number of input cells/bins.

origp0 Original value of p0.

rates Event rate associated with each block.

redges Times of right edges of output blocks.

widths Width of each output block.

Bin start/stop times are best derived from a 1D Voronoi tesselation of the event arrival times, with some kind of global observation start/stop time setting the extreme edges. Or they can be set from "good time intervals" if observations were toggled on or off as in an X-ray telescope.

If *intersect_with_bins* is True, the true Bayesian Blocks bins (BBBs) are intersected with the "good time intervals" (GTIs) defined by the *tstarts* and *tstops* variables. One GTI may contain multiple BBBs if the event rate appears to change within the GTI, and one BBB may span multiple GTIs if the event date does *not* appear to change between the GTIs. The intersection will ensure that no BBB intervals cross the edge of a GTI. If this would happen, the BBB is split into multiple, partially redundant records. Each of these records will have the **same** value for the *counts*, *rates*, and *widths* values. However, the *ledges*, *redges*, and *midpoints* values will be recalculated. Note that in this mode, it is not necessarily true that `widths = ledges - redges` as is usually the case. When this flag is true, keep in mind that multiple bins are therefore *not* necessarily independent statistical samples.

pwkit.bblocks.**bs_tt_bblock**(*times*, *tstarts*, *tstops*, *p0=0.05*, *nbootstrap=512*)

Bayesian Blocks for time-tagged events with bootstrapping uncertainty assessment. THE UNCERTAINTIES ARE NOT VERY GOOD! Arguments:

tstarts - Array of input bin start times. tstops - Array of input bin stop times. times - Array of event arrival times. p0=0.05 - Probability of preferring solutions with additional bins. nbootstrap=512 - Number of bootstrap runs to perform.

Returns a Holder with:

blockstarts - Start times of output blocks. bsrates - Mean event rate in each bin from bootstrap analysis. bsrstds - ~Uncertainty: stddev of event rate in each bin from bootstrap analysis. counts - Number of events in each output block. finalp0 - Final value of p0, after iteration to minimize *nblocks*. ledges - Times of left edges of output blocks. midpoints - Times of midpoints of output blocks. nblocks - Number of output blocks. ncells - Number of input cells/bins. origp0 - Original value of p0. rates - Event rate associated with each block. redges - Times of right edges of output blocks. widths - Width of each output block.

3.4 Constants in CGS units (pwkit.cgs)

pwkit.cgs - Physical constants in CGS.

Specifically, ESU-CGS im which the electron charge is measured in esu Franklin statcoulomb.

a0 - Bohr radius [cm] alpha - Fine structure constant [ø] arad - Radiation constant [erg/cm^3/K^4] aupercm - AU per cm c - Speed of light [cm/s] cgsperjy - [erg/s/cm^2/Hz] per Jy cmperau - cm per AU cmperpc - cm per parsec conjaaev - eV/Angstrom conjugation factor: AA = conjaaev / eV [Å·eV] e - electron charge [esu] ergperev - erg per eV euler - Euler's constant (2.71828...) [ø] evpererg - eV per erg G - Gravitational constant [cm^3/g/s^2] h - Planck's constant [erg s] hbar - Reduced Planck's constant [erg·s] jypercgs - Jy per [erg/s/cm^2/Hz] k - Boltzmann's constant [erg/K] lsun - Luminosity of the Sun [erg/s] me - Mass of the electron [g] mearth - Mass of the Earth [g] mjup - Mass of Jupiter [g] mp - Mass of the proton [g] msun - Mass of the Sun [g] mu_e - Magnetic moment of the electron [esu·cm^2/s] pcpercm - parsec per cm pi - Pi [ø] r_e - Classical radius of the electron [cm] rearth - Radius of the earth [cm] rjup - Radius of Jupiter [cm] rsun - Radius of the Sun [cm] ryd1 - Rydberg energy [erg] sigma - Stefan-Boltzmann constant [erg/s/K^4] sigma_T - Thomson cross section of the electron [cm^2] spersyr - Seconds per sidereal year syrpers - Sidereal years per second tsun - Effective temperature of the Sun [K]

Functions:

blambda - Planck function (Hz, K) -> erg/s/cm^2/Hz/sr. bnu - Planck function (cm, K) -> erg/s/cm^2/cm/sr. exp - Numpy exp() function. log - Numpy log() function. log10 - Numpy log10() function. sqrt - Numpy sqrt() function.

For reference: the esu has dimensions of g^(1/2) cm^(3/2) s^-1. Electric and magnetic field have g^(1/2) cm^(-1/2) s^-1. [esu * field] = dyne.

3.5 Simple synchrotron radiation emission coefficients (pwkit.dulk_models)

Model radio-wavelength radiative transfer using the Dulk (1985) equations.

Note that the gyrosynchrotron and relativistic synchrotron expressions can give *very* different answers! For s=20, delta=3, theta=0.7, the results differ by three orders of magnitude for η! The paper is a bit vague but mentions that the gyrosynchrotron case is when "γ <~ 2 or 3". The synchrotron functions give results compatible with Symphony/Rimphony; the gyrosynchrotron ones do not, although I've only tentatively explored what happens if you given Symphony/Rimphony very low cuts on their gamma values.

The models are from Dulk (1985; 1985ARA&A..23..169D; doi:10.1146/annurev.aa.23.090185.001125). There are three versions:

- *Free-free emission*
- *Gyrosynchrotron emission*
- *Relativistic synchrotron emission*
- *Helpers*

3.5.1 Free-free emission

calc_freefree_kappa(ne, t, hz)	Dulk (1985) eq 20, assuming pure hydrogen.
calc_freefree_eta(ne, t, hz)	Dulk (1985) equations 7 and 20, assuming pure hydrogen.
calc_freefree_snu_ujy(ne, t, width, ...)	Calculate a flux density from pure free-free emission.

pwkit.dulk_models.**calc_freefree_kappa**(*ne*, *t*, *hz*)
Dulk (1985) eq 20, assuming pure hydrogen.

pwkit.dulk_models.**calc_freefree_eta**(*ne*, *t*, *hz*)
Dulk (1985) equations 7 and 20, assuming pure hydrogen.

pwkit.dulk_models.**calc_freefree_snu_ujy**(*ne*, *t*, *width*, *elongation*, *dist*, *ghz*)
Calculate a flux density from pure free-free emission.

3.5.2 Gyrosynchrotron emission

calc_gs_kappa(b, ne, delta, sinth, nu)	Calculate the gyrosynchrotron absorption coefficient κ_ν.
calc_gs_eta(b, ne, delta, sinth, nu)	Calculate the gyrosynchrotron emission coefficient η_ν.
calc_gs_snu_ujy(b, ne, delta, sinth, width, ...)	Calculate a flux density from pure gyrosynchrotron emission.

pwkit.dulk_models.**calc_gs_kappa**(*b*, *ne*, *delta*, *sinth*, *nu*)
Calculate the gyrosynchrotron absorption coefficient κ_ν.

This is Dulk (1985) equation 36, which is a fitting function assuming a power-law electron population. Arguments are:

b Magnetic field strength in Gauss

ne The density of electrons per cubic centimeter with energies greater than 10 keV.

delta The power-law index defining the energy distribution of the electron population, with `n(E) ~ E^(-delta)`. The equation is valid for `2 <~ delta <~ 7`.

sinth The sine of the angle between the line of sight and the magnetic field direction. The equation is valid for $\theta > 20°$ or `sinth > 0.34` or so.

nu The frequency at which to calculate η, in Hz. The equation is valid for `10 <~ nu/nu_b <~ 100`, which sets a limit on the ratio of `nu` and `b`.

The return value is the absorption coefficient, in units of `cm^-1`.

No complaints are raised if you attempt to use the equation outside of its range of validity.

pwkit.dulk_models.**calc_gs_eta**(*b*, *ne*, *delta*, *sinth*, *nu*)
Calculate the gyrosynchrotron emission coefficient η_ν.

This is Dulk (1985) equation 35, which is a fitting function assuming a power-law electron population. Arguments are:

b Magnetic field strength in Gauss

ne The density of electrons per cubic centimeter with energies greater than 10 keV.

delta The power-law index defining the energy distribution of the electron population, with `n(E) ~ E^(-delta)`. The equation is valid for `2 <~ delta <~ 7`.

sinth The sine of the angle between the line of sight and the magnetic field direction. The equation is valid for $\theta > 20°$ or `sinth > 0.34` or so.

nu The frequency at which to calculate η, in Hz. The equation is valid for `10 <~ nu/nu_b <~ 100`, which sets a limit on the ratio of `nu` and `b`.

The return value is the emission coefficient (AKA "emissivity"), in units of `erg s^-1 Hz^-1 cm^-3 sr^-1`.

No complaints are raised if you attempt to use the equation outside of its range of validity.

pwkit.dulk_models.**calc_gs_snu_ujy**(*b*, *ne*, *delta*, *sinth*, *width*, *elongation*, *dist*, *ghz*)
Calculate a flux density from pure gyrosynchrotron emission.

This combines Dulk (1985) equations 35 and 36, which are fitting functions assuming a power-law electron population, with standard radiative transfer through a uniform medium. Arguments are:

b Magnetic field strength in Gauss

ne The density of electrons per cubic centimeter with energies greater than 10 keV.

delta The power-law index defining the energy distribution of the electron population, with `n(E) ~ E^(-delta)`. The equation is valid for `2 <~ delta <~ 7`.

sinth The sine of the angle between the line of sight and the magnetic field direction. The equation is valid for $\theta > 20°$ or `sinth > 0.34` or so.

width The characteristic cross-sectional width of the emitting region, in cm.

elongation The the elongation of the emitting region; `depth = width * elongation`.

dist The distance to the emitting region, in cm.

ghz The frequencies at which to evaluate the spectrum, **in GHz**.

The return value is the flux density **in** μ**Jy**. The arguments can be Numpy arrays.

No complaints are raised if you attempt to use the equations outside of their range of validity.

3.5.3 Relativistic synchrotron emission

calc_synch_kappa(b, ne, delta, sinth, nu[, E0])	Calculate the relativstic synchrotron absorption coefficient κ_ν.
calc_synch_eta(b, ne, delta, sinth, nu[, E0])	Calculate the relativistic synchrotron emission coefficient η_ν.
calc_synch_snu_ujy(b, ne, delta, sinth, ...)	Calculate a flux density from pure gyrosynchrotron emission.

pwkit.dulk_models.**calc_synch_kappa**(*b*, *ne*, *delta*, *sinth*, *nu*, *E0=1.0*)
Calculate the relativstic synchrotron absorption coefficient κ_ν.

This is Dulk (1985) equation 41, which is a fitting function assuming a power-law electron population. Arguments are:

b Magnetic field strength in Gauss

ne The density of electrons per cubic centimeter with energies greater than E0.

delta The power-law index defining the energy distribution of the electron population, with `n(E) ~ E^(-delta)`. The equation is valid for `2 <~ delta <~ 5`.

sinth The sine of the angle between the line of sight and the magnetic field direction. It's not specified for what range of values the expressions work well.

nu The frequency at which to calculate η, in Hz. The equation is valid for It's not specified for what range of values the expressions work well.

E0 The minimum energy of electrons to consider, in MeV. Defaults to 1 so that these functions can be called identically to the gyrosynchrotron functions.

The return value is the absorption coefficient, in units of `cm^-1`.

No complaints are raised if you attempt to use the equation outside of its range of validity.

pwkit.dulk_models.**calc_synch_eta**(*b*, *ne*, *delta*, *sinth*, *nu*, *E0=1.0*)
Calculate the relativistic synchrotron emission coefficient η_ν.

This is Dulk (1985) equation 40, which is an approximation assuming a power-law electron population. Arguments are:

b Magnetic field strength in Gauss

ne The density of electrons per cubic centimeter with energies greater than E0.

delta The power-law index defining the energy distribution of the electron population, with `n(E) ~ E^(-delta)`. The equation is valid for `2 <~ delta <~ 5`.

sinth The sine of the angle between the line of sight and the magnetic field direction. It's not specified for what range of values the expressions work well.

nu The frequency at which to calculate η, in Hz. The equation is valid for It's not specified for what range of values the expressions work well.

E0 The minimum energy of electrons to consider, in MeV. Defaults to 1 so that these functions can be called identically to the gyrosynchrotron functions.

The return value is the emission coefficient (AKA "emissivity"), in units of `erg s^-1 Hz^-1 cm^-3 sr^-1`.

No complaints are raised if you attempt to use the equation outside of its range of validity.

pwkit.dulk_models.**calc_synch_snu_ujy**(*b*, *ne*, *delta*, *sinth*, *width*, *elongation*, *dist*, *ghz*, *E0=1.0*)
Calculate a flux density from pure gyrosynchrotron emission.

This combines Dulk (1985) equations 40 and 41, which are fitting functions assuming a power-law electron population, with standard radiative transfer through a uniform medium. Arguments are:

b Magnetic field strength in Gauss

ne The density of electrons per cubic centimeter with energies greater than 10 keV.

delta The power-law index defining the energy distribution of the electron population, with `n(E) ~ E^(-delta)`. The equation is valid for `2 <~ delta <~ 5`.

sinth The sine of the angle between the line of sight and the magnetic field direction. It's not specified for what range of values the expressions work well.

width The characteristic cross-sectional width of the emitting region, in cm.

elongation The the elongation of the emitting region; `depth = width * elongation`.

dist The distance to the emitting region, in cm.

ghz The frequencies at which to evaluate the spectrum, **in GHz**.

E0 The minimum energy of electrons to consider, in MeV. Defaults to 1 so that these functions can be called identically to the gyrosynchrotron functions.

The return value is the flux density **in** μ**Jy**. The arguments can be Numpy arrays.

No complaints are raised if you attempt to use the equations outside of their range of validity.

3.5.4 Helpers

calc_nu_b(b)	Calculate the cyclotron frequency in Hz given a magnetic field strength in Gauss.
calc_snu(eta, kappa, width, elongation, dist)	Calculate the flux density S_ν given a simple physical configuration.

pwkit.dulk_models.**calc_nu_b**(*b*)
Calculate the cyclotron frequency in Hz given a magnetic field strength in Gauss.

This is in cycles per second not radians per second; i.e. there is a 2π in the denominator: ν_B = e B / (2π m_e c)

pwkit.dulk_models.**calc_snu**(*eta*, *kappa*, *width*, *elongation*, *dist*)
Calculate the flux density S_ν given a simple physical configuration.

This is basic radiative transfer as per Dulk (1985) equations 5, 6, and 11.

eta The emissivity, in units of `erg s^-1 Hz^-1 cm^-3 sr^-1`.

kappa The absorption coefficient, in units of `cm^-1`.

width The characteristic cross-sectional width of the emitting region, in cm.

elongation The the elongation of the emitting region; `depth = width * elongation`.

dist The distance to the emitting region, in cm.

The return value is the flux density, in units of `erg s^-1 cm^-2 Hz^-1`. The angular size of the source is taken to be `(width / dist)**2`.

3.6 Representations of and computations with ellipses (pwkit.ellipses)

pwkit.ellipses - utilities for manipulating 2D Gaussians and ellipses

XXXXXXX XXX this code is in an incomplete state of being vectorized!!! XXXXXXX

Useful for sources and bivariate error distributions. We can express the shape of the function in several ways, which have different strengths and weaknesses:

- "biv", as in Gaussian bivariate: sigma x, sigma y, cov(x,y)
- "ell", as in ellipse: major, minor, PA [*]
- "abc": coefficients such that z = exp (ax^2 + bxy + cy^2)

[*] Any slice through a 2D Gaussian is an ellipse. Ours is defined such it is the same as a Gaussian bivariate when major = minor.

Note that when considering astronomical position angles, conventionally defined as East from North, the Dec/lat axis should be considered the X axis and the RA/long axis should be considered the Y axis.

pwkit.ellipses.**sigmascale**(*nsigma*)
: Say we take a Gaussian bivariate and convert the parameters of the distribution to an ellipse (major, minor, PA). By what factor should we scale those axes to make the area of the ellipse correspond to the n-sigma confidence interval?

  Negative or zero values result in NaN.

pwkit.ellipses.**clscale**(*cl*)
: Say we take a Gaussian bivariate and convert the parameters of the distribution to an ellipse (major, minor, PA). By what factor should we scale those axes to make the area of the ellipse correspond to the confidence interval CL? (I.e. 0 < CL < 1)

pwkit.ellipses.**bivell**(*sx*, *sy*, *cxy*)
: Given the parameters of a Gaussian bivariate distribution, compute the parameters for the equivalent 2D Gaussian in ellipse form (major, minor, pa).

  Inputs:

  - sx: standard deviation (not variance) of x var
  - sy: standard deviation (not variance) of y var
  - cxy: covariance (not correlation coefficient) of x and y

  Outputs:

  - mjr: major axis of equivalent 2D Gaussian (sigma, not FWHM)
  - mnr: minor axis
  - pa: position angle, rotating from +x to +y

  Lots of sanity-checking because it's obnoxiously easy to have numerics that just barely blow up on you.

pwkit.ellipses.**bivnorm**(*sx*, *sy*, *cxy*)
: Given the parameters of a Gaussian bivariate distribution, compute the correct normalization for the equivalent 2D Gaussian. It's 1 / (2 pi sqrt (sx**2 sy**2 - cxy**2). This function adds a lot of sanity checking.

  Inputs:

  - sx: standard deviation (not variance) of x var
  - sy: standard deviation (not variance) of y var
  - cxy: covariance (not correlation coefficient) of x and y

  Returns: the scalar normalization

pwkit.ellipses.**bivabc**(*sx*, *sy*, *cxy*)
: Compute nontrivial parameters for evaluating a bivariate distribution as a 2D Gaussian. Inputs:

  - sx: standard deviation (not variance) of x var
  - sy: standard deviation (not variance) of y var

- cxy: covariance (not correlation coefficient) of x and y

Returns: (a, b, c), where z = k exp (ax^2 + bxy + cy^2)

The proper value for k can be obtained from bivnorm().

pwkit.ellipses.**databiv**(*xy*, *coordouter=False*, ***kwargs*)

Compute the main parameters of a bivariate distribution from data. The parameters are returned in the same format as used in the rest of this module.

- xy: a 2D data array of shape (2, nsamp) or (nsamp, 2)
- coordouter: if True, the coordinate axis is the outer axis; i.e. the shape is (2, nsamp). Otherwise, the coordinate axis is the inner axis; i.e. shape is (nsamp, 2).

Returns: (sx, sy, cxy)

In both cases, the first slice along the coordinate axis gives the X data (i.e., xy[0] or xy[:,0]) and the second slice gives the Y data (xy[1] or xy[:,1]).

pwkit.ellipses.**bivrandom**(*x0*, *y0*, *sx*, *sy*, *cxy*, *size=None*)

Compute random values distributed according to the specified bivariate distribution.

Inputs:

- x0: the center of the x distribution (i.e. its intended mean)
- y0: the center of the y distribution
- sx: standard deviation (not variance) of x var
- sy: standard deviation (not variance) of y var
- cxy: covariance (not correlation coefficient) of x and y
- size (optional): the number of values to compute

Returns: array of shape (size, 2); or just (2,), if size was not specified.

The bivariate parameters of the generated data are approximately recoverable by calling 'databiv(retval)'.

pwkit.ellipses.**ellpoint**(*mjr*, *mnr*, *pa*, *th*)

Compute a point on an ellipse parametrically. Inputs:

- mjr: major axis (sigma not FWHM) of the ellipse
- mnr: minor axis (sigma not FWHM) of the ellipse
- pa: position angle (from +x to +y) of the ellipse, radians
- th: the parameter, 0 <= th < 2pi: the eccentric anomaly

Returns: (x, y)

th may be a vector, in which case x and y will be as well.

pwkit.ellipses.**elld2**(*x0*, *y0*, *mjr*, *mnr*, *pa*, *x*, *y*)

Given an 2D Gaussian expressed as an ellipse (major, minor, pa), compute a "squared distance parameter" such that

z = exp (-0.5 * d2)

Inputs:

- x0: position of Gaussian center on x axis
- y0: position of Gaussian center on y axis

- mjr: major axis (sigma not FWHM) of the Gaussian
- mnr: minor axis (sigma not FWHM) of the Gaussian
- pa: position angle (from +x to +y) of the Gaussian, radians
- x: x coordinates of the locations for which to evaluate d2
- y: y coordinates of the locations for which to evaluate d2

Returns: d2, distance parameter defined as above.

x0, y0, mjr, and mnr may be in any units so long as they're consistent. x and y may be arrays (of the same shape), in which case d2 will be an array as well.

pwkit.ellipses.**ellbiv**(*mjr*, *mnr*, *pa*)

Given a 2D Gaussian expressed as an ellipse (major, minor, pa), compute the equivalent parameters for a Gaussian bivariate distribution. We assume that the ellipse is normalized so that the functions evaluate identicall for major = minor.

Inputs:

- mjr: major axis (sigma not FWHM) of the Gaussian
- mnr: minor axis (sigma not FWHM) of the Gaussian
- pa: position angle (from +x to +y) of the Gaussian, radians

Returns:

- sx: standard deviation (not variance) of x var
- sy: standard deviation (not variance) of y var
- cxy: covariance (not correlation coefficient) of x and y

pwkit.ellipses.**ellabc**(*mjr*, *mnr*, *pa*)

Given a 2D Gaussian expressed as an ellipse (major, minor, pa), compute the nontrivial parameters for its evaluation.

- mjr: major axis (sigma not FWHM) of the Gaussian
- mnr: minor axis (sigma not FWHM) of the Gaussian
- pa: position angle (from +x to +y) of the Gaussian, radians

Returns: (a, b, c), where z = exp (ax^2 + bxy + cy^2)

pwkit.ellipses.**ellplot**(*mjr*, *mnr*, *pa*)

Utility for debugging.

pwkit.ellipses.**abcell**(*a*, *b*, *c*)

Given the nontrivial parameters for evaluation a 2D Gaussian as a polynomial, compute the equivalent ellipse parameters (major, minor, pa)

Inputs: (a, b, c), where z = exp (ax^2 + bxy + cy^2)

Returns:

- mjr: major axis (sigma not FWHM) of the Gaussian
- mnr: minor axis (sigma not FWHM) of the Gaussian
- pa: position angle (from +x to +y) of the Gaussian, radians

pwkit.ellipses.**abcd2**(*x0*, *y0*, *a*, *b*, *c*, *x*, *y*)

Given an 2D Gaussian expressed as the ABC polynomial coefficients, compute a "squared distance parameter" such that

z = exp (-0.5 * d2)

Inputs:

- x0: position of Gaussian center on x axis
- y0: position of Gaussian center on y axis
- a: such that z = exp (ax^2 + bxy + cy^2)
- b: see above
- c: see above
- x: x coordinates of the locations for which to evaluate d2
- y: y coordinates of the locations for which to evaluate d2

Returns: d2, distance parameter defined as above.

This is pretty trivial.

3.7 Run the Fleischman & Kuznetsov (2010) synchrotron code (`pwkit.fk10`)

This module helps you run the synchrotron code described in Fleischman & Kuznetsov (2010; hereafter FK10) [DOI:10.1088/0004-637X/721/2/1127]. The code is provided as a precompiled binary module. It's meant to be called from IDL, but we won't let that stop us!

The main interface to the code is the `Calculator` class. But before you can use it, you must install the code, as described below.

3.7.1 Installing the code

To do anything useful with this module, you must first obtain the precompiled module. This isn't the sort of module you'd want to install into your system shared libraries, so for applications you'll probably be storing it in some random directory. Therefore, all actions in this module start by specifying the path to the library.

The module can be downloaded from as part of a Supplementary Data archive attached to the journal paper. At the moment, the direct link is here, but that might change over time. The journal's website for the paper should always have a link.

The archive contains compiled versions of the code for Windows, 32-bit Linux, and 64-bit Linux. It is quite worrisome that maybe one day these files will stop working, but that's what we've got right now.

Anyway, you should download and unpack the archive and copy the desired file to wherever makes the most sense for your software environment and application. On 64-bit Linux, the file name is `libGS_Std_HomSrc_CEH.so.64`. Any variable named *shlib_path* that comes up in the API should be a path to this file. Note that relative paths should include a directory component (e.g. `./libGS_Std_HomSrc_CEH.so.64`); the `ctypes` module treats bare filenames specially.

class pwkit.fk10.**Calculator**(*shlib_path*)

An interface to the FK10 synchrotron routines.

This class maintains state about the input parameters that can be passed to the routines, and can invoke them for you.

Constructor arguments

shlib_path The path to the compiled FK10 code, as described in the module-level documentation.

Newly-constructed objects are initialized with:

```
self.set_hybrid_parameters(12, 12)
self.set_ignore_q_terms(False)
self.set_trapezoidal_integration(15)
```

Setting parameters

set_bfield(B_G)	Set the strength of the local magnetic field.
set_bfield_for_s0(s0)	Set B to probe a certain harmonic number.
set_edist_powerlaw(emin_mev, emax_mev, ...)	Set the energy distribution function to a power law.
set_edist_powerlaw_gamma(gmin, gmax, delta, ...)	Set the energy distribution function to a power law in the Lorentz factor
set_freqs(n, f_lo_ghz, f_hi_ghz)	Set the frequency grid on which to perform the calculations.
set_hybrid_parameters(s_C, s_WH[, do_renorm])	Set the hybrid/renormalization control parameters.
set_ignore_q_terms(ignore_q_terms)	Set whether "Q" terms are ignored.
set_obs_angle(theta_rad)	Set the observer angle relative to the field.
set_one_freq(f_ghz)	Set the code to calculate results at just one frequency.
set_padist_gaussian_loss_cone(boundary_sd, ...)	Set the pitch-angle distribution to a Gaussian loss cone.
set_padist_isotropic()	Set the pitch-angle distribution to be isotropic.
set_thermal_background(T_K, nth_cc)	Set the properties of the background thermal plasma.
set_trapezoidal_integration(n)	Set the code to use trapezoidal integration.

Running calculations

find_rt_coefficients([depth0])	Figure out emission and absorption coefficients for the current parameters.
find_rt_coefficients_tot_intens([depth0])	Figure out total-intensity emission and absorption coefficients for the current parameters.

set_bfield(*B_G*)

Set the strength of the local magnetic field.

Call signature

B_G The magnetic field strength, in Gauss

Returns *self* for convenience in chaining.

set_bfield_for_s0(*s0*)

Set B to probe a certain harmonic number.

Call signature

s0 The harmonic number to probe at the lowest frequency

Returns *self* for convenience in chaining.

This just proceeds from the relation `nu = s nu_c = s e B / 2 pi m_e c`. Since *s* and *nu* scale with each other, if multiple frequencies are being probed, the harmonic numbers being probed will scale

in the same way.

set_edist_powerlaw(*emin_mev*, *emax_mev*, *delta*, *ne_cc*)
Set the energy distribution function to a power law.

Call signature

emin_mev The minimum energy of the distribution, in MeV

emax_mev The maximum energy of the distribution, in MeV

delta The power-law index of the distribution

ne_cc The number density of energetic electrons, in cm^-3.

Returns *self* for convenience in chaining.

set_edist_powerlaw_gamma(*gmin*, *gmax*, *delta*, *ne_cc*)
Set the energy distribution function to a power law in the Lorentz factor

Call signature

gmin The minimum Lorentz factor of the distribution

gmax The maximum Lorentz factor of the distribution

delta The power-law index of the distribution

ne_cc The number density of energetic electrons, in cm^-3.

Returns *self* for convenience in chaining.

set_freqs(*n*, *f_lo_ghz*, *f_hi_ghz*)
Set the frequency grid on which to perform the calculations.

Call signature

n The number of frequency points to sample.

f_lo_ghz The lowest frequency to sample, in GHz.

f_hi_ghz The highest frequency to sample, in GHz.

Returns *self* for convenience in chaining.

set_hybrid_parameters(*s_C*, *s_WH*, *do_renorm=True*)
Set the hybrid/renormalization control parameters.

Call signature

s_C The harmonic number above which the continuous approximation is used (with special behavior; see below).

s_WH The harmonic number above which the Wild-Hill BEssel function approximations are used.

do_renorm **(default True)** Whether to do any renormalization at all.

Returns *self* for convenience in chaining.

FK10 uses frequency parameters f^C_cr and f^WH_cr to control some of its optimizations. This function sets these parameters as multiples of the electron cyclotron frequency (f_Be in FK10 notation): e.g., `f^C_cr = s_C * f_Be`.

At frequencies above f^C_cr, the "continuum" approximation is introduced, replacing the "exact" sum with an integral. At frequencies above f^WH_cr, the Wild-Hild approximations to the Bessel functions are used. In both cases, the activation of the optimizations can result in normalization shifts in the calculations. "Renormalization" computes these shifts (by doing both kinds of calculations at the transition frequencies)

and attempts to correct them. (Some of the FK10 documentation seems to refer to renormalization as "R-optimization".)

If f^C_cr is below the lowest frequency integrated, all calculations will be done in continuum mode. In this case, the sign of *s_C* sets whether Wild-Hill renormalization is applied. If *s_C* is negative and f^WH_cr is above the lowest frequency integration, renormalization is done. Otherwise, it is not.

The documentation regarding f^WH_cr is confusing. It states that f^WH_cr only matters if (1) s_WH < s_C or (2) s_C < 0 and f^WH_cr > f_0. It is not obvious to me why s_WH > s_C should only matter if s_C < 0, but that's what's implied.

In most examples in FK10, both of these parameters are set to 12.

set_ignore_q_terms(*ignore_q_terms*)
Set whether "Q" terms are ignored.

Call signature

ignore_q_terms If true, ignore "Q" terms in the integrations.

Returns *self* for convenience in chaining.

See Section 4.3 of FK10 and `OnlineII.pdf` in the Supplementary Data for a precise explanation. The default is to *not* ignore the terms.

set_obs_angle(*theta_rad*)
Set the observer angle relative to the field.

Call signature

theta_rad The angle between the ray path and the local magnetic field, in radians.

Returns *self* for convenience in chaining.

set_one_freq(*f_ghz*)
Set the code to calculate results at just one frequency.

Call signature

f_ghz The frequency to sample, in GHz.

Returns *self* for convenience in chaining.

set_padist_gaussian_loss_cone(*boundary_rad*, *expwidth*)
Set the pitch-angle distribution to a Gaussian loss cone.

Call signature

boundary_rad The angle inside which there are no losses, in radians.

expwidth The characteristic width of the Gaussian loss profile *in direction-cosine units*.

Returns *self* for convenience in chaining.

See `OnlineI.pdf` in the Supplementary Data for a precise definition. (And note the distinction between α_c and μ_c since not everything is direction cosines.)

set_padist_isotropic()
Set the pitch-angle distribution to be isotropic.

Returns *self* for convenience in chaining.

set_thermal_background(*T_K*, *nth_cc*)
Set the properties of the background thermal plasma.

Call signature

T_K The temperature of the background plasma, in Kelvin.

nth_cc The number density of thermal electrons, in cm^-3.

Returns *self* for convenience in chaining.

Note that the parameters set here are the same as the ones that describe the thermal electron distribution, if you choose one of the electron energy distributions that explicitly models a thermal component ("thm", "tnt", "tnp", "tng", "kappa" in the code's terminology). For the power-law-y electron distributions, these parameters are used to calculate dispersion parameters (e.g. refractive indices) and a free-free contribution, but their synchrotron contribution is ignored.

set_trapezoidal_integration(*n*)
Set the code to use trapezoidal integration.

Call signature

n Use this many nodes

Returns *self* for convenience in chaining.

find_rt_coefficients(*depth0=None*)
Figure out emission and absorption coefficients for the current parameters.

Argument

depth0 **(default None)** A first guess to use for a good integration depth, in cm. If None, the most recent value is used.

Return value

A tuple `(j_O, alpha_O, j_X, alpha_X)`, where:

j_O The O-mode emission coefficient, in erg/s/cm^3/Hz/sr.

alpha_O The O-mode absorption coefficient, in cm^-1.

j_X The X-mode emission coefficient, in erg/s/cm^3/Hz/sr.

alpha_X The X-mode absorption coefficient, in cm^-1.

The main outputs of the FK10 code are intensities and "damping factors" describing a radiative transfer integration of the emission from a homogeneous source. But there are times when we'd rather just know what the actual emission and absorption coefficients are. These can be backed out from the FK10 outputs, but only if the "damping factor" takes on an intermediate value not extremely close to either 0 or 1. Unfortunately, there's no way for us to know a priori what choice of the "depth" parameter will yield a nice value for the damping factor. This routine automatically figures one out, by repeatedly running the calculation.

To keep things simple, this routine requires that you only be solving for coefficients for one frequency at a time (e.g., *set_one_freq()*).

find_rt_coefficients_tot_intens(*depth0=None*)
Figure out total-intensity emission and absorption coefficients for the current parameters.

Argument

depth0 **(default None)** A first guess to use for a good integration depth, in cm. If None, the most recent value is used.

Return value

A tuple `(j_I, alpha_I)`, where:

j_I The total intensity emission coefficient, in erg/s/cm^3/Hz/sr.

alpha_I The total intensity absorption coefficient, in cm^-1.

See *find_rt_coefficients()* for an explanation how this routine works. This version merely postprocesses the results from that method to convert the coefficients to refer to total intensity.

3.8 Modeling sources in images (pwkit.immodel)

pwkit.immodel - Analytical modeling of astronomical images.

This is derived from copl/pylib/bgfit.py and copl/bin/imsrcdebug. I keep on wanting this code so I should put it somewhere more generic. Such as here. Also, given the history, there are a lot more bells and whistles in the code than the currently exposed UI really needs.

3.9 Bayesian confidence intervals for count rates (pwkit.kbn_conf)

pwkit.kbn_conf - calculate Poisson-like confidence intervals assuming a background

This module implements the Bayesian confidence intervals for Poisson processes in a background using the approach described in Kraft, Burrows, & Nousek (1991). That paper provides tables of values; this module can calculate intervals for arbitrary inputs. Requires *scipy*.

This implementation almost directly transcribes the equations. We do, however, work in log-gamma space to try to avoid overflows with large values of N or B.

Functions:

kbn_conf - Compute a single confidence limit. vec_kbn_conf - Vectorized version of *kbn_conf*.

TODO: tests!

pwkit.kbn_conf.**kbn_conf**(*N*, *B*, *CL*)

Given a (integer) number of observed Poisson events *N* and a (real) expected number of background events *B* and a confidence limit *CL* (between 0 and 1), return the confidence interval on the source event rate.

Returns: (Smin, Smax)

This interval is calculated using the Bayesian formalism of Kraft, Burrows, & Nousek (1991), which assumes no uncertainty in *B* and returns the smallest possible interval that satisfies the above properties.

Example: in a certain time interval, 3 events were recorded. Based on external knowledge, it is expected that on average 0.5 background events will be recorded in the same interval. The 95% confidence interval on the source event rate is

```
>>> kbn_conf.kbn_conf (3, 0.5, 0.95)
<<< (0.22156, 7.40188)
```

which agrees with the entry in Table 2 of KBN91.

Reference info: 1991ApJ...374..344K, doi:10.1086/170124

3.10 Nonlinear least-squares minimization with Levenberg-Marquardt (pwkit.lmmin)

pwkit.lmmin - Pythonic, Numpy-based Levenberg-Marquardt least-squares minimizer

Basic usage:

```
from pwkit.lmmin import Problem, ResidualProblem

def yfunc(params, vals):
    vals[:] = {stuff with params}
def jfunc(params, jac):
    jac[i,j] = {deriv of val[j] w.r.t. params[i]}
    # i.e. jac[i] = {deriv of val wrt params[i]}

p = Problem(npar, nout, yfunc, jfunc=None)
solution = p.solve(guess)

p2 = Problem()
p2.set_npar(npar) # enables configuration of parameter meta-info
p2.set_func(nout, yfunc, jfunc)
```

Main Solution properties:

> prob - The Problem. status - Set of strings; presence of 'ftol', 'gtol', or 'xtol' suggests success. params - Final parameter values. perror - 1σ uncertainties on params. covar - Covariance matrix of parameters. fnorm - Final norm of function output. fvec - Final vector of function outputs. fjac - Final Jacobian matrix of d(fvec)/d(params).

Automatic least-squares model-fitting (subtracts "observed" Y values and multiplies by inverse errors):

> **def yrfunc(params, modelyvalues):** modelyvalues[:] = {stuff with params}
>
> **def yjfunc(params, modelyjac):** jac[i,j] = {deriv of modelyvalue[j] w.r.t. params[i]}
>
> p.set_residual_func(yobs, errinv, yrfunc, jrfunc, reckless=False) p = ResidualProblem(npar, yobs, errinv, yrfunc, jrfunc=None, reckless=False)

Parameter meta-information:

> p.p_value(paramindex, value, fixed=False) p.p_limit(paramindex, lower=-inf, upper=+inf) p.p_step(paramindex, stepsize, maxstep=info, isrel=False) p.p_side(paramindex, sidedness) # one of 'auto', 'pos', 'neg', 'two' p.p_tie(paramindex, tiefunc) # pval = tiefunc(params)

solve() status codes:

Solution.status is a set of strings. The presence of a string in the set means that the specified condition was active when the iteration terminated. Multiple conditions may contribute to ending the iteration. The algorithm likely did not converge correctly if none of 'ftol', 'xtol', or 'gtol' are in status upon termination.

'ftol' (MINPACK/MPFIT equiv: 1, 3) "Termination occurs when both the actual and predicted relative reductions in the sum of squares are at most FTOL. Therefore, FTOL measures the relative error desired in the sum of squares."

'xtol' (MINPACK/MPFIT equiv: 2, 3) "Termination occurs when the relative error between two consecutive iterates is at most XTOL. Therefore, XTOL measures the relative error desired in the approximate solution."

'gtol' (MINPACK/MPFIT equiv: 4) "Termination occurs when the cosine of the angle between fvec and any column of the jacobian is at most GTOL in absolute value. Therefore, GTOL measures the orthogonality desired between the function vector and the columns of the jacobian."

'maxiter' (MINPACK/MPFIT equiv: 5) Number of iterations exceeds maxiter.

'feps' (MINPACK/MPFIT equiv: 6) "ftol is too small. no further reduction in the sum of squares is possible."

'xeps' (MINPACK/MPFIT equiv: 7) "xtol is too small. no further improvement in the approximate solution x is possible."

'geps' (MINPACK/MPFIT equiv: 8) "gtol is too small. fvec is orthogonal to the columns of the jacobian to machine precision."

(This docstring contains only usage information. For important information regarding provenance, license, and academic references, see comments in the module source code.)

class pwkit.lmmin.**Problem**(*npar=None*, *nout=None*, *yfunc=None*, *jfunc=None*, *solclass=<class 'pwkit.lmmin.Solution'>*)

A Levenberg-Marquardt problem to be solved. Attributes:

damp Tanh damping factor of extreme function values.

debug_calls If true, information about function calls is printed.

debug_jac If true, information about jacobian calls is printed.

diag Scale factors for parameter derivatives, used to condition the problem.

epsilon The floating-point epsilon value, used to determine step sizes in automatic Jacobian computation.

factor The step bound is *factor* times the initial value times *diag*.

ftol The relative error desired in the sum of squares.

gtol The orthogonality desired between the function vector and the columns of the Jacobian.

maxiter The maximum number of iterations allowed.

normfunc A function to compute the norm of a vector.

solclass A factory for Solution instances.

xtol The relative error desired in the approximate solution.

Methods:

copy Duplicate this *Problem*.

get_ndof Get the number of degrees of freedom in the problem.

get_nfree Get the number of free parameters (fixed/tied/etc pars are not free).

p_value Set the initial or fixed value of a parameter.

p_limit Set limits on parameter values.

p_step Set the stepsize for a parameter.

p_side Set the sidedness with which auto-derivatives are computed for a par.

p_tie Set a parameter to be a function of other parameters.

set_func Set the function to be optimized.

set_npar Set the number of parameters; allows p_* to be called.

set_residual_func Set the function to a standard model-fitting style.

solve Run the algorithm.

solve_scipy Run the algorithm using the Scipy implementation (for testing).

p_side(*idx*, *sidedness*)

Acceptable values for *sidedness* are "auto", "pos", "neg", and "two".

class pwkit.lmmin.**Solution**(*prob*)

A parameter solution from the Levenberg-Marquard algorithm. Attributes:

ndof - The number of degrees of freedom in the problem. prob - The *Problem*. status - A set of strings indicating which stop condition(s) arose. niter - The number of iterations needed to obtain the solution. perror - The 1σ

errors on the final parameters. params - The final best-fit parameters. covar - The covariance of the function parameters. fnorm - The final function norm. fvec - The final function outputs. fjac - The final Jacobian. nfev - The number of function evaluations needed to obtain the solution. njev - The number of Jacobian evaluations needed to obtain the solution.

The presence of 'ftol', 'gtol', or 'xtol' in *status* suggests success.

3.11 Fitting generic models with least-squares minimization (`pwkit.lsqmdl`)

Model data with least-squares fitting

This module provides tools for fitting models to data using least-squares optimization.

There are four basic approaches all offering a common programming interface:

- *Generic Nonlinear Modeling*
- *One-dimensional Polynomial Modeling*
- *Modeling of a Single Scale Factor*
- *Modeling With Pluggable Components*

`ModelBase`(data[, invsigma])	An abstract base class holding data and a model for least-squares fitting.
`Parameter`(owner, index)	Information about a parameter in a least-squares model.

class pwkit.lsqmdl.**ModelBase**(*data*, *invsigma=None*)

An abstract base class holding data and a model for least-squares fitting.

The models implemented in this module all derive from this class and so inherit the attributes and methods described below.

A `Parameter` data structure may be obtained by indexing this object with either the parameter's numerical index or its name. I.e.:

```
m = Model(...).solve(...)
p = m['slope']
print(p.name, p.value, p.uncert, p.uval)
```

chisq = None

After fitting, the χ^2 of the fit.

covar = None

After fitting, the variance-covariance matrix representing the parameter uncertainties.

data = None

The data to be modeled; an n-dimensional Numpy array.

invsigma = None

Data weights: $1/\sigma$ for each data point.

make_frozen_func(*params*)

Return a data-generating model function frozen at the specified parameters.

As with the `mfunc` attribute, the resulting function may or may not take arguments depending on the particular kind of model being evaluated.

mdata = None
After fitting, the modeled data at the best parameters.

mfunc = None
After fitting, a callable function evaluating the model fixed at best params.

The resulting function may or may not take arguments depending on the particular kind of model being evaluated.

params = None
After fitting, a Numpy ndarray of solved model parameters.

plot(*modelx*, *dlines=False*, *xmin=None*, *xmax=None*, *ymin=None*, *ymax=None*, ***kwargs*)
Plot the data and model (requires *omega*).

This assumes that *data* is 1D and that *mfunc* takes one argument that should be treated as the X variable.

pnames = None
A list of textual names for the parameters.

print_soln()
Print information about the model solution.

puncerts = None
After fitting, a Numpy ndarray of 1σ uncertainties on the model parameters.

rchisq = None
After fitting, the reduced χ^2 of the fit, or None if there are no degrees of freedom.

resids = None
After fitting, the residuals: `resids = data - mdata`.

set_data(*data*, *invsigma=None*)
Set the data to be modeled.

Returns *self*.

show_corr()
Show the parameter correlation matrix with *pwkit.ndshow_gtk3*.

show_cov()
Show the parameter covariance matrix with *pwkit.ndshow_gtk3*.

class pwkit.lsqmdl.**Parameter**(*owner*, *index*)
Information about a parameter in a least-squares model.

These data may only be obtained after solving least-squares problem. These objects reference information from their parent objects, so changing the parent will alter the apparent contents of these objects.

index
The parameter's index in the Model's arrays.

name
The parameter's name.

uncert
The uncertainty in *value*.

uval
Accesses *value* and *uncert* as a *pwkit.msmt.Uval*.

value
The parameter's value.

3.11.1 Generic Nonlinear Modeling

Model(simple_func, data[, invsigma, args])	Models data with a generic nonlinear optimizer
Parameter(owner, index)	Information about a parameter in a least-squares model.

class pwkit.lsqmdl.**Model**(*simple_func*, *data*, *invsigma=None*, *args=()*)

Models data with a generic nonlinear optimizer

Basic usage is:

```
def func(p1, p2, x):
    simulated_data = p1 * x + p2
    return simulated_data

x = [1, 2, 3]
data = [10, 14, 15.8]
mdl = Model(func, data, args=(x,)).solve(guess).print_soln()
```

The *Model* constructor can take an optional argument `invsigma` after `data`; it specifies *inverse sigmas*, **not** inverse *variances* (the usual statistical weights), for the data points. Since most applications deal in sigmas, take care to write:

```
m = Model(func, data, 1. / uncerts) # right!
```

not:

```
m = Model(func, data, uncerts) # WRONG
```

If you have zero uncertainty on a measurement, you must wind a way to express that constraint without including that measurement as part of the `data` vector.

lm_prob = None

A *pwkit.lmmin.Problem* instance describing the problem to be solved.

After setting up the data-generating function, you can access this item to tune the solver.

make_frozen_func(*params*)

Returns a model function frozen to the specified parameter values.

Any remaining arguments are left free and must be provided when the function is called.

For this model, the returned function is the application of `functools.partial()` to the `func` property of this object.

set_func(*func*, *pnames*, *args=()*)

Set the model function to use an efficient but tedious calling convention.

The function should obey the following convention:

```
def func(param_vec, *args):
    modeled_data = { do something using param_vec }
    return modeled_data
```

This function creates the *pwkit.lmmin.Problem* so that the caller can futz with it before calling *solve()*, if so desired.

Returns *self*.

set_simple_func(*func*, *args=()*)
Set the model function to use a simple but somewhat inefficient calling convention.

The function should obey the following convention:

```
def func(param0, param1, ..., paramN, *args):
    modeled_data = { do something using the parameters }
    return modeled_data
```

Returns *self*.

solve(*guess*)
Solve for the parameters, using an initial guess.

This uses the Levenberg-Marquardt optimizer described in *pwkit.lmmin*.

Returns *self*.

3.11.2 One-dimensional Polynomial Modeling

class pwkit.lsqmdl.**PolynomialModel**(*maxexponent*, *x*, *data*, *invsigma=None*)
Least-squares polynomial fit.

Because this is a very specialized kind of problem, we don't need an initial guess to solve, and we can use fast built-in numerical routines.

The output parameters are named "a0", "a1", ... and are stored in that order in PolynomialModel.params[]. We have `y = sum(x**i * a[i])`, so "a2" = "params[2]" is the quadratic term, etc.

This model does *not* give uncertainties on the derived coefficients. The as_nonlinear() method can be use to get a *Model* instance with uncertainties.

Methods:

as_nonlinear - Return a (lmmin-based) *Model* equivalent to self.

as_nonlinear(*params=None*)
Return a *Model* equivalent to this object. The nonlinear solver is less efficient, but lets you freeze parameters, compute uncertainties, etc.

If the *params* argument is provided, solve() will be called on the returned object with those parameters. If it is *None* and this object has parameters in *self.params*, those will be use. Otherwise, solve() will not be called on the returned object.

make_frozen_func(*params*)
Return a data-generating model function frozen at the specified parameters.

As with the `mfunc` attribute, the resulting function may or may not take arguments depending on the particular kind of model being evaluated.

3.11.3 Modeling of a Single Scale Factor

class pwkit.lsqmdl.**ScaleModel**(*x*, *data*, *invsigma=None*)
Solve *data* = *m* * *x* for *m*.

make_frozen_func(*params*)
Return a data-generating model function frozen at the specified parameters.

As with the `mfunc` attribute, the resulting function may or may not take arguments depending on the particular kind of model being evaluated.

3.11.4 Modeling With Pluggable Components

ComposedModel(component, data[, invsigma])	
ModelComponent([name])	
AddConstantComponent([name])	
AddValuesComponent(nvals[, name])	XXX terminology between this and AddConstant is mushy.
AddPolynomialComponent(maxexponent, x[, name])	
SeriesComponent([components, name])	Apply a set of subcomponents in series, isolating each from the other.
MatMultComponent(k[, name])	Given a component yielding k**2 data points and k additional components, each yielding n data points.
ScaleComponent([subcomp, name])	

class pwkit.lsqmdl.**ComposedModel**(*component*, *data*, *invsigma=None*)

debug_derivative(*guess*)
returns (explicit, auto)

make_frozen_func()
Return a data-generating model function frozen at the specified parameters.

As with the `mfunc` attribute, the resulting function may or may not take arguments depending on the particular kind of model being evaluated.

class pwkit.lsqmdl.**ModelComponent**(*name=None*)

deriv(*pars*, *jac*)
Compute the Jacobian. *jac[i]* is d‘mdata‘/d‘pars[i]‘.

extract(*pars*, *perr*, *cov*)
Extract fit results into the object for ease of inspection.

finalize_setup()
If the component has subcomponents, this should set their *name*, *setguess*, *setvalue*, and *setlimit* properties. It should also set *npar* (on self) to the final value.

model(*pars*, *mdata*)
Modify *mdata* based on *pars*.

prep_params()
This should make any necessary calls to *setvalue* or *setlimit*, though in straightforward cases it should just be up to the user to do this. If the component has subcomponents, their *prep_params* functions should be called.

class pwkit.lsqmdl.**AddConstantComponent**(*name=None*)

deriv(*pars*, *jac*)
Compute the Jacobian. *jac[i]* is d‘mdata‘/d‘pars[i]‘.

extract(*pars*, *perr*, *cov*)
Extract fit results into the object for ease of inspection.

model(*pars*, *mdata*)
Modify *mdata* based on *pars*.

class pwkit.lsqmdl.**AddValuesComponent**(*nvals*, *name=None*)
XXX terminology between this and AddConstant is mushy.

deriv(*pars*, *jac*)
Compute the Jacobian. *jac[i]* is d‘mdata‘/d‘pars[i]‘.

extract(*pars*, *perr*, *cov*)
Extract fit results into the object for ease of inspection.

model(*pars*, *mdata*)
Modify *mdata* based on *pars*.

class pwkit.lsqmdl.**AddPolynomialComponent**(*maxexponent*, *x*, *name=None*)

deriv(*pars*, *jac*)
Compute the Jacobian. *jac[i]* is d‘mdata‘/d‘pars[i]‘.

extract(*pars*, *perr*, *cov*)
Extract fit results into the object for ease of inspection.

model(*pars*, *mdata*)
Modify *mdata* based on *pars*.

class pwkit.lsqmdl.**SeriesComponent**(*components=()*, *name=None*)
Apply a set of subcomponents in series, isolating each from the other. This is only valid if every subcomponent except the first is additive – otherwise, the Jacobian won't be right.

add(*component*)
This helps, but direct manipulation of self.components should be supported.

deriv(*pars*, *jac*)
Compute the Jacobian. *jac[i]* is d‘mdata‘/d‘pars[i]‘.

extract(*pars*, *perr*, *cov*)
Extract fit results into the object for ease of inspection.

finalize_setup()
If the component has subcomponents, this should set their *name*, *setguess*, *setvalue*, and *setlimit* properties. It should also set *npar* (on self) to the final value.

model(*pars*, *mdata*)
Modify *mdata* based on *pars*.

prep_params()
This should make any necessary calls to *setvalue* or *setlimit*, though in straightforward cases it should just be up to the user to do this. If the component has subcomponents, their *prep_params* functions should be called.

class pwkit.lsqmdl.**MatMultComponent**(*k*, *name=None*)
Given a component yielding k**2 data points and k additional components, each yielding n data points. The result is [A]×[B], where A is the square matrix formed from the first component's output, and B is the (k, n) matrix of stacked output from the final k components.

Parameters are ordered in same way as the components named above.

deriv(*pars*, *jac*)
Compute the Jacobian. *jac[i]* is d‘mdata‘/d‘pars[i]‘.

extract(*pars*, *perr*, *cov*)
Extract fit results into the object for ease of inspection.

finalize_setup()
If the component has subcomponents, this should set their *name*, *setguess*, *setvalue*, and *setlimit* properties. It should also set *npar* (on self) to the final value.

model(*pars*, *mdata*)
Modify *mdata* based on *pars*.

prep_params()
This should make any necessary calls to *setvalue* or *setlimit*, though in straightforward cases it should just be up to the user to do this. If the component has subcomponents, their *prep_params* functions should be called.

class pwkit.lsqmdl.**ScaleComponent**(*subcomp=None*, *name=None*)

deriv(*pars*, *jac*)
Compute the Jacobian. *jac[i]* is d‘mdata‘/d‘pars[i]‘.

extract(*pars*, *perr*, *cov*)
Extract fit results into the object for ease of inspection.

finalize_setup()
If the component has subcomponents, this should set their *name*, *setguess*, *setvalue*, and *setlimit* properties. It should also set *npar* (on self) to the final value.

model(*pars*, *mdata*)
Modify *mdata* based on *pars*.

prep_params()
This should make any necessary calls to *setvalue* or *setlimit*, though in straightforward cases it should just be up to the user to do this. If the component has subcomponents, their *prep_params* functions should be called.

3.12 Math with uncertain and censored measurements (pwkit.msmt)

pwkit.msmt - Working with uncertain measurements.

Classes:

Uval - An empirical uncertain value represented by numerical samples. LimitError - Raised on illegal operations on upper/lower limits. Lval - Container for either precise values or upper/lower limits. Textual - A measurement recorded in textual form.

Generic unary functions on measurements:

absolute - abs(x) arccos - As named. arcsin - As named. arctan - As named. cos - As named. errinfo - Get (limtype, repval, plus_1_sigma, minus_1_sigma) expm1 - exp(x) - 1 exp - As named. fmtinfo - Get (typetag, text, is_imprecise) for textual round-tripping. isfinite - True if the value is well-defined and finite. liminfo - Get (limtype, repval) limtype - -1 if the datum is an upper limit; 1 if lower; 0 otherwise. log10 - As named. log1p - log(1+x) log2 - As named. log - As named. negative - -x reciprocal - 1/x repval - Get a “representative” value if x (in case it is uncertain). sin - As named. sqrt - As named. square - x**2 tan - As named. unwrap - Get a version of x on which algebra can be performed.

Generic binary mathematical-ish functions:

add - x + y divide - x / y, never with floor-integer division floor_divide- x // y multiply - x * y power - x ** y subtract - x - y true_divide - x / y, never with floor-integer division typealign - Return (x*, y*) cast to same algebra-friendly type: float, Uval, or Lval.

Miscellaneous functions:

is_measurement - Check whether an object is numerical find_gamma_params - Compute reasonable Γ distribution parameters given mode/stddev. pk_scoreatpercentile - Simplified version of scipy.stats.scoreatpercentile. sample_double_norm - Sample from a quasi-normal distribution with asymmetric variances. sample_gamma - Sample from a Γ distribution with α/β parametrization.

Variables:

lval_unary_math - Dict of unary math functions operating on Lvals. parsers - Dict of type tag to parsing functions. scalar_unary_math - Dict of unary math functions operating on scalars. textual_unary_math - Dict of unary math functions operating on Textuals. UQUANT_UNCERT - Scale of uncertainty assumed for in cases where it's unquantified. uval_default_repval_method - Default method for computing Uval representative values. uval_dtype - The Numpy dtype of Uval data (often ignored!) uval_nsamples - Number of samples used when constructing Uvals uval_unary_math - Dict of unary math functions operating on Uvals.

exception pwkit.msmt.**LimitError**

class pwkit.msmt.**Lval**(*kind*, *value*)

A container for either precise values or upper/lower limits. Constructed as `Lval(kind, value)`, where *kind* is `"exact"`, `"uncertain"`, `"toinf"`, `"tozero"`, `"pastzero"`, or `"undef"`. Most easily constructed via `Textual.parse()`. Can also be constructed with `Lval.from_other()`.

Supported operations are `unicode() str() repr() -(neg) abs() + - * / ** += -= *= /= **=`.

class pwkit.msmt.**Textual**(*tkind*, *dkind*, *data*)

A measurement recorded in textual form.

Textual.from_exact(text, tkind='none') - *text* is passed to float() Textual.parse(text, tkind='none') - *text* as described below.

Transformation kinds are 'none', 'log10', or 'positive'. Expressions for values take the form '1.234', '<2', '>3', '~7', '6to8', '7pm0.1', or '12p1m0.3'.

Methods:

unparse() - Return parsed text (but not tkind!) unwrap() - Express as float/Uval/Lval as appropriate. repval(limitsok=False) - Get single scalar "representative" value. limtype() - -1 if upper limit; +1 if lower; 0 otherwise.

Supported operations: unicode() str() repr() [latexification] -(neg) abs() + - * / **

limtype()

Return -1 if this value is an upper limit, 1 if it is a lower limit, 0 otherwise.

repval(*limitsok=False*)

Get a best-effort representative value as a float. This can be DANGEROUS because it discards limit information, which is rarely wise.

class pwkit.msmt.**Uval**(*data*)

An empirical uncertain value, represented by samples.

Constructors are:

- `Uval.from_other()`
- `Uval.from_fixed()`
- `Uval.from_norm()`
- `Uval.from_unif()`
- `Uval.from_double_norm()`
- `Uval.from_gamma()`

- *Uval.from_pcount()*

Key methods are:

- *repvals()*
- *text_pieces()*
- `format()`
- `debug_distribution()`

Supported operations are: `unicode() str() repr() [latexification] + -(sub) * // / % ** += -= *= //= %= /= **= -(neg) ~ abs()`

static from_pcount(*nevents*)
We assume a Poisson process. nevents is the number of events in some interval. The distribution of values is the distribution of the Poisson rate parameter given this observed number of events, where the "rate" is in units of events per interval of the same duration. The max-likelihood value is nevents, but the mean value is nevents + 1. The gamma distribution is obtained by assuming an improper, uniform prior for the rate between 0 and infinity.

repvals(*method*)
Compute representative statistical values for this Uval. *method* may be either 'pct' or 'gauss'.

Returns (best, plus_one_sigma, minus_one_sigma), where *best* is the "best" value in some sense, and the others correspond to values at the ~84 and 16 percentile limits, respectively. Because of the sampled nature of the Uval system, there is no single method to compute these numbers.

The "pct" method returns the 50th, 15.866th, and 84.134th percentile values.

The "gauss" method computes the mean μ and standard deviation σ of the samples and returns [μ, μ+σ, μ-σ].

text_pieces(*method*, *uplaces=2*, *use_exponent=True*)
Return (main, dhigh, dlow, sharedexponent), all as strings. The delta terms do not have sign indicators. Any item except the first may be None.

method is passed to Uval.repvals() to compute representative statistical limits.

pwkit.msmt.**errinfo**(*msmt*)
Return (limtype, repval, errval1, errval2). Like m_liminfo, but also provides error bar information for values that have it.

pwkit.msmt.**fmtinfo**(*value*)
Returns (typetag, text, is_imprecise). Unlike other functions that operate on measurements, this also operates on bools, ints, and strings.

pwkit.msmt.**liminfo**(*msmt*)
Return (limtype, repval). *limtype* is -1 for upper limits, 1 for lower limits, and 0 otherwise; repval is a best-effort representative scalar value for this measurement.

pwkit.msmt.**limtype**(*msmt*)
Return -1 if this value is some kind of upper limit, 1 if this value is some kind of lower limit, 0 otherwise.

pwkit.msmt.**repval**(*msmt*, *limitsok=False*)
Get a best-effort representative value as a float. This is DANGEROUS because it discards limit information, which is rarely wise. m_liminfo() or m_unwrap() are recommended instead.

pwkit.msmt.**unwrap**(*msmt*)
Convert the value into the most basic representation that we can do math on: float if possible, then Uval, then Lval.

pwkit.msmt.**find_gamma_params**(*mode*, *std*)
Given a modal value and a standard deviation, compute corresponding parameters for the gamma distribution.

Intended to be used to replace normal distributions when the value must be positive and the uncertainty is comparable to the best value. Conversion equations determined from the relations given in the sample_gamma() docs.

pwkit.msmt.**sample_double_norm**(*mean*, *std_upper*, *std_lower*, *size*)
Note that this function requires Scipy.

pwkit.msmt.**sample_gamma**(*alpha*, *beta*, *size*)
This is mostly about recording the conversion between Numpy/Scipy conventions and Wikipedia conventions. Some equations:

mean = alpha / beta variance = alpha / beta**2 mode = (alpha - 1) / beta [if alpha > 1; otherwise undefined] skewness = 2 / sqrt(alpha)

pwkit.msmt.**UQUANT_UNCERT = 0.2**
Some values are known to be uncertain, but their uncertainties have not been quantified. This is lame but it happens. In this case, assume a 20% uncertainty.

We could infer uncertainties from the number of written digits: i.e., assuming "1.2" is uncertain by 0.05 or so, while "1.2000" is uncertain by 0.00005 or so. But there are many cases in astronomy where people just list values that are 20% uncertain and give them to multiple decimal places. I'd rather be conservative with these values than overly optimistic.

Code to do the appropriate parsing is in the Python uncertainties package, in its __init__.py:parse_error_in_parentheses().

pwkit.msmt.**uval_dtype**
alias of numpy.float64

3.13 Period-finding with Phase Dispersion Minimization (pwkit.pdm)

pwkit.pdm - period-finding with phase dispersion minimization

As defined in Stellingwerf (1978ApJ...224..953S). See the update in Schwarzenberg-Czerny (1997ApJ...489..941S), however, which corrects the significance test formally; Linnell Nemec & Nemec (1985AJ.....90.2317L) provide a Monte Carlo approach. Also, Stellingwerf has developed "PDM2" which attempts to improve a few aspects; see

- Stellingwerf's page
- The Wikipedia article

class pwkit.pdm.**PDMResult**(*thetas*, *imin*, *pmin*, *mc_tmins*, *mc_pvalue*, *mc_pmins*, *mc_puncert*)

imin
Alias for field number 1

mc_pmins
Alias for field number 5

mc_puncert
Alias for field number 6

mc_pvalue
Alias for field number 4

mc_tmins
Alias for field number 3

pmin
Alias for field number 2

thetas
Alias for field number 0

pwkit.pdm.**pdm**(*t*, *x*, *u*, *periods*, *nbin*, *nshift=8*, *nsmc=256*, *numc=256*, *weights=False*, *parallel=True*)
Perform phase dispersion minimization.

t [1D array] time coordinate

x [1D array, same size as *t*] observed value

u [1D array, same size as *t*] uncertainty on observed value; same units as *x*

periods [1D array] set of candidate periods to sample; same units as *t*

nbin [int] number of phase bins to construct

nshift [int=8] number of shifted binnings to sample to combact statistical flukes

nsmc [int=256] number of Monte Carlo shufflings to compute, to evaluate the significance of the minimal theta value.

numc [int=256] number of Monte Carlo added-noise datasets to compute, to evaluate the uncertainty in the location of the minimal theta value.

weights [bool=False] if True, 'u' is actually weights, not uncertainties. Usually weights = u**-2.

parallel [default True] Controls parallelization of the algorithm. Default uses all available cores. See *pwkit.parallel.make_parallel_helper*.

Returns named tuple of:

thetas [1D array] values of theta statistic, same size as *periods*

imin index of smallest (best) value in *thetas*

pmin the *period* value with the smallest (best) *theta*

mc_tmins 1D array of size *nsmc* with Monte Carlo samplings of minimal theta values for shufflings of the data; assesses significance of the peak

mc_pvalue probability (between 0 and 1) of obtaining the best theta value in a randomly-shuffled dataset

mc_pmins 1D array of size *numc* with Monte Carlo samplings of best period values for noise-added data; assesses uncertainty of *pmin*

mc_puncert standard deviation of *mc_pmins*; approximate uncertainty on *pmin*.

We don't do anything clever, so runtime scales at least as `t.size * periods.size * nbin * nshift * (nsmc + numc + 1)`.

3.14 Loading the outputs of PHOENIX atmospheric models (`pwkit.phoenix`)

pwkit.phoenix - Working with Phoenix atmospheric models.

Functions:

- load_spectrum - Load a model spectrum into a Pandas DataFrame.

Requires Pandas.

Individual data files for the BT-Settl models are about 120 MB, and there are a million variations, so we do not consider bundling them with pwkit. Therefore, we can safely expect that the model will be accessible as a path on the filesystem.

Current BT-Settl models may be downloaded from a SPECTRA directory within the BT-Settl download site (see the README). E.g.:

```
https://phoenix.ens-lyon.fr/Grids/BT-Settl/CIFIST2011bc/SPECTRA/
```

File names are generally:

```
lte{Teff/100}-{Logg}{[M/H]}a[alpha/H].GRIDNAME.spec.7.[gz|bz2|xz]
```

The first three columns are wavelength in Å, log10(F_λ), and log10(B_λ), where the latter is the blackbody flux for the given Teff. The fluxes can nominally be converted into absolute units with an offset of 8 in log space, but I doubt that can be trusted much. Subsequent columns are related to various spectral lines. See https://phoenix.ens-lyon.fr/Grids/FORMAT .

The files do not come sorted!

pwkit.phoenix.**load_spectrum**(*path*, *smoothing=181*, *DF=-8.0*)

Load a Phoenix model atmosphere spectrum.

path [string] The file path to load.

smoothing [integer] Smoothing to apply. If None, do not smooth. If an integer, smooth with a Hamming window. Otherwise, the variable is assumed to be a different smoothing window, and the data will be convolved with it.

DF: float Numerical factor used to compute the emergent flux density.

Returns a Pandas DataFrame containing the columns:

wlen Sample wavelength in Angstrom.

flam Flux density in erg/cm^2/s/Å. See *pwkit.synphot* for related tools.

The values of *flam* returned by this function are computed from the second column of the data file as specified in the documentation: `flam = 10**(col2 + DF)`. The documentation states that the default value, -8, is appropriate for most modern models; but some older models use other values.

Loading takes about 5 seconds on my current laptop. Un-smoothed spectra have about 630,000 samples.

3.15 Flux density models of radio calibrators (pwkit.radio_cal_models)

pwkit.radio_cal_models - models of radio calibrator flux densities.

From the command line:

```
python -m pwkit.radio_cal_models [-f] <source> <freq[mhz]>
python -m pwkit.radio_cal_models [-f] CasA     <freq[mhz]> <year>
```

Print the flux density of the specified calibrator at the specified frequency, in Janskys.

Arguments:

<source> the source name (e.g., 3c348)

<freq> the observing frequency in MHz (e.g., 1420)

<year> is the decimal year of the observation (e.g., 2007.8). Only needed if <source> is CasA.

-f activates "flux" mode, where a three-item string is printed that can be passed to MIRIAD tasks that accept a model flux and spectral index argument.

pwkit.radio_cal_models.**cas_a**(*freq_mhz*, *year*)
Return the flux of Cas A given a frequency and the year of observation. Based on the formula given in Baars et al., 1977.

Parameters:

freq - Observation frequency in MHz. year - Year of observation. May be floating-point.

Returns: s, flux in Jy.

pwkit.radio_cal_models.**init_cas_a**(*year*)
Insert an entry for Cas A into the table of models. Need to specify the year of the observations to account for the time variation of Cas A's emission.

3.16 Helpers for X-ray spectral modeling with the Sherpa packge (pwkit.sherpa)

This module contains helpers for modeling X-ray spectra with the Sherpa package.

This module includes a grab-bag of helpers in following broad topics:

- *Additional Spectral Models*
- *Tools for Plotting with Sherpa Data Objects*
- *Data Structure Utilities*

3.16.1 Additional Spectral Models

The *pwkit.sherpa* module provides several tools for constructing models not provided in the standard Sherpa distribution.

class pwkit.sherpa.**PowerLawApecDemModel**(*name*, *kt_array=None*)
A model with contributions from APEC plasmas at a range of temperatures, scaling with temperature.

Constructor arguments are:

name The Sherpa name of the resulting model instance.

kt_array **= None** An array of temperatures to use for the plasma models. If left at the default of None, a hard-coded default is used that spans temperatures of ~0.03 to 10 keV with logarithmic spacing.

The contribution at each temperature scales with kT as a power law. The model parameters are:

gfac The power-law normalization parameter. The contribution at temperature *kT* is `norm * kT**gfac`.

Abundanc The standard APEC abundance parameter.

redshift The standard APEC redshift parameter.

norm The standard overall normalization parameter.

This model is only efficient to compute if *Abundanc* and *redshift* are frozen.

pwkit.sherpa.**make_fixed_temp_multi_apec**(*kTs*, *name_template='apec%d'*, *norm=None*)
Create a model summing multiple APEC components at fixed temperatures.

kTs An iterable of temperatures for the components, in keV.

name_template = **'apec%d'** A template to use for the names of each component; it is string-formatted with the 0-based component number as an argument.

norm = **None** An initial normalization to be used for every component, or None to use the Sherpa default.

Returns: A tuple `(total_model, sub_models)`, where *total_model* is a Sherpa model representing the sum of the APEC components and *sub_models* is a list of the individual models.

This function creates a vector of APEC model components and sums them. Their *kT* parameters are set and then frozen (using `sherpa.astro.ui.freeze()`), so that upon exit from this function, the amplitude of each component is the only free parameter.

3.16.2 Tools for Plotting with Sherpa Data Objects

get_source_qq_data([id])	Get data for a quantile-quantile plot of the source data and model.
get_bkg_qq_data([id, bkg_id])	Get data for a quantile-quantile plot of the background data and model.
make_qq_plot(kev, obs, mdl, unit, key_text)	Make a quantile-quantile plot comparing events and a model.
make_multi_qq_plots(arrays, key_text)	Make a quantile-quantile plot comparing multiple sets of events and models.
make_spectrum_plot(model_plot, data_plot, desc)	Make a plot of a spectral model and data.
make_multi_spectrum_plots(model_plot, ..., [, ...])	Make a plot of multiple spectral models and data.

pwkit.sherpa.**get_source_qq_data**(*id=None*)
Get data for a quantile-quantile plot of the source data and model.

id The dataset id for which to get the data; defaults if unspecified.

Returns: An ndarray of shape `(3, npts)`. The first slice is the energy axis in keV; the second is the observed values in each bin (counts, or rate, or rate per keV, etc.); the third is the corresponding model value in each bin.

The inputs are implicit; the data are obtained from the current state of the Sherpa `ui` module.

pwkit.sherpa.**get_bkg_qq_data**(*id=None*, *bkg_id=None*)
Get data for a quantile-quantile plot of the background data and model.

id The dataset id for which to get the data; defaults if unspecified.

bkg_id The identifier of the background; defaults if unspecified.

Returns: An ndarray of shape `(3, npts)`. The first slice is the energy axis in keV; the second is the observed values in each bin (counts, or rate, or rate per keV, etc.); the third is the corresponding model value in each bin.

The inputs are implicit; the data are obtained from the current state of the Sherpa `ui` module.

pwkit.sherpa.**make_qq_plot**(*kev*, *obs*, *mdl*, *unit*, *key_text*)
Make a quantile-quantile plot comparing events and a model.

kev A 1D, sorted array of event energy bins measured in keV.

obs A 1D array giving the number or rate of events in each bin.

mdl A 1D array giving the modeled number or rate of events in each bin.

unit Text describing the unit in which *obs* and *mdl* are measured; will be shown on the plot axes.

key_text Text describing the quantile-quantile comparison quantity; will be shown on the plot legend.

Returns: An `omega.RectPlot` instance.

TODO: nothing about this is Sherpa-specific. Same goes for some of the plotting routines in `pkwit.environments.casa.data`; might be reasonable to add a submodule for generic X-ray-y plotting routines.

`pwkit.sherpa.`**`make_multi_qq_plots`**(*arrays*, *key_text*)

Make a quantile-quantile plot comparing multiple sets of events and models.

arrays

X.

key_text Text describing the quantile-quantile comparison quantity; will be shown on the plot legend.

Returns: An `omega.RectPlot` instance.

TODO: nothing about this is Sherpa-specific. Same goes for some of the plotting routines in `pkwit.environments.casa.data`; might be reasonable to add a submodule for generic X-ray-y plotting routines.

TODO: Some gross code duplication here.

`pwkit.sherpa.`**`make_spectrum_plot`**(*model_plot*, *data_plot*, *desc*, *xmin_clamp=0.01*, *min_valid_x=None*, *max_valid_x=None*)

Make a plot of a spectral model and data.

model_plot A model plot object returned by Sherpa from a call like *ui.get_model_plot()* or *ui.get_bkg_model_plot()*.

data_plot A data plot object returned by Sherpa from a call like *ui.get_source_plot()* or *ui.get_bkg_plot()*.

desc Text describing the origin of the data; will be shown in the plot legend (with "Model" and "Data" appended).

xmin_clamp The smallest "x" (energy axis) value that will be plotted; default is 0.01. This is needed to allow the plot to be shown on a logarithmic scale if the energy axes of the model go all the way to 0.

min_valid_x Either None, or the smallest "x" (energy axis) value in which the model and data are valid; this could correspond to a range specified in the "notice" command during analysis. If specified, a gray band will be added to the plot showing the invalidated regions.

max_valid_x Like *min_valid_x* but for the largest "x" (energy axis) value in which the model and data are valid.

Returns: A tuple `(plot, xlow, xhigh)`, where *plot* an OmegaPlot RectPlot instance, *xlow* is the left edge of the plot bounds, and *xhigh* is the right edge of the plot bounds.

`pwkit.sherpa.`**`make_multi_spectrum_plots`**(*model_plot*, *plotids*, *data_getter*, *desc*, *xmin_clamp=0.01*, *min_valid_x=None*, *max_valid_x=None*)

Make a plot of multiple spectral models and data.

model_plot A model plot object returned by Sherpa from a call like `ui.get_model_plot()` or `ui.get_bkg_model_plot()`.

data_plots An iterable of data plot objects returned by Sherpa from calls like `ui.get_source_plot(id)` or `ui.get_bkg_plot(id)`.

desc Text describing the origin of the data; will be shown in the plot legend (with "Model" and "Data #<number>" appended).

xmin_clamp The smallest "x" (energy axis) value that will be plotted; default is 0.01. This is needed to allow the plot to be shown on a logarithmic scale if the energy axes of the model go all the way to 0.

min_valid_x Either None, or the smallest "x" (energy axis) value in which the model and data are valid; this could correspond to a range specified in the "notice" command during analysis. If specified, a gray band will be added to the plot showing the invalidated regions.

max_valid_x Like *min_valid_x* but for the largest "x" (energy axis) value in which the model and data are valid.

Returns: A tuple `(plot, xlow, xhigh)`, where *plot* an OmegaPlot RectPlot instance, *xlow* is the left edge of the plot bounds, and *xhigh* is the right edge of the plot bounds.

TODO: not happy about the code duplication with *make_spectrum_plot()* but here we are.

3.16.3 Data Structure Utilities

expand_rmf_matrix(rmf)	Expand an RMF matrix stored in compressed form.
derive_identity_arf(name, arf)	Create an "identity" ARF that has uniform sensitivity.
derive_identity_rmf(name, rmf)	Create an "identity" RMF that does not mix energies.

pwkit.sherpa.**expand_rmf_matrix**(*rmf*)
Expand an RMF matrix stored in compressed form.

rmf An RMF object as might be returned by `sherpa.astro.ui.get_rmf()`.

Returns: A non-sparse RMF matrix.

The Response Matrix Function (RMF) of an X-ray telescope like Chandra can be stored in a sparse format as defined in OGIP Calibration Memo CAL/GEN/92-002. For visualization and analysis purposes, it can be useful to de-sparsify the matrices stored in this way. This function does that, returning a two-dimensional Numpy array.

pwkit.sherpa.**derive_identity_arf**(*name*, *arf*)
Create an "identity" ARF that has uniform sensitivity.

name The name of the ARF object to be created; passed to Sherpa.

arf An existing ARF object on which to base this one.

Returns: A new ARF1D object that has a uniform spectral response vector.

In many X-ray observations, the relevant background signal does not behave like an astrophysical source that is filtered through the telescope's response functions. However, I have been unable to get current Sherpa (version 4.9) to behave how I want when working with backround models that are *not* filtered through these response functions. This function constructs an "identity" ARF response function that has uniform sensitivity as a function of detector channel.

pwkit.sherpa.**derive_identity_rmf**(*name*, *rmf*)
Create an "identity" RMF that does not mix energies.

name The name of the RMF object to be created; passed to Sherpa.

rmf An existing RMF object on which to base this one.

Returns: A new RMF1D object that has a response matrix that is as close to diagonal as we can get in energy space, and that has a constant sensitivity as a function of detector channel.

In many X-ray observations, the relevant background signal does not behave like an astrophysical source that is filtered through the telescope's response functions. However, I have been unable to get current Sherpa (version 4.9) to behave how I want when working with backround models that are *not* filtered through these response functions. This function constructs an "identity" RMF response matrix that provides the best possible approximation of a passthrough "instrumental response": it mixes energies as little as possible and has a uniform sensitivity as a function of detector channel.

3.17 Synthetic photometry (`pwkit.synphot`)

Synthetic photometry and database of instrumental bandpasses.

The basic structure is that we have a registry of bandpass info. You can use it to create Bandpass objects that can perform various calculations, especially the computation of synthetic photometry given a spectral model. Some key attributes of each bandpass are pre-computed so that certain operations can be done without needing to load the actual bandpass profile (though so far none of these profiles are very large at all).

The bandpass definitions built into this module are:

- 2MASS (JHK)
- Bessell (UBVRI)
- GALEX (NUV, FUV)
- LMIRCam on LBT
- MEarth
- Mauna Kea Observatory (MKO) (JHKLM)
- SDSS (u' g' r' i' z')
- Swift (UVW1)
- WISE (1234)

Classes:

AlreadyDefinedError(fmt, *args)	Raised when re-registering bandpass info.
Bandpass	Computations regarding a particular filter bandpass.
NotDefinedError(fmt, *args)	Raised when needed bandpass info is unavailable.
Registry()	A registry of known bandpass properties.

Functions:

get_std_registry()	Get a Registry object pre-filled with information for standard telescopes.

Various internal utilities may be useful for reference but are not documented here.

Variables:

builtin_registrars	Hashtable of functions to register the builtin telescopes.

3.17.1 Example

```
from pwkit import synphot as ps, cgs as pc, msmt as pm
reg = ps.get_std_registry()
print(reg.telescopes()) # list known telescopes
print(reg.bands('2MASS')) # list known 2MASS bands
bp = reg.get('2MASS', 'Ks')
mag = 12.83
mjy = pm.repval(bp.mag_to_fnu(mag) * pc.jypercgs * 1e3)
print('%.2f mag is %.2f mjy in 2MASS/Ks' % (mag, mjy))
```

3.17.2 Conventions

It is very important to maintain consistent conventions throughout.

Wavelengths are measured in angstroms. Flux densities are either per-wavelength (f_λ, "flam") or per-frequency (f_ν, "fnu"). These are measured in units of erg/s/cm^2/Å and erg/s/cm^2/Hz, respectively. Janskys can be converted to f_ν by multiplying by cgs.cgsperjy. f_ν's and f_λ's can be interconverted for a given filter if you know its "pivot wavelength". Some of the routines below show how to calculate this and do the conversion. "AB magnitudes" can be directly converted to Janskys and, thus, f_ν's.

Filter bandpasses can be expressed in two conventions: either "equal-energy" (EE) or "quantum-efficiency" (QE). The former gives the response per unit energy across the band, while the latter gives the response per photon. The EE convention can be integrated directly against a model spectrum, so we store all bandpasses internally in this convention. CCDs are photon-counting devices and so their response curves are generally expressed in the QE convention. Interconversion is easy: EE = QE * λ.

We don't expect any particular normalization of bandpass response curves.

The "width" of a bandpass is not a well-defined quantity, but is often needed for display purposes or approximate calculations. We use the locations of the half-maximum points (in the EE convention) to define the band edges.

This module requires Scipy and Pandas. It doesn't reeeeallllly need Pandas but it's convenient.

3.17.3 References

Casagrande & VandenBerg (2014; arxiv:1407.6095) has a lot of good stuff; see also references therein.

References for specific bandpasses are given in their implementation docstrings.

3.17.4 The Registry class

pwkit.synphot.**get_std_registry**()
Get a Registry object pre-filled with information for standard telescopes.

class pwkit.synphot.**Registry**
A registry of known bandpass properties.

Instances of *Registry* have the following methods:

bands(telescope)	Return a list of bands associated with the specified telescope.
get(telescope, band)	Get a Bandpass object for a known telescope and filter.
register_bpass(telescope, klass)	Register a Bandpass class.

Continued on next page

Table 21 – continued from previous page

register_halfmaxes(telescope, band, lower, upper)	Register precomputed half-max points.
register_pivot_wavelength(telescope, band, wlen)	Register precomputed pivot wavelengths.
telescopes()	Return a list of telescopes known to this registry.

Registry.**bands**(*telescope*)
Return a list of bands associated with the specified telescope.

Registry.**get**(*telescope*, *band*)
Get a Bandpass object for a known telescope and filter.

Registry.**register_bpass**(*telescope*, *klass*)
Register a Bandpass class.

Registry.**register_halfmaxes**(*telescope*, *band*, *lower*, *upper*)
Register precomputed half-max points.

Registry.**register_pivot_wavelength**(*telescope*, *band*, *wlen*)
Register precomputed pivot wavelengths.

Registry.**telescopes**()
Return a list of telescopes known to this registry.

pwkit.synphot.**builtin_registrars = {'2MASS': <function register_2mass>, 'Bessell': <functi**
Hashtable of functions to register the builtin telescopes.

3.17.5 The Bandpass class

class pwkit.synphot.**Bandpass**
Computations regarding a particular filter bandpass.

The underlying bandpass shape is assumed to be sampled at discrete points. It is stored in `_data` and loaded on-demand. The object is a Pandas DataFrame containing at least the columns `wlen` and `resp`. The former holds the wavelengths of the sample points, in Ångström and in ascending order. The latter gives the response curve in the EE convention. No particular normalization is assumed. Other columns may be present but are not used generically.

Instances of *Bandpass* have the following attributes:

band	The name of this bandpass' associated band.
native_flux_kind	Which kind of flux this bandpass is calibrated to: 'flam', 'fnu', or 'none'.
registry	This object's parent Registry instance.
telescope	The name of this bandpass' associated telescope.

And the following methods:

calc_halfmax_points()	Calculate the wavelengths of the filter half-maximum values.
calc_pivot_wavelength()	Compute and return the bandpass' pivot wavelength.
halfmax_points()	Get the bandpass' half-maximum wavelengths.
jy_to_flam(jy)	Convert a f_ν flux density measured in Janskys to a f_λ flux density.

Continued on next page

Table 23 – continued from previous page

mag_to_flam(mag)	Convert a magnitude in this band to a f_λ flux density.
mag_to_fnu(mag)	Convert a magnitude in this band to a f_ν flux density.
pivot_wavelength()	Get the bandpass' pivot wavelength.
synphot(wlen, flam)	*wlen* and *flam* give a tabulated model spectrum in wavelength and f_λ units.
blackbody(T)	Calculate the contribution of a blackbody through this filter.

Detailed descriptions of attributes

Bandpass.**band = None**
The name of this bandpass' associated band.

Bandpass.**native_flux_kind = 'none'**
Which kind of flux this bandpass is calibrated to: 'flam', 'fnu', or 'none'.

Bandpass.**registry = None**
This object's parent Registry instance.

Bandpass.**telescope = None**
The name of this bandpass' associated telescope.

Detailed descriptions of methods

Bandpass.**calc_halfmax_points**()
Calculate the wavelengths of the filter half-maximum values.

Bandpass.**calc_pivot_wavelength**()
Compute and return the bandpass' pivot wavelength.

This value is computed directly from the bandpass data, not looked up in the Registry. Most of the values in the Registry were in fact derived from this function originally.

Bandpass.**halfmax_points**()
Get the bandpass' half-maximum wavelengths. These can be used to compute a representative bandwidth, or for display purposes.

Unlike calc_halfmax_points(), this function will use a cached value if available.

Bandpass.**jy_to_flam**(*jy*)
Convert a f_ν flux density measured in Janskys to a f_λ flux density.

This conversion is bandpass-dependent because it depends on the pivot wavelength of the bandpass used to measure the flux density.

Bandpass.**mag_to_flam**(*mag*)
Convert a magnitude in this band to a f_λ flux density.

It is assumed that the magnitude has been computed in the appropriate photometric system. The definition of "appropriate" will vary from case to case.

Bandpass.**mag_to_fnu**(*mag*)
Convert a magnitude in this band to a f_ν flux density.

It is assumed that the magnitude has been computed in the appropriate photometric system. The definition of "appropriate" will vary from case to case.

Bandpass.**pivot_wavelength**()
Get the bandpass' pivot wavelength.

Unlike calc_pivot_wavelength(), this function will use a cached value if available.

Bandpass.**synphot**(*wlen*, *flam*)
wlen and *flam* give a tabulated model spectrum in wavelength and f_λ units. We interpolate linearly over both the model and the bandpass since they're both discretely sampled.

Note that quadratic interpolation is both much slower and can blow up fatally in some cases. The latter issue might have to do with really large X values that aren't zero-centered, maybe?

I used to use the quadrature integrator, but Romberg doesn't issue complaints the way quadrature did. I should probably acquire some idea about what's going on under the hood.

Bandpass.**blackbody**(*T*)
Calculate the contribution of a blackbody through this filter. *T* is the blackbody temperature in Kelvin. Returns a band-averaged spectrum in f_λ units.

We use the composite Simpson's rule to integrate over the points at which the filter response is sampled. Note that this is a different technique than used by *synphot*, and so may give slightly different answers than that function.

3.17.6 Simple, careful conversions

fnu_cgs_to_flam_ang(fnu_cgs, pivot_angstrom)	erg/s/cm^2/Hz $\rightarrow$ erg/s/cm^2/Å
flam_ang_to_fnu_cgs(flam_ang, pivot_angstrom)	erg/s/cm^2/Å $\rightarrow$ erg/s/cm^2/Hz
abmag_to_fnu_cgs(abmag)	Convert an AB magnitude to f_ν in erg/s/cm^2/Hz.
abmag_to_flam_ang(abmag, pivot_angstrom)	Convert an AB magnitude to f_λ in erg/s/cm^2/Å.
ghz_to_ang(ghz)	Convert a photon frequency in GHz to its wavelength in Ångström.
flat_ee_bandpass_pivot_wavelength(wavelen1, ...)	Compute the pivot wavelength of a bandpass that's flat in equal-energy terms.
pivot_wavelength_ee(bpass)	Compute pivot wavelength assuming equal-energy convention.
pivot_wavelength_qe(bpass)	Compute pivot wavelength assuming quantum-efficiency convention.

pwkit.synphot.**fnu_cgs_to_flam_ang**(*fnu_cgs*, *pivot_angstrom*)
erg/s/cm^2/Hz $\rightarrow$ erg/s/cm^2/Å

pwkit.synphot.**flam_ang_to_fnu_cgs**(*flam_ang*, *pivot_angstrom*)
erg/s/cm^2/Å $\rightarrow$ erg/s/cm^2/Hz

pwkit.synphot.**abmag_to_fnu_cgs**(*abmag*)
Convert an AB magnitude to f_ν in erg/s/cm^2/Hz.

pwkit.synphot.**abmag_to_flam_ang**(*abmag*, *pivot_angstrom*)
Convert an AB magnitude to f_λ in erg/s/cm^2/Å. AB magnitudes are f_ν quantities, so a pivot wavelength is needed.

pwkit.synphot.**ghz_to_ang**(*ghz*)
Convert a photon frequency in GHz to its wavelength in Ångström.

pwkit.synphot.**flat_ee_bandpass_pivot_wavelength**(*wavelen1*, *wavelen2*)
Compute the pivot wavelength of a bandpass that's flat in equal-energy terms. It turns out to be their harmonic

mean.

pwkit.synphot.**pivot_wavelength_ee**(*bpass*)
Compute pivot wavelength assuming equal-energy convention.

bpass should have two properties, *resp* and *wlen*. The units of *wlen* can be anything, and *resp* need not be normalized in any particular way.

pwkit.synphot.**pivot_wavelength_qe**(*bpass*)
Compute pivot wavelength assuming quantum-efficiency convention. Note that this is NOT what we generally use in this module.

bpass should have two properties, *resp* and *wlen*. The units of *wlen* can be anything, and *resp* need not be normalized in any particular way.

3.17.7 Exceptions

AlreadyDefinedError(fmt, *args)	Raised when re-registering bandpass info.
NotDefinedError(fmt, *args)	Raised when needed bandpass info is unavailable.

class pwkit.synphot.**AlreadyDefinedError**(*fmt*, **args*)
Raised when re-registering bandpass info.

class pwkit.synphot.**NotDefinedError**(*fmt*, **args*)
Raised when needed bandpass info is unavailable.

3.18 Scaling relations for physical properties of ultra-cool dwarfs (pwkit.ucd_physics)

pwkit.ucd_physics - Physical calculations for (ultra)cool dwarfs.

These functions generally implement various nontrivial physical relations published in the literature. See docstrings for references.

Functions:

bcj_from_spt J-band bolometric correction from SpT.

bck_from_spt K-band bolometric correction from SpT.

load_bcah98_mass_radius Load Baraffe+ 1998 mass/radius data.

mass_from_j Mass from absolute J magnitude.

mk_radius_from_mass_bcah98 Radius from mass, using BCAH98 models.

tauc_from_mass Convective turnover time from mass.

pwkit.ucd_physics.**bcj_from_spt**(*spt*)
Calculate a bolometric correction constant for a J band magnitude based on a spectral type, using the fit of Wilking+ (1999AJ....117..469W).

spt - Numerical spectral type. M0=0, M9=9, L0=10, ...

Returns: the correction *bcj* such that *m_bol* = *j_abs* + *bcj*, or NaN if *spt* is out of range.

Valid values of *spt* are between 0 and 10.

pwkit.ucd_physics.**bck_from_spt**(*spt*)

Calculate a bolometric correction constant for a J band magnitude based on a spectral type, using the fits of Wilking+ (1999AJ....117..469W), Dahn+ (2002AJ....124.1170D), and Nakajima+ (2004ApJ...607..499N).

spt - Numerical spectral type. M0=0, M9=9, L0=10, ...

Returns: the correction *bck* such that *m_bol* = *k_abs* + *bck*, or NaN if *spt* is out of range.

Valid values of *spt* are between 2 and 30.

pwkit.ucd_physics.**load_bcah98_mass_radius**(*tablelines*, *metallicity=0*, *heliumfrac=0.275*, *age_gyr=5.0*, *age_tol=0.05*)

Load mass and radius from the main data table for the famous models of Baraffe+ (1998A&A...337..403B).

tablelines An iterable yielding lines from the table data file. I've named the file '1998A&A...337..403B_tbl1-3.dat' in some repositories (it's about 150K, not too bad).

metallicity The metallicity of the model to select.

heliumfrac The helium fraction of the model to select.

age_gyr The age of the model to select, in Gyr.

age_tol The tolerance on the matched age, in Gyr.

Returns: (mass, radius), where both are Numpy arrays.

The ages in the data table vary slightly at fixed metallicity and helium fraction. Therefore, there needs to be a tolerance parameter for matching the age.

pwkit.ucd_physics.**mass_from_j**(*j_abs*)

Estimate mass in cgs from absolute J magnitude, using the relationship of Delfosse+ (2000A&A...364..217D).

j_abs - The absolute J magnitude.

Returns: the estimated mass in grams.

If j_abs > 11, a fixed result of 0.1 Msun is returned. Values of j_abs < 5.5 are illegal and get NaN. There is a discontinuity in the relation at j_abs = 11, which yields 0.0824 Msun.

pwkit.ucd_physics.**mk_radius_from_mass_bcah98**(**args*, ***kwargs*)

Create a function that maps (sub)stellar mass to radius, based on the famous models of Baraffe+ (1998A&A...337..403B).

tablelines An iterable yielding lines from the table data file. I've named the file '1998A&A...337..403B_tbl1-3.dat' in some repositories (it's about 150K, not too bad).

metallicity The metallicity of the model to select.

heliumfrac The helium fraction of the model to select.

age_gyr The age of the model to select, in Gyr.

age_tol The tolerance on the matched age, in Gyr.

Returns: a function mtor(mass_g), return a radius in cm as a function of a mass in grams. The mass must be between 0.05 and 0.7 Msun.

The ages in the data table vary slightly at fixed metallicity and helium fraction. Therefore, there needs to be a tolerance parameter for matching the age.

This function requires Scipy.

pwkit.ucd_physics.**tauc_from_mass**(*mass_g*)

Estimate the convective turnover time from mass, using the method described in Cook+ (2014ApJ...785...10C).

mass_g - UCD mass in grams.

Returns: the convective turnover timescale in seconds.

Masses larger than 1.3 Msun are out of range and yield NaN. If the mass is <0.1 Msun, the turnover time is fixed at 70 days.

The Cook method was inspired by the description in McLean+ (2012ApJ...746...23M). It is a hybrid of the method described in Reiners & Basri (2010ApJ...710..924R) and the data shown in Kiraga & Stepien (2007AcA....57..149K). However, this version imposes the 70-day cutoff in terms of mass, not spectral type, so that it is entirely defined in terms of a single quantity.

There are discontinuities between the different break points! Any future use should tweak the coefficients to make everything smooth.

CHAPTER 4

Command-line tools

This documentation has a lot of stubs.

4.1 Quick astronomical calculations (`astrotool`)

pwkit.cli.astrotool - the ‘astrotool’ program.

4.2 Quick operations on astronomical images (`pwkit.cli.imtool`)

pwkit.cli.imtool - the ‘imtool’ program.

4.3 Single-command compilation of LaTeX documents (`latexdriver`)

pwkit.cli.latexdriver - the ‘latexdriver’ program.

This used to be a nice little shell script, but for portability it’s better to do this in Python. And now we can optionally provide some BibTeX-related magic.

4.4 Wrap the output of a sub-program with extra information (`wrapout`)

pwkit.cli.wrapout - the ‘wrapout’ program.

CHAPTER 5

Data Visualization

This documentation has a lot of stubs.

5.1 Mapping arbitrary data to color scales (`pwkit.colormaps`)

pwkit.colormaps – tools to conver arrays of real-valued data to other formats (usually, RGB24) for visualization.

TODO: "heated body" map.

The main interface is the *factory_map* dictionary from colormap names to factory functions. *base_factory_names* lists the names of a set of color maps. Additional ones are available with the suffixes "_reverse" and "_sqrt" that apply the relevant transforms.

The factory functions return another function, the "mapper". Each mapper takes a single argument, an array of values between 0 and 1, and returns the mapped colors. If the input array has shape S, the returned value has a shape (S + (3,)), with mapped[. . . ,0] being the R values, between 0 and 1, etc.

Example:

> data = np.array ([<things between 0 and 1>]) mapper = factory_map['cubehelix_blue']() rgb = mapper (data) green_values = rgb[:,1] last_rgb = rgb[-1]

The basic colormap names are:

moreland_bluered Divergent colormap from intense blue (at 0) to intense red (at 1), passing through white

cubehelix_dagreen From black to white through rainbow colors

cubehelix_blue From black to white, with blue hues

pkgw From black to red, through purplish

black_to_white, black_to_red, black_to_green, black_to_blue From black to the named colors.

white_to_black, white_to_red, white_to_green, white_to_blue From white to the named colors.

The mappers can also take keyword arguments, including at least "transform", which specifies simple transforms that can be applied to the colormaps. These are (in terms of symbolic constants and literal string values):

'none' - No transform (the default) 'reverse' - x -> 1 - x (reverses the colormap) 'sqrt' - x -> sqrt (x)

For each transform other than "none", *factory_map* contains an entry with an underscore and the transform name applied (e.g., "pkgw_reverse") that has that transform applied.

The initial inspiration was an implementation of the ideas in "Diverging Color Maps for Scientific Visualization (Expanded)", Kenneth Moreland,

http://www.cs.unm.edu/~kmorel/documents/ColorMaps/index.html

I've realized that I'm not too fond of the white mid-values in these color maps in many cases. So I also added an implementation of the "cube helix" color map, described by D. A. Green in

"A colour scheme for the display of astronomical intensity images" http://adsabs.harvard.edu/abs/2011BASI...39..289G (D. A. Green, 2011 Bull. Ast. Soc. of India, 39 289)

I made up the pkgw map myself (who'd have guessed?).

5.2 Tracing contours (`pwkit.contours`)

pwkit.contours - Tracing contours in functions and data.

Uses my own homebrew algorithm. So far, it's only tested on extremely well-behaved functions, so probably doesn't cope well with poorly-behaved ones.

`pwkit.contours.`**`analytic_2d`**(*f*, *df*, *x0*, *y0*, *maxiters=5000*, *defeta=0.05*, *netastep=12*, *vtol1=0.001*, *vtol2=1e-08*, *maxnewt=20*, *dorder=7*, *goright=False*)

Sample a contour in a 2D analytic function. Arguments:

f A function, mapping (x, y) -> z.

df The partial derivative: df (x, y) -> [dz/dx, dz/dy]. If None, the derivative of f is approximated numerically with scipy.derivative.

x0 Initial x value. Should be of "typical" size for the problem; avoid 0.

y0 Initial y value. Should be of "typical" size for the problem; avoid 0.

Optional arguments:

maxiters Maximum number of points to create. Default 5000.

defeta Initially offset by distances of defeta*[df/dx, df/dy] Default 0.05.

netastep Number of steps between defeta and the machine resolution in which we test eta values for goodness. (OMG FIXME doc). Default 12.

vtol1 Tolerance for constancy in the value of the function in the initial offset step. The value is only allowed to vary by `f(x0,y0) * vtol1`. Default 1e-3.

vtol2 Tolerance for constancy in the value of the function in the along the contour. The value is only allowed to vary by `f(x0,y0) * vtol2`. Default 1e-8.

maxnewt Maximum number of Newton's method steps to take when attempting to hone in on the desired function value. Default 20.

dorder Number of function evaluations to perform when evaluating the derivative of f numerically. Must be an odd integer greater than 1. Default 7.

goright If True, trace the contour rightward (as looking uphill), rather than leftward (the default).

5.3 Utilities for data visualization (pwkit.data_gui_helpers)

pwkit.data_gui_helpers - helpers for GUIs looking at data arrays

Classes:

Clipper - Map data into [0,1] ColorMapper - Map data onto RGB colors using *pwkit.colormaps* Stretcher - Map data within [0,1] using a stretch like sqrt, etc.

Functions:

data_to_argb32 - Turn arbitrary data values into ARGB32 colors. data_to_imagesurface - Turn arbitrary data values into a Cairo ImageSurface.

pwkit.data_gui_helpers.**data_to_argb32**(*data*, *cmin=None*, *cmax=None*, *stretch='linear'*, *cmap='black_to_blue'*)

Turn arbitrary data values into ARGB32 colors.

There are three steps to this process: clipping the data values to a maximum and minimum; stretching the spacing between those values; and converting their amplitudes into colors with some kind of color map.

***data* - Input data; can (and should) be a MaskedArray if some values are** invalid.

***cmin* - The data clip minimum; all values <= cmin are treated** identically. If None (the default), *data.min()* is used.

***cmax* - The data clip maximum; all values >= cmax are treated** identically. If None (the default), *data.max()* is used.

***stretch* - The stretch function name; 'linear', 'sqrt', or 'square'; see** the Stretcher class.

***cmap* - The color map name; defaults to 'black_to_blue'. See the** *pwkit.colormaps* module for more choices.

Returns a Numpy array of the same shape as *data* with dtype *np.uint32*, which represents the ARGB32 colorized version of the data. If your colormap is restricted to a single R or G or B channel, you can make color images by bitwise-or'ing together different such arrays.

pwkit.data_gui_helpers.**data_to_imagesurface**(*data*, ***kwargs*)

Turn arbitrary data values into a Cairo ImageSurface.

The method and arguments are the same as data_to_argb32, except that the data array will be treated as 2D, and higher dimensionalities are not allowed. The return value is a Cairo ImageSurface object.

Combined with the write_to_png() method on ImageSurfaces, this is an easy way to quickly visualize 2D data.

class pwkit.data_gui_helpers.**Stretcher**(*mode*)

Assumes that its inputs are in [0, 1]. Maps its outputs to the same range.

offset_cbrt(*dest*)

This stretch is useful when you have values that are symmetrical around zero, and you want to enhance contrasts at small values while preserving sign.

5.4 Easy visualization of matrices with GTK+ version 2 (pwkit.ndshow_gtk2)

5.5 Easy visualization of matrices with GTK+ version 3 (pwkit.ndshow_gtk3)

CHAPTER 6

Data input and output

This documentation has a lot of stubs.

6.1 Streaming output from other programs (`pwkit.slurp`)

The *`pwkit.slurp`* module makes it convenient to read output generated by other programs. This is accomplished with a context-manager class known as *`Slurper`*, which is built on top of the standard `subprocess.Popen` module.

The chief advantage of *`Slurper`* above `subprocess.Popen` is that it provides convenient, *streaming* access to subprogram output, maintaining the distinction between "stdout" (standard output, written to file descriptor #1) and "stderr" (standard error, written to file descriptor #2). It can also forward signals to the child program.

Standard usage might look like:

```
from pwkit import slurp
argv = ['cat', '/etc/passwd']

with slurp.Slurper (argv, linebreak=True) as slurper:
    for etype, data in slurper:
        if etype == 'stdout':
            print ('got line:', data)

print ('exit code was:', slurper.proc.returncode)
```

`Slurper` is a context manager to ensure that the child process is always cleaned up. Within the context manager body, you should iterate over the *`Slurper`* instance to get a series of "event" 2-tuples consisting of a Unicode string giving the event type, and the event data. Most, but not all, events have to do with receiving data over the stdout or stderr pipes. The events are:

Event type	Event data	Description
`"stdout"`	The output	Data were received from the subprogram's standard output.
`"stderr"`	The output	Data were received from the subprogram's standard error.
`"forwarded-signal"`	The signal number	This process received a signal and forwarded it to the child.
`"timeout"`	`None`	No data were received from the child within a fixed timeout.

The data provided on the `"stdout"` and `"stderr"` events follow the usual Python patterns for EOF. Namely, when either of those pipes is closed by the subprocess, a final event is sent in which the data payload has zero length. (It may be either a bytes object or a Unicode string depending on whether decoding is enabled; see below.)

Warning: It is important to realize that programs that use the standard C I/O routines, such as Python programs, buffer their output by default. The *pwkit.slurp* module may appear to be having problems while really the child program is batching up its output and writing it all at once. This can be surprising because the default behavior is line-buffered when `stdout` is connected to a TTY (as when you run programs in your terminal), but buffered in large blocks when connected to a pipe (as when using this module). On systems built on `glibc`, you can control this by using the `stdbuf` program to launch your subprogram with different buffering options. To run the command `foo bar` with both stdout and stderr buffered at the line level, run `stdbuf -oL -eL foo bar`. To disable buffering on both streams, run `stdbuf -o0 -e0 foo bar`.

class pwkit.slurp.**Slurper**(*argv=None*, *env=None*, *cwd=None*, *propagate_signals=True*, *timeout=10*, *linebreak=False*, *encoding=None*, *stdin=slurp.Redirection.DevNull*, *stdout=slurp.Redirection.Pipe*, *stderr=slurp.Redirection.Pipe*, *executable=None*)

Construct a context manager used to read output from a subprogram. *argv* is used to launch the subprogram using `subprocess.Popen` with the *shell* keyword set to False. *env*, *cwd*, *executable*, *stdin*, *stdout*, and *stderr* are forwarded to the `subprocess.Popen` constructor as well.

Regarding the redirection parameters *stdin*, *stdout*, and *stderr*, the constants in the *Redirection* object gives more user-friendly names to the analogues provided by the `subprocess` module, with the addition of a `Redirection.DevNull` option emulating behavior added in Python 3. Otherwise these values are passed to `subprocess.Popen` verbatim, so you can use anyting that `subprocess.Popen` would accept. Keep in mind that you can only fetch the subprogram's output if one or both of the output paramers are set to `Redirection.Pipe`!

If *propagate_signals* is true, signals received by the parent process will be forwarded to the child process. This can be valuable to obtain correct behavior on SIGINT, for instance. Forwarded signals are SIGHUP, SIGINT, SIGQUIT, SIGTERM, SIGUSR1, and SIGUSR2. This is done by overwriting the calling process' Python signal handlers; the original handlers are restored upon exit from the with-statement block.

If *linebreak* is true, output from the child process will be gathered into whole lines (split by `"\n"`) before being sent to the caller. *The newline characters will be discarded*, making it impossible to tell whether the final line of output ended with a newline or not.

If *encoding* is not `None`, a decoder will be created with `codecs.getincrementaldecoder()` and the subprocess output will be converted from bytes to Unicode before being returned to the calling process.

timeout sets the timeout for the internal `select.select()` call used to check for output from the subprogram. It is measured in seconds.

Slurper instances have attributes *argv*, *env*, *cwd*, *executable*, *propagate_signals*, :*timeout*, *linebreak*, attr:*encoding*, *stdin*, :*stdout*, and *stderr* recording the construction parameters.

pwkit.slurp.**Redirection**

An enum-like object defining ways to redirect the I/O streams of the subprogram. These values are identical to those used in `subprocess` but with nicer names.

Constant	Meaning
`Redirection.Pipe`	Pipe output to the calling program.
`Redirection.Stdout`	Only valid for `stderr`; merge it with `stdout`
`Redirection.DevNull`	Direct input from `/dev/null`, or output thereto.

The whole *raison d'être* of *`pwkit.slurp`* is to make it easy to communicate output between programs, so you probably will probably want to use `Redirection.Pipe` for `stdout` and `stderr` most of the time.

6.1.1 `Slurper` reference

`Slurper.`**`proc`**
The `subprocess.Popen` instance of the child program. After the program has exited, you can access its exit code as `Slurper.proc.returncode`.

`Slurper.`**`argv`**
The `argv` of the program to be launched.

`Slurper.`**`env`**
Environment dictionary for the program to be launched.

`Slurper.`**`cwd`**
The working directory for the program to be launched.

`Slurper.`**`executable`**
The name of the executable to launch (`argv[0]` is allowed to differ from this).

`Slurper.`**`propagate_signals`**
Whether to forward the subprogram any signals that are received by the calling process.

`Slurper.`**`timeout`**
The timeout (in seconds) for waiting for output from the child program. If nothing is received, a `"timeout"` event is generated.

`Slurper.`**`linebreak`**
Whether to gather the subprogram output into textual lines.

`Slurper.`**`encoding`**
The encoding to be used to decode the subprogram output from bytes to Unicode, or `None` if no such decoding is to be done.

`Slurper.`**`stdin`**
How to redirect the standard input of the subprogram, if at all.

`Slurper.`**`stdout`**
How to redirect the standard output of the subprogram, if at all. If not `Pipe`, no `"stdout"` events will be received.

`Slurper.`**`stderr`**
How to redirect the standard error of the subprogram, if at all. If not `Pipe`, no `"stderr"` events will be received. If `Stdout`, events that would have had a type of `"stderr"` will have a type of `"stdout"` instead.

6.2 A simple "ini" file format (`pwkit.inifile`)

A simple parser for ini-style files that's better than Python's ConfigParser/configparser.

Functions:

read Generate a stream of *pwkit.Holder* instances from an ini-format file.

mutate Rewrite an ini file chunk by chunk.

write Write a stream of *pwkit.Holder* instances to an ini-format file.

mutate_stream Lower-level version; only operates on streams, not path names.

read_stream Lower-level version; only operates on streams, not path names.

write_stream Lower-level version; only operates on streams, not path names.

mutate_in_place Rewrite an ini file specififed by its path name, in place.

exception pwkit.inifile.**InifileError**(*fmt*, **args*)

pwkit.inifile.**mutate_stream**(*instream*, *outstream*)
Python 3 compat note: we're assuming *stream* gives bytes not unicode.

pwkit.inifile.**read_stream**(*stream*)
Python 3 compat note: we're assuming *stream* gives bytes not unicode.

pwkit.inifile.**write_stream**(*stream*, *holders*, *defaultsection=None*)
Very simple writing in ini format. The simple stringification of each value in each Holder is printed, and no escaping is performed. (This is most relevant for multiline values or ones containing pound signs.) *None* values are skipped.

Arguments:

stream A text stream to write to.

holders An iterable of objects to write. Their fields will be written as sections.

defaultsection=None Section name to use if a holder doesn't contain a *section* field.

pwkit.inifile.**write**(*stream_or_path*, *holders*, ***kwargs*)
Very simple writing in ini format. The simple stringification of each value in each Holder is printed, and no escaping is performed. (This is most relevant for multiline values or ones containing pound signs.) *None* values are skipped.

Arguments:

stream A text stream to write to.

holders An iterable of objects to write. Their fields will be written as sections.

defaultsection=None Section name to use if a holder doesn't contain a *section* field.

6.3 Outputting data in LaTeX format (pwkit.latex)

pwkit.latex - various helpers for the LaTeX typesetting system.

6.3.1 Classes

Referencer Accumulate a numbered list of bibtex references, then output them.

TableBuilder Create awesome deluxetables programmatically.

6.3.2 Functions

latexify_l3col Format value in LaTeX, suitable for tables of limit values.

latexify_n2col Format a number in LaTeX in 2-column decimal-aligned formed.

latexify_u3col Format value in LaTeX, suitable for tables of uncertain values.

latexify Format a value in LaTeX appropriately.

6.3.3 Helpers for TableBuilder

AlignedNumberFormatter Format numbers, aligning them at the decimal point.

BasicFormatter Base class for formatters.

BoolFormatter Format a boolean; default is True -> bullet, False -> nothing.

LimitFormatter Format measurements for a table of limits.

MaybeNumberFormatter Format numbers with a fixed number of decimal places, or objects with __pk_latex__().

UncertFormatter Format measurements for a table of detailed uncertainties.

WideHeader Helper for multi-column headers.

XXX: Barely tested!

class pwkit.latex.**AlignedNumberFormatter**(*nplaces=1*)

Format numbers. Allows the number of decimal places to be specified, and aligns the numbers at the decimal point.

colinfo(*builder*)

Return (nlcol, colspec, headprefix), where:

nlcol - The number of LaTeX columns encompassed by this logical column.

colspec - Its LaTeX column specification (None to force user to specify).

headprefix - Prefix applied before heading items in {} (e.g., "colhead").

class pwkit.latex.**BasicFormatter**

Base class for formatting table cells in a TableBuilder.

Generally a formatter will also provide methods for turning input data into fancified LaTeX output that can be used by the column's "data function".

colinfo(*builder*)

Return (nlcol, colspec, headprefix), where:

nlcol - The number of LaTeX columns encompassed by this logical column.

colspec - Its LaTeX column specification (None to force user to specify).

headprefix - Prefix applied before heading items in {} (e.g., "colhead").

class pwkit.latex.**BoolFormatter**

Format booleans. Attributes *truetext* and *falsetext* set what shows up for true and false values, respectively.

colinfo(*builder*)

Return (nlcol, colspec, headprefix), where:

nlcol - The number of LaTeX columns encompassed by this logical column.

colspec - Its LaTeX column specification (None to force user to specify).

headprefix - Prefix applied before heading items in {} (e.g., "colhead").

class pwkit.latex.**LimitFormatter**

Format measurements (cf pwkit.msmt) with nice-looking limit information. Specific uncertainty information is discarded. The default formats do not involve fancy subscripts or superscripts, so row struts are not needed ... by default.

colinfo(*builder*)

Return (nlcol, colspec, headprefix), where:

nlcol - The number of LaTeX columns encompassed by this logical column.

colspec - Its LaTeX column specification (None to force user to specify).

headprefix - Prefix applied before heading items in {} (e.g., "colhead").

class pwkit.latex.**MaybeNumberFormatter**(*nplaces=1*, *align='c'*)

Format Python objects. If it's a number, format it as such, without any fancy column alignment, but with a specifiable number of decimal places. Otherwise, call latexify() on it.

colinfo(*builder*)

Return (nlcol, colspec, headprefix), where:

nlcol - The number of LaTeX columns encompassed by this logical column.

colspec - Its LaTeX column specification (None to force user to specify).

headprefix - Prefix applied before heading items in {} (e.g., "colhead").

class pwkit.latex.**Referencer**

Accumulate a numbered list of bibtex references. Methods:

refkey(bibkey) Return a string that should be used to give a numbered reference to the given bibtex key. "this-work" is handled specially.

dump() Return a string with citet{} commands identifing all of the numbered references.

Attributes:

thisworktext text referring to "this work"; defaults to that.

thisworkmarker special symbol used to denote "this work"; defaults to star.

Bibtex keys beginning with asterisks have the rest of their value used for the citation text, rather than "citet{<key>}".

class pwkit.latex.**TableBuilder**(*label*)

Build and then emit a nice deluxetable.

Methods:

addcol(headings, datafunc, formatter=None, colspec=None, numbering='(%d)') Define a logical column.

addnote(key, text) Define a table note that can appear in cells.

addhcline(headerrowix, logcolidx, latexdeltastart, latexdeltaend) Add a horizontal line between columns.

notemark(key) Return a tablenotemark{} command for the specified note key.

emit(stream, items) Write the table, with one row for each thing in *items*, to the stream.

If an item has an attribute *tb_row_preamble*, that text is written verbatim before that corresponding row is output.

Attributes:

environment The name of the latex environment to use, default "deluxetable". You may want to specify "deluxetable*", or "mydeluxetable" if using a hacked package.

label The latex reference label of the table. Mandatory.

note A note at the table footer ("tablecomments{}" in LaTeX).

preamble Commands for table preamble. See below.

refs Contents of the table References section.

title Table title. Default "Untitled table".

widthspec Passed to tablewidth{}; default "0em" = auto-widen.

numbercols If True, number each column. This can be disabled on a col-by-col basis by calling *addcol* with *numbering* set to False.

final_double_backslash If True, end the final table row with a "". AAStex6 requires this, giving an error about a misplaced 'omit' if you don't provide one. On the other hand, classic TeX tables look worse if you do provide this.

Legal preamble commands are:

```
\rotate
\tablenum{<manual table identifier>}
\tabletypesize{<font size command>}
```

The commands tablecaption, tablecolumns, tablehead, and tablewidth are handled specially.

If tablewidth{} is not provided, the table is set at full width, not its natural width, which is a lame default. The default *widthspec* lets us auto-widen while providing a clear avenue to customizing the width.

addcol(*headings*, *datafunc*, *formatter=None*, *colspec=None*, *numbering='(%d)'*)
Define a logical column. Arguments:

headings A string, or list of strings and WideHeaders. The headings are stacked vertically in the table header section.

datafunc Return LaTeX for this cell. Call spec should be (item, [formatter, [tablebuilder]]).

formatter The formatter to use; defaults to a new BasicFormatter.

colspec The LaTeX column specification letters to use; defaults to 'c's.

numbering If non-False, a format for writing this column's number; if False, no number is written.

addhcline(*headerrowidx*, *logcolidx*, *latexdeltastart*, *latexdeltaend*)
Adds a horizontal line below a limited range of columns in the header section. Arguments:

headerrowidx - The 0-based row number *below* which the line will be drawn; i.e. 0 means that the line will be drawn below the first row of header cells.

logcolidx - The 0-based 'logical' column number relative to which the line will be placed; i.e. 1 means that the line placement will be relative to the second column defined in an addcol() call.

latexdeltastart - The relative position at which to start drawing the line relative to that logical column, in LaTeX columns; typically going to be zero.

latexdeltaend - The relative position at which to finish drawing the line, in the standard Python non-inclusive sense. I.e., if you want to underline two LaTeX columns, latexdeltaend = latexdeltastart + 2.

class pwkit.latex.**UncertFormatter**
Format measurements (cf. pwkit.msmt) with detailed uncertainty information, possibly including asymmetric uncertainties. Because of the latter possibility, table rows have to be made extra-high to maintain evenness.

colinfo (*builder*)
Return (nlcol, colspec, headprefix), where:

nlcol - The number of LaTeX columns encompassed by this logical column.

colspec - Its LaTeX column specification (None to force user to specify).

headprefix - Prefix applied before heading items in {} (e.g., "colhead").

class pwkit.latex.**WideHeader** (*nlogcols*, *content*, *align='c'*)
Information needed for constructing wide table headers.

nlogcols - Number of logical columns consumed by this header. content - The LaTeX to insert for this header's content. align - The alignment of this header; default 'c'.

Rendered as multicolumn{nlatex}{align}{content}, where *nlatex* is the number of LaTeX columns spanned by this header – which may be larger than *nlogcols* if certain logical columns span multiple LaTeX columns.

pwkit.latex.**latexify_l3col** (*obj*, ***kwargs*)
Convert an object to special LaTeX for limit tables.

This conversion is meant for limit values in a table. The return value should span three columns. The first column is the limit indicator: <, >, ~, etc. The second column is the whole part of the value, up until just before the decimal point. The third column is the decimal point and the fractional part of the value, if present. If the item being formatted does not fit this schema, it can be wrapped in something like 'multicolumn{3}{c}{...}'.

pwkit.latex.**latexify_n2col** (*x*, *nplaces=None*, ***kwargs*)
Render a number into LaTeX in a 2-column format, where the columns split immediately to the left of the decimal point. This gives nice alignment of numbers in a table.

pwkit.latex.**latexify_u3col** (*obj*, ***kwargs*)
Convert an object to special LaTeX for uncertainty tables.

This conversion is meant for uncertain values in a table. The return value should span three columns. The first column ends just before the decimal point in the main number value, if it has one. It has no separation from the second column. The second column goes from the decimal point until just before the "plus-or-minus" indicator. The third column goes from the "plus-or-minus" until the end. If the item being formatted does not fit this schema, it can be wrapped in something like 'multicolumn{3}{c}{...}'.

pwkit.latex.**latexify** (*obj*, ***kwargs*)
Render an object in LaTeX appropriately.

6.4 Reading and writing data tables with types and uncertainties (pwkit.tabfile)

pwkit.tabfile - I/O with typed tables of uncertain measurements.

Functions:

read - Read a typed table file. vizread - Read a headerless table file, with columns specified separately write - Write a typed table file.

The table format is line-oriented text. Hashes denote comments. Initial lines of the form "colname = value" set a column name that gets the same value for every item in the table. The header line is prefixed with an @ sign. Subsequent lines are data rows.

pwkit.tabfile.**read** (*path*, *tabwidth=8*, ***kwargs*)
Read a typed tabular text file into a stream of Holders.

Arguments:

path The path of the file to read.

tabwidth=8 The tab width to assume. Please don't monkey with it.

mode='rt' The file open mode (passed to io.open()).

noexistok=False If True and the file is missing, treat it as empty.

****kwargs** Passed to io.open ().

Returns a generator for a stream of *pwkit.Holder's, each of which will contain ints, strings, or some kind of measurement (cf 'pwkit.msmt*).

pwkit.tabfile.**vizread**(*descpath*, *descsection*, *tabpath*, *tabwidth=8*, ***kwargs*)
Read a headerless tabular text file into a stream of Holders.

Arguments:

descpath The path of the table description ini file.

descsection The section in the description file to use.

tabpath The path to the actual table data.

tabwidth=8 The tab width to assume. Please don't monkey with it.

mode='rt' The table file open mode (passed to io.open()).

noexistok=False If True and the file is missing, treat it as empty.

****kwargs** Passed to io.open ().

Returns a generator of a stream of *pwkit.Holder's, each of which will contain ints, strings, or some kind of measurement (cf 'pwkit.msmt*). In this version, the table file does not contain a header, as seen in Vizier data files. The corresponding section in the description ini file has keys of the form "colname = <start> <end> [type]", where <start> and <end> are the **1-based** character numbers defining the column, and [type] is an optional specified of the measurement type of the column (one of the usual b, i, f, u, Lu, Pu).

pwkit.tabfile.**write**(*stream*, *items*, *fieldnames*, *tabwidth=8*)
Write a typed tabular text file to the specified stream.

Arguments:

stream The destination stream.

items An iterable of items to write. Two passes have to be made over the items (to discover the needed column widths), so this will be saved into a list.

fieldnames Either a list of field name strings, or a single string. If the latter, it will be split into a list with .split().

tabwidth=8 The tab width to use. Please don't monkey with it.

Returns nothing.

6.5 An "ini" file format with typed, uncertain data (pwkit.tinifile)

pwkit.tinifile - Dealing with typed ini-format files full of measurements.

Functions:

read Generate *pwkit.Holder* instances of measurements from an ini-format file.

write Write *pwkit.Holder* instances of measurements to an ini-format file.

read_stream Lower-level version; only operates on streams, not path names.

write_stream Lower-level version; only operates on streams, not path names.

pwkit.tinifile.**write_stream**(*stream*, *holders*, *defaultsection=None*, *extrapos=()*, *sha1sum=False*, ***kwargs*)

extrapos is basically a hack for multi-step processing. We have some flux measurements that are computed from luminosities and distances. The flux value is therefore an unwrapped Uval, which doesn't retain memory of any positivity constraint it may have had. Therefore, if we write out such a value using this routine, we may get something like *fx:u = 1pm1*, and the next time it's read in we'll get negative fluxes. Fields listed in *extrapos* will have a "P" constraint added if they are imprecise and their typetag is just "f" or "u".

6.6 Converting Unicode to LaTeX notation (pwkit.unicode_to_latex)

unicode_to_latex - what it says

Provides `unicode_to_latex(u)`, `unicode_to_latex_bytes(u)`, and `unicode_to_latex_string(u)`.

`unicode_to_latex_bytes` returns ASCII bytes that can be fed to LaTeX to reproduce the Unicode string 'u' as closely as possible.

`unicode_to_latex_string` returns a Unicode string rather than bytes.

`unicode_to_latex` returns the `str` type: bytes on Python 2, Unicode on Python 3.

CHAPTER 7

External Software Environments

This documentation has a lot of stubs.

7.1 CASA (`pwkit.environments.casa`)

The *`pwkit.environments.casa`* package provides convenient interfaces to the CASA package for analysis of radio interferometric data. In particular, it makes it much easier to build scripts and modules for automated data analysis.

This module does *not* require a full CASA installation, but it does depend on the availability of the `casac` Python module, which provides Python access to the C++ code that drives most of CASA's low-level functionality. By far the easiest way to obtain this module is to use an installation of Anaconda or Miniconda Python and install the casa-python package provided by Peter Williams, which builds on the infrastructure provided by the conda-forge project.

Alternatively, you can try to install CASA and extract the `casac` module from its files as described here. Or you can try to install *this module* inside the Python environment bundled with CASA. Or you can compile and underlying CASA C++ code yourself. But, using the pre-built packages is going to be by far the simplest approach and is **strongly** recommended.

7.1.1 Outline of functionality

This package provides several kinds of functionality.

- The *`pwkit.environments.casa.tasks`* module provides straightforward programmatic access to a wide selection of commonly-used CASA takes like `gaincal` and `setjy`.
- `pwkit` installs a command-line program, `casatask`, which provides command-line access to the tasks implemented in the *`tasks`* module, much as MIRIAD tasks can be driven straight from the command line.
- The *`pwkit.environments.casa.util`* module provides the lowest-level access to the "tool" structures defined in the C++ code.
- Several modules like *`pwkit.environments.casa.dftphotom`* provide original analysis features; *`dftphotom`* extracts light curves of point sources from calibrated visibility data.

- If you do have a full CASA installation available on your compuer, the *pwkit.environments.casa.scripting* module allows you to drive it from Python code in a way that allows you to analyze its output, check for error conditions, and so on. This is useful for certain features that are not currently available in the *tasks* module.

7.1.2 More detailed documentation

Programmatic access to CASA tasks (`pwkit.environments.casa.tasks`)

The way that the official `casapy` code is written, it's basically impossible to import its tasks into a straight-Python environment. (Trust me, I've tried.) So, this module more-or-less duplicates lots of CASA code. But this module also tries to provide to provide saner semantics and interfaces.

The goal is to make task-like functionality available in a real Python library, with no side effects, so that data processing can be scripted tractably. These tasks are also accessible through the `casatask` command line program provided with *pwkit*.

Example programmatic usage:

```
from pwkit.environments.casa import tasks

vis_path = 'mydataset.ms'

# A basic listobs:

for output_line in tasks.listobs(vis_path):
    print(output_line)

# Split a dataset with filtering and averaging:

cfg = tasks.SplitConfig()
cfg.vis = vis_path
cfg.out = 'new-' + vis_path
cfg.spw = '0~8'
cfg.timebin = 60 # seconds
tasks.split(cfg)
```

This module implements the following analysis tasks. Some of them are extremely close to CASA tasks of the same name; some are streamlined; some are not provided in CASA at all.

- *applycal* — use calibration tables to generate `CORRECTED_DATA` from `DATA`.
- *bpplot* — plot a bandpass calibration table; an order of magnitude faster than the CASA equivalent.
- *clearcal* — fill calibration tables with default.
- *concat* — concatenate two data sets.
- *delcal* — delete the `MODEL_DATA` and/or `CORRECTED_DATA` MS columns.
- *elplot* — plot elevations of the fields observed in an MS.
- *extractbpflags* — extract a table of channel flags from a bandpass calibration table.
- *flagcmd* — apply flags to an MS using a generic infrastructure.
- *flaglist* — apply a textual list of flag commands to an MS.
- *flagzeros* — flag zero-valued visibilites in an MS.
- *fluxscale* — use a flux density model to absolutely scale a gain calibration table.

- *ft* — generate model visibilities from an image.
- *gaincal* — solve for a gain calibration table.
- *gencal* — generate various calibration tables that do not depend on the actual visibility data in an MS.
- *getopacities* — estimate atmospheric opacities for an observation.
- *gpdetrend* — remove long-term phase trends from a complex-gain calibration table.
- *gpplot* — plot a complex-gain calibration table in a sensible way.
- *image2fits* — convert a CASA image to FITS format.
- *importalma* — convert an ALMA SDM file to MS format.
- *importevla* — convert an EVLA SDM file to MS format.
- *listobs* — print out the basic observational characteristics in an MS data set.
- *listsdm* — print out the basic observational characteristics in an SDM data set.
- *mfsclean* — image calibrated data using MFS and CLEAN.
- *mjd2date* — convert an MJD to a date in the textual format used by CASA.
- *mstransform* — perform basic streaming transforms on an MS data, such as time averaging, Hanning smoothing, and/or velocity resampling.
- *plotants* — plot the positions of the antennas used in an MS.
- *plotcal* — plot a complex-gain calibration table using CASA's default infrastructure.
- *setjy* — insert absolute flux density calibration information into a dataset.
- *split* — extract a subset of an MS.
- *tsysplot* — plot how the typical system temperature varies over time.
- *uvsub* — fill `CORRECTED_DATA` with `DATA - MODEL_DATA`.
- *xyphplot* — plot a frequency-dependent X/Y phase calibration table.

The following tasks are provided by the associated command line program, `casatask`, but do not have dedicated functions in this module.

- `closures` — see `closures`.
- *delmod* — this is too trivial to need its own function.
- `dftdynspec` — see `dftdynspec`.
- `dftphotom` — see `dftphotom`.
- `dftspect` — see `dftspect`.
- *flagmanager* — more specialized functions should be used in code.
- `gpdiagnostics` — see `gpdiagnostics`.
- `polmodel` — see `polmodel`.
- `spwglue` — see `spwglue`.

Tasks

This documentation is automatically generated from text that is targeted at the command-line tasks, and so may read a bit strangely at times.

applycal

pwkit.environments.casa.tasks.**applycal**(*cfg*)

The `applycal` task.

cfg A *ApplycalConfig* object.

This function runs the `applycal` task. *For documentation of the general functionality of this task and the parameters it takes*, see the documentation for the *ApplycalConfig* object below. Example:

```
from pwkit.environments.casa import tasks

cfg = tasks.ApplycalConfig()
cfg.vis = 'mydataset.ms'
# ... set other parameters ...
tasks.applycal(cfg)
```

This task may also be invoked through the command line, as `casatask applycal`. Run `casatask applycal --help` to see another version of the documentation provided below.

class pwkit.environments.casa.tasks.**ApplycalConfig**

This is a configuration object for the `applycal` task. This object contains no methods. Rather it contains placeholders (and default values) for parameters that can be passed to *applycal()*.

The following documentation is written to target the **command-line** version of this task, which may be invoked as `casatask applycal`. "Keywords" refer attributes of this structure, "comma-separated lists" should become Python lists, and so on.

Fill in the CORRECTED_DATA column of a visibility dataset using the raw data and a set of calibration tables.

vis= The MS to modify

calwt= Write out calibrated weights as well as calibrated visibilities. Default: false

Pre-applied calibrations

gaintable= Comma-separated list of calibration tables to apply on-the-fly before solving

gainfield= SEMICOLON-separated list of field selections to apply for each gain table. If there are fewer items than there are gaintable items, the list is padded with blank items, implying no selection by field.

interp= COMMA-separated list of interpolation types to use for each gain table. If there are fewer items, the list is padded with 'linear' entries. Allowed values: nearest linear cubic spline

spwmap= SEMICOLON-separated list of spectral window mappings for each existing gain table; each record is a COMMA-separated list of integers. For the i'th spw in the dataset, spwmap[i] specifies the record in the gain table to use. For instance [0, 0, 1, 1] maps four spws in the UV data to just two spectral windows in the preexisting gain table.

opacity= Comma-separated list of opacities in nepers. One for each spw; if there are more spws than entries, the last entry is used for the remaining spws.

gaincurve= Whether to apply VLA-specific built in gain curve correction (default: false)

parang= Whether to apply parallactic angle rotation correction (default: false)

Standard data selection keywords This task can filter input data using any of the following keywords, specified as in the standard CASA interface: `antenna`, `array`, `correlation`, `field`, `intent`, `observation`, `scan`, `spw`, `taql`, `timerange`, `uvrange`.

loglevel= Level of detail from CASA logging system. Default value is `warn`. Allowed values are: `severe`, `warn`, `info`, `info1`, `info2`, `info3`, `info4`, `info5`, `debug1`, `debug2`, `debugging`.

bpplot

pwkit.environments.casa.tasks.**bpplot**(*cfg*)

The `bpplot` task.

cfg A *BpplotConfig* object.

This function runs the `bpplot` task. *For documentation of the general functionality of this task and the parameters it takes*, see the documentation for the *BpplotConfig* object below. Example:

```
from pwkit.environments.casa import tasks

cfg = tasks.BpplotConfig()
cfg.vis = 'mydataset.ms'
# ... set other parameters ...
tasks.bpplot(cfg)
```

This task may also be invoked through the command line, as `casatask bpplot`. Run `casatask bpplot --help` to see another version of the documentation provided below.

class pwkit.environments.casa.tasks.**BpplotConfig**

This is a configuration object for the `bpplot` task. This object contains no methods. Rather it contains placeholders (and default values) for parameters that can be passed to *bpplot()*.

The following documentation is written to target the **command-line** version of this task, which may be invoked as `casatask bpplot`. "Keywords" refer attributes of this structure, "comma-separated lists" should become Python lists, and so on.

Plot a bandpass calibration table. Currently, the supported format is a series of pages showing amplitude and phase against normalized channel number, with each page showing a particular antenna and polarization. Polarizations are always reported as "R" and "L" since the relevant information is not stored within the bandpass data set.

This task also works well to plot frequency-dependent polarimetric leakage calibration tables.

caltable=MS The input calibration Measurement Set

dest=PATH If specified, plots are saved to this file – the format is inferred from the extension, which must allow multiple pages to be saved. If unspecified, the plots are displayed using a Gtk3 backend.

dims=WIDTH,HEIGHT If saving to a file, the dimensions of a each page. These are in points for vector formats (PDF, PS) and pixels for bitmaps (PNG). Defaults to 1000, 600.

margins=TOP,RIGHT,LEFT,BOTTOM If saving to a file, the plot margins in the same units as the dims. The default is 4 on every side.

loglevel= Level of detail from CASA logging system. Default value is `warn`. Allowed values are: `severe`, `warn`, `info`, `info1`, `info2`, `info3`, `info4`, `info5`, `debug1`, `debug2`, `debugging`.

clearcal

pwkit.environments.casa.tasks.**clearcal**(*vis*, *weightonly=False*)

Fill the imaging and calibration columns (`MODEL_DATA`, `CORRECTED_DATA`, `IMAGING_WEIGHT`) of each measurement set with default values, creating the columns if necessary.

vis (string) Path to the input measurement set

weightonly (boolean) If true, just create the `IMAGING_WEIGHT` column; do not fill in the visibility data columns.

If you want to reset calibration models, these days you probably want *delmod_cli()*. If you want to quickly make the columns go away, you probably want *delcal()*.

Example:

```
from pwkit.environments.casa import tasks
tasks.clearcal('myvis.ms')
```

concat

pwkit.environments.casa.tasks.**concat**(*invises*, *outvis*, *timesort=False*)

Concatenate visibility measurement sets.

invises (list of str) Paths to the input measurement sets

outvis (str) Path to the output measurement set.

timesort (boolean) If true, sort the output in time after concatenation.

Example:

```
from pwkit.environments.casa import tasks
tasks.concat(['epoch1.ms', 'epoch2.ms'], 'combined.ms')
```

delcal

pwkit.environments.casa.tasks.**delcal**(*mspath*)

Delete the MODEL_DATA and CORRECTED_DATA columns from a measurement set.

mspath (str) The path to the MS to modify

Example:

```
from pwkit.environments.casa import tasks
tasks.delcal('dataset.ms')
```

delmod

pwkit.environments.casa.tasks.**delmod_cli**(*argv*, *alter_logger=True*)

Command-line access to delmod functionality.

The delmod task deletes "on-the-fly" model information from a Measurement Set. It is so easy to implement that a standalone function is essentially unnecessary. Just write:

```
from pwkit.environments.casa import util
cb = util.tools.calibrater()
cb.open('datasaet.ms', addcorr=False, addmodel=False)
cb.delmod(otf=True, scr=False)
cb.close()
```

If you want to delete the scratch columns, use *delcal()*. If you want to clear the scratch columns, use *clearcal()*.

elplot

pwkit.environments.casa.tasks.**elplot**(*cfg*)

The `elplot` task.

cfg A *ElplotConfig* object.

This function runs the `elplot` task. *For documentation of the general functionality of this task and the parameters it takes*, see the documentation for the *ElplotConfig* object below. Example:

```
from pwkit.environments.casa import tasks

cfg = tasks.ElplotConfig()
cfg.vis = 'mydataset.ms'
# ... set other parameters ...
tasks.elplot(cfg)
```

This task may also be invoked through the command line, as `casatask elplot`. Run `casatask elplot --help` to see another version of the documentation provided below.

class pwkit.environments.casa.tasks.**ElplotConfig**

This is a configuration object for the `elplot` task. This object contains no methods. Rather it contains placeholders (and default values) for parameters that can be passed to *elplot()*.

The following documentation is written to target the **command-line** version of this task, which may be invoked as `casatask elplot`. "Keywords" refer attributes of this structure, "comma-separated lists" should become Python lists, and so on.

Plot elevations of fields observed in a MeasurementSet.

vis=MS The input Measurement Set.

dest=PATH If specified, plots are saved to this file – the format is inferred from the extension, which must allow multiple pages to be saved. If unspecified, the plots are displayed using a Gtk3 backend.

dims=WIDTH,HEIGHT If saving to a file, the dimensions of a each page. These are in points for vector formats(PDF, PS) and pixels for bitmaps(PNG). Defaults to 1000, 600.

margins=TOP,RIGHT,LEFT,BOTTOM If saving to a file, the plot margins in the same units as the dims. The default is 4 on every side.

loglevel= Level of detail from CASA logging system. Default value is `warn`. Allowed values are: `severe`, `warn`, `info`, `info1`, `info2`, `info3`, `info4`, `info5`, `debug1`, `debug2`, `debugging`.

extractbpflags

pwkit.environments.casa.tasks.**extractbpflags**(*calpath*, *deststream*)

Make a flags file out of a bandpass calibration table

calpath (str) The path to the bandpass calibration table

deststream (stream-like object, e.g. an opened file) Where to write the flags data

Below is documentation written for the command-line interface to this functionality:

When CASA encounters flagged channels in bandpass calibration tables, it interpolates over them as best it can – even if interp='<any>,nearest'. This means that if certain channels are unflagged in some target data but entirely flagged in your BP cal, they'll get multiplied by some number that may or may not be reasonable, not flagged. This is scary if, for instance, you're using an automated system to find RFI, or you flag edge channels in some uneven way.

This script writes out a list of flagging commands corresponding to the flagged channels in the bandpass table to ensure that the data without bandpass solutions are flagged.

Note that, because we can't select by antpol, we can't express a situation in which the R and L bandpass solutions have different flags. But in CASA the flags seem to always be the same.

We're assuming that the channelization of the bandpass solution and the data are the same.

flagcmd

pwkit.environments.casa.tasks.**flagcmd**(*cfg*)

The `flagcmd` task.

cfg A *FlagcmdConfig* object.

This function runs the `flagcmd` task. *For documentation of the general functionality of this task and the parameters it takes*, see the documentation for the *FlagcmdConfig* object below. Example:

```
from pwkit.environments.casa import tasks

cfg = tasks.FlagcmdConfig()
cfg.vis = 'mydataset.ms'
# ... set other parameters ...
tasks.flagcmd(cfg)
```

This task may also be invoked through the command line, as `casatask flagcmd`. Run `casatask flagcmd --help` to see another version of the documentation provided below.

class pwkit.environments.casa.tasks.**FlagcmdConfig**

This is a configuration object for the `flagcmd` task. This object contains no methods. Rather it contains placeholders (and default values) for parameters that can be passed to *flagcmd()*.

The following documentation is written to target the **command-line** version of this task, which may be invoked as `casatask flagcmd`. "Keywords" refer attributes of this structure, "comma-separated lists" should become Python lists, and so on.

Flag data using auto-generated lists of flagging commands.

flaglist

pwkit.environments.casa.tasks.**flaglist**(*cfg*)

The `flaglist` task.

cfg A *FlaglistConfig* object.

This function runs the `flaglist` task. *For documentation of the general functionality of this task and the parameters it takes*, see the documentation for the *FlaglistConfig* object below. Example:

```
from pwkit.environments.casa import tasks

cfg = tasks.FlaglistConfig()
cfg.vis = 'mydataset.ms'
# ... set other parameters ...
tasks.flaglist(cfg)
```

This task may also be invoked through the command line, as `casatask flaglist`. Run `casatask flaglist --help` to see another version of the documentation provided below.

class pwkit.environments.casa.tasks.**FlaglistConfig**

This is a configuration object for the `flaglist` task. This object contains no methods. Rather it contains placeholders (and default values) for parameters that can be passed to *flaglist()*.

The following documentation is written to target the **command-line** version of this task, which may be invoked as `casatask flaglist`. "Keywords" refer attributes of this structure, "comma-separated lists" should become Python lists, and so on.

Flag data using a list of flagging commands stored in a text file. This is approximately equivalent to 'flagcmd(vis=, inpfile=, inpmode='list', flagbackup=False)'.

This implementation must emulate the CASA modules that load up the flagging commands and may not be precisely compatible with the CASA version.

flagmanager

pwkit.environments.casa.tasks.**flagmanager_cli**(*argv*, *alter_logger=True*)

Command-line access to `flagmanager` functionality.

The `flagmanager` task manages tables of flags associated with measurement sets. Its features are easy to implement that a standalone library function is essentially unnecessary. See the source code to this function for the tool calls that implement different parts of the `flagmanager` functionality.

flagzeros

pwkit.environments.casa.tasks.**flagzeros**(*cfg*)

The `flagzeros` task.

cfg A *FlagzerosConfig* object.

This function runs the `flagzeros` task. *For documentation of the general functionality of this task and the parameters it takes*, see the documentation for the *FlagzerosConfig* object below. Example:

```
from pwkit.environments.casa import tasks

cfg = tasks.FlagzerosConfig()
cfg.vis = 'mydataset.ms'
# ... set other parameters ...
tasks.flagzeros(cfg)
```

This task may also be invoked through the command line, as `casatask flagzeros`. Run `casatask flagzeros --help` to see another version of the documentation provided below.

class pwkit.environments.casa.tasks.**FlagzerosConfig**

This is a configuration object for the `flagzeros` task. This object contains no methods. Rather it contains placeholders (and default values) for parameters that can be passed to *flagzeros()*.

The following documentation is written to target the **command-line** version of this task, which may be invoked as `casatask flagzeros`. "Keywords" refer attributes of this structure, "comma-separated lists" should become Python lists, and so on.

Flag zero data points in the specified data column.

fluxscale

pwkit.environments.casa.tasks.**fluxscale**(*cfg*)

The `fluxscale` task.

cfg A *FluxscaleConfig* object.

This function runs the `fluxscale` task. *For documentation of the general functionality of this task and the parameters it takes*, see the documentation for the *FluxscaleConfig* object below. Example:

```
from pwkit.environments.casa import tasks

cfg = tasks.FluxscaleConfig()
cfg.vis = 'mydataset.ms'
# ... set other parameters ...
tasks.fluxscale(cfg)
```

This task may also be invoked through the command line, as `casatask fluxscale`. Run `casatask fluxscale --help` to see another version of the documentation provided below.

class pwkit.environments.casa.tasks.**FluxscaleConfig**

This is a configuration object for the `fluxscale` task. This object contains no methods. Rather it contains placeholders (and default values) for parameters that can be passed to *fluxscale()*.

The following documentation is written to target the **command-line** version of this task, which may be invoked as `casatask fluxscale`. "Keywords" refer attributes of this structure, "comma-separated lists" should become Python lists, and so on.

Write a new calibation table determining the fluxes for intermediate calibrators from known reference sources

vis= The visibility dataset.(Shouldn't be needed, but . . .)

caltable= The preexisting calibration table with gains associated with more than one source.

fluxtable= The path of a new calibration table to create

reference= Comma-separated names of sources whose model fluxes are assumed to be well-known.

transfer= Comma-separated names of sources whose fluxes should be computed from the gains.

listfile= If specified, write out flux information to this path.

append= Boolean, default false. If true, append to existing cal table rather than overwriting.

refspwmap= Comma-separated list of integers. If gains are only available for some spws, map from the data to the gains. For instance, refspwmap=1,1,3,3 means that spw 0 will have its flux calculated using the gains for spw 1.

loglevel= Level of detail from CASA logging system. Default value is `warn`. Allowed values are: `severe`, `warn`, `info`, `info1`, `info2`, `info3`, `info4`, `info5`, `debug1`, `debug2`, `debugging`.

ft

pwkit.environments.casa.tasks.**ft**(*cfg*)

The `ft` task.

cfg A *FtConfig* object.

This function runs the `ft` task. *For documentation of the general functionality of this task and the parameters it takes*, see the documentation for the *FtConfig* object below. Example:

```
from pwkit.environments.casa import tasks

cfg = tasks.FtConfig()
cfg.vis = 'mydataset.ms'
# ... set other parameters ...
tasks.ft(cfg)
```

This task may also be invoked through the command line, as `casatask ft`. Run `casatask ft --help` to see another version of the documentation provided below.

class pwkit.environments.casa.tasks.**FtConfig**

This is a configuration object for the `ft` task. This object contains no methods. Rather it contains placeholders (and default values) for parameters that can be passed to *ft()*.

The following documentation is written to target the **command-line** version of this task, which may be invoked as `casatask ft`. "Keywords" refer attributes of this structure, "comma-separated lists" should become Python lists, and so on.

Fill in(or update) the MODEL_DATA column of a Measurement Set with visibilities computed from an image or list of components.

vis= The path to the measurement set

model= Comma-separated list of model images, each giving successive Taylor terms of a spectral model for each source.(It's fine to have just one model, and this will do what you want.) The reference frequency for the Taylor expansion is taken from the first image.

complist= Path to a CASA ComponentList Measurement Set to use in the modeling. I don't know what happens if you specify both this and "model". They might both get applied?

incremental= Bool, default false, meaning that the MODEL_DATA column will be replaced with the new values computed here. If true, the new values are added to whatever's already in MODEL_DATA.

wprojplanes= Optional integer. If provided, W-projection will be used in the computation of the model visibilities, using the specified number of planes. Note that this *does* make a difference even now that this task only embeds information in a MS to enable later on-the-fly computation of the UV model.

usescratch= Foo.

Standard data selection keywords This task can filter input data using any of the following keywords, specified as in the standard CASA interface: `antenna`, `array`, `correlation`, `field`, `intent`, `observation`, `scan`, `spw`, `taql`, `timerange`, `uvrange`.

loglevel= Level of detail from CASA logging system. Default value is `warn`. Allowed values are: `severe`, `warn`, `info`, `info1`, `info2`, `info3`, `info4`, `info5`, `debug1`, `debug2`, `debugging`.

gaincal

pwkit.environments.casa.tasks.**gaincal**(*cfg*)

The `gaincal` task.

cfg A *GaincalConfig* object.

This function runs the `gaincal` task. *For documentation of the general functionality of this task and the parameters it takes*, see the documentation for the *GaincalConfig* object below. Example:

```
from pwkit.environments.casa import tasks

cfg = tasks.GaincalConfig()
```

(continues on next page)

(continued from previous page)

```
cfg.vis = 'mydataset.ms'
# ... set other parameters ...
tasks.gaincal(cfg)
```

This task may also be invoked through the command line, as `casatask gaincal`. Run `casatask gaincal --help` to see another version of the documentation provided below.

class pwkit.environments.casa.tasks.**GaincalConfig**

This is a configuration object for the `gaincal` task. This object contains no methods. Rather it contains placeholders (and default values) for parameters that can be passed to *gaincal()*.

The following documentation is written to target the **command-line** version of this task, which may be invoked as `casatask gaincal`. "Keywords" refer attributes of this structure, "comma-separated lists" should become Python lists, and so on.

Compute calibration parameters from data. Encompasses the functionality of CASA tasks 'gaincal' *and* 'bandpass'.

vis= Input dataset

caltable= Output calibration table (can exist if append=True)

gaintype=

Kind of gain solution: G - gains per poln and spw(default) T - like G, but one value for all polns GSPLINE - like G, with a spline fit. "Experimental" B - bandpass BPOLY - bandpass with polynomial fit. "Somewhat experimental" K - antenna-based delays KCROSS - global cross-hand delay ; use parang=True D - solve for instrumental leakage Df - above with per-channel leakage terms D+QU - solve for leakage and apparent source polarization Df+QU - above with per-channel leakage terms X - solve for absolute position angle phase term Xf - above with per-channel phase terms D+X - D and X. "Not normally done" Df+X - Df and X. Presumably also not normally done. XY+QU - ? XYf+QU - ?

calmode= What parameters to solve for: amplitude("a"), phase("p"), or both ("ap"). Default is "ap". Not used in bandpass solutions.

solint= Solution interval; this is an upper bound, but solutions will be broken across certain boundaries according to "combine". 'inf' - solutions as long as possible(the default) 'int' - one solution per integration (str) - a specific time with units(e.g. '5min') (number) - a specific time in seconds

combine= Comma-separated list of boundary types; solutions will NOT be broken across these boundaries. Types are: `field`, `scan`, `spw`.

refant= Comma-separated list of reference antennas in decreasing priority order.

solnorm= Normalize solution amplitudes to 1 after solving (only applies to gaintypes G, T, B). Also normalizes bandpass phases to zero when solving for bandpasses. Default: false.

append= Whether to append solutions to an existing table. If the table exists and append=False, the table is overwritten! (Default: false)

Pre-applied calibrations

gaintable= Comma-separated list of calibration tables to apply on-the-fly before solving

gainfield= SEMICOLON-separated list of field selections to apply for each gain table. If there are fewer items than there are gaintable items, the list is padded with blank items, implying no selection by field.

interp= COMMA-separated list of interpolation types to use for each gain table. If there are fewer items, the list is padded with 'linear' entries. Allowed values: nearest linear cubic spline

spwmap= SEMICOLON-separated list of spectral window mappings for each existing gain table; each record is a COMMA-separated list of integers. For the i'th spw in the dataset, spwmap[i] specifies the record in the gain table to use. For instance [0, 0, 1, 1] maps four spws in the UV data to just two spectral windows in the preexisting gain table.

opacity= Comma-separated list of opacities in nepers. One for each spw; if there are more spws than entries, the last entry is used for the remaining spws.

gaincurve= Whether to apply VLA-specific built in gain curve correction (default: false)

parang= Whether to apply parallactic angle rotation correction (default: false)

Low-level parameters

minblperant= Number of baselines for each ant in order to solve (default: 4)

minsnr= Min. SNR for acceptable solutions (default: 3.0)

preavg= Interval for pre-averaging data within each solution interval, in seconds. Default is -1, meaning not to pre-average.

smodel=I,Q,U,V Full-stokes point source model to use, if none is embedded in the vis file.

Standard data selection keywords This task can filter input data using any of the following keywords, specified as in the standard CASA interface: `antenna`, `array`, `correlation`, `field`, `intent`, `observation`, `scan`, `spw`, `taql`, `timerange`, `uvrange`.

loglevel= Level of detail from CASA logging system. Default value is `warn`. Allowed values are: `severe`, `warn`, `info`, `info1`, `info2`, `info3`, `info4`, `info5`, `debug1`, `debug2`, `debugging`.

gencal

pwkit.environments.casa.tasks.**gencal**(*cfg*)

The `gencal` task.

cfg A *GencalConfig* object.

This function runs the `gencal` task. *For documentation of the general functionality of this task and the parameters it takes*, see the documentation for the *GencalConfig* object below. Example:

```
from pwkit.environments.casa import tasks

cfg = tasks.GencalConfig()
cfg.vis = 'mydataset.ms'
# ... set other parameters ...
tasks.gencal(cfg)
```

This task may also be invoked through the command line, as `casatask gencal`. Run `casatask gencal --help` to see another version of the documentation provided below.

class pwkit.environments.casa.tasks.**GencalConfig**

This is a configuration object for the `gencal` task. This object contains no methods. Rather it contains placeholders (and default values) for parameters that can be passed to *gencal()*.

The following documentation is written to target the **command-line** version of this task, which may be invoked as `casatask gencal`. "Keywords" refer attributes of this structure, "comma-separated lists" should become Python lists, and so on.

Generate certain calibration tables that don't need to be solved for from the actual data.

If you want to generate antenna position corrections for Jansky VLA data, you can just specify *caltype=antpos* and leave off the "parameter" keyword. This will cause the task will talk to an NRAO server and automatically download the correct position corrections. Other telescopes do not support this functionality, but if you can obtain the position corrections, you can use the "antenna" and "parameter" keywords to build the desired calibration table manually.

vis= Input dataset

caltable= Output calibration table (appended to if preexisting)

caltype= The kind of table to generate: amp - generic amplitude correction; needs parameter(s) ph - generic phase correction; needs parameter(s) sbd - single-band delay: phase slope for each SPW; needs parameter(s) mbd - multi-band delay: phase slope for all SPWs; needs parameter(s) antpos - antenna position corrections in ITRF; what you want; accepts parameter(s) antposvla - antenna position corrections in VLA frame; **not what you want**; accepts parameter(s) tsys - tsys from ALMA syscal table swpow - EVLA switched-power and requantizer gains("experimental") opac - tropospheric opacity; needs parameter gc - (E)VLA elevation-dependent gain curve eff - (E)VLA antenna efficiency correction gceff - combination of "gc" and "eff" rq - EVLA requantizer gains; not what you want swp/rq - EVLA switched-power gains divided by "rq"; not what you want

parameter= Custom parameters for various caltypes. Dimensionality depends on selections applied. amp - gain; dimensionless ph - phase; degrees sbd - delay; nanosec mbd - delay; nanosec antpos - position offsets; ITRF meters(or look up automatically for EVLA if unspecified) antposvla - position offsets; meters in VLA reference frame opac - opacity; dimensionless(nepers?)

antenna= Selection keyword, governing which solutions to generate and controlling shape of "parameter" keyword.

pol= Analogous to "antenna"

spw= Analogous to "antenna"

loglevel= Level of detail from CASA logging system. Default value is `warn`. Allowed values are: `severe`, `warn`, `info`, `info1`, `info2`, `info3`, `info4`, `info5`, `debug1`, `debug2`, `debugging`.

getopacities

`pwkit.environments.casa.tasks.`**`getopacities`**(*ms*, *plotdest*)

gpdetrend

`pwkit.environments.casa.tasks.`**`gpdetrend`**(*cfg*)

The `gpdetrend` task.

cfg A *`GpdetrendConfig`* object.

This function runs the `gpdetrend` task. *For documentation of the general functionality of this task and the parameters it takes*, see the documentation for the *`GpdetrendConfig`* object below. Example:

```
from pwkit.environments.casa import tasks

cfg = tasks.GpdetrendConfig()
cfg.vis = 'mydataset.ms'
# ... set other parameters ...
tasks.gpdetrend(cfg)
```

This task may also be invoked through the command line, as `casatask gpdetrend`. Run `casatask gpdetrend --help` to see another version of the documentation provided below.

class pwkit.environments.casa.tasks.**GpdetrendConfig**

This is a configuration object for the gpdetrend task. This object contains no methods. Rather it contains placeholders (and default values) for parameters that can be passed to *gpdetrend()*.

The following documentation is written to target the **command-line** version of this task, which may be invoked as casatask gpdetrend. "Keywords" refer attributes of this structure, "comma-separated lists" should become Python lists, and so on.

Remove long-term phase trends from a complex-gain calibration table. For each antenna/spw/pol, the complex gains are divided into separate chunks(e.g., the intention is for one chunk for each visit to the complex-gain calibrator). The mean phase within each chunk is divided out. The effect is to remove long-term phase trends from the calibration table, but preserve short-term ones.

caltable=MS The input calibration Measurement Set

maxtimegap=int Measured in minutes. Gaps between solutions of this duration or longer will lead to a new segment being considered. Default is four times the smallest time gap seen in the data set.

loglevel= Level of detail from CASA logging system. Default value is warn. Allowed values are: severe, warn, info, info1, info2, info3, info4, info5, debug1, debug2, debugging.

gpplot

pwkit.environments.casa.tasks.**gpplot**(*cfg*)

The gpplot task.

cfg A *GpplotConfig* object.

This function runs the gpplot task. *For documentation of the general functionality of this task and the parameters it takes*, see the documentation for the *GpplotConfig* object below. Example:

```
from pwkit.environments.casa import tasks

cfg = tasks.GpplotConfig()
cfg.vis = 'mydataset.ms'
# ... set other parameters ...
tasks.gpplot(cfg)
```

This task may also be invoked through the command line, as casatask gpplot. Run casatask gpplot --help to see another version of the documentation provided below.

class pwkit.environments.casa.tasks.**GpplotConfig**

This is a configuration object for the gpplot task. This object contains no methods. Rather it contains placeholders (and default values) for parameters that can be passed to *gpplot()*.

The following documentation is written to target the **command-line** version of this task, which may be invoked as casatask gpplot. "Keywords" refer attributes of this structure, "comma-separated lists" should become Python lists, and so on.

Plot a gain calibration table. Currently, the supported format is a series of pages showing amplitude and phase against time, with each page showing a particular antenna and polarization. Polarizations are always reported as "R" and "L" since the relevant information is not stored within the bandpass data set.

caltable=MS The input calibration Measurement Set

dest=PATH If specified, plots are saved to this file – the format is inferred from the extension, which must allow multiple pages to be saved. If unspecified, the plots are displayed using a Gtk3 backend.

dims=WIDTH,HEIGHT If saving to a file, the dimensions of a each page. These are in points for vector formats(PDF, PS) and pixels for bitmaps(PNG). Defaults to 1000, 600.

margins=TOP,RIGHT,LEFT,BOTTOM If saving to a file, the plot margins in the same units as the dims. The default is 4 on every side.

maxtimegap=10 Solutions separated by more than this number of minutes will be drawn with separate lines for clarity.

mjdrange=START,STOP If specified, gain solutions outside of the MJDs STOP and START will be ignored.

phaseonly=false If True, plot only phases, and not amplitudes.

loglevel= Level of detail from CASA logging system. Default value is `warn`. Allowed values are: `severe`, `warn`, `info`, `info1`, `info2`, `info3`, `info4`, `info5`, `debug1`, `debug2`, `debugging`.

image2fits

pwkit.environments.casa.tasks.**image2fits**(*mspath*, *fitspath*, *velocity=False*, *optical=False*, *bitpix=-32*, *minpix=0*, *maxpix=-1*, *overwrite=False*, *dropstokes=False*, *stokeslast=True*, *history=True*, ***kwargs*)

Convert an image in MS format to FITS format.

mspath (str) The path to the input MS.

fitspath (str) The path to the output FITS file.

velocity (boolean) (To be documented.)

optical (boolean) (To be documented.)

bitpix (integer) (To be documented.)

minpix (integer) (To be documented.)

maxpix (integer) (To be documented.)

overwrite (boolean) Whether the task is allowed to overwrite an existing destination file.

dropstokes (boolean) Whether the "Stokes" (polarization) axis of the image should be dropped.

stokeslast (boolean) Whether the "Stokes" (polarization) axis of the image should be placed as the last (innermost?) axis of the image cube.

history (boolean) (To be documented.)

****kwargs** Forwarded on to the `tofits` function of the CASA `image` tool.

importalma

pwkit.environments.casa.tasks.**importalma**(*asdm*, *ms*)

Convert an ALMA low-level ASDM dataset to Measurement Set format.

asdm (str) The path to the input ASDM dataset.

ms (str) The path to the output MS dataset.

This implementation automatically infers the value of the "tbuff" parameter.

Example:

```
from pwkit.environments.casa import tasks
tasks.importalma('myalma.asdm', 'myalma.ms')
```

importevla

pwkit.environments.casa.tasks.**importevla**(*asdm*, *ms*)

Convert an EVLA low-level SDM dataset to Measurement Set format.

asdm (str) The path to the input ASDM dataset.

ms (str) The path to the output MS dataset.

This implementation automatically infers the value of the "tbuff" parameter.

Example:

```
from pwkit.environments.casa import tasks
tasks.importevla('myvla.sdm', 'myvla.ms')
```

listobs

pwkit.environments.casa.tasks.**listobs**(*vis*)

Textually describe the contents of a measurement set.

vis (str) The path to the dataset.

Returns A generator of lines of human-readable output

Errors can only be detected by looking at the output. Example:

```
from pwkit.environments.casa import tasks

for line in tasks.listobs('mydataset.ms'):
  print(line)
```

listsdm

pwkit.environments.casa.tasks.**listsdm**(*sdm*, *file=None*)

Generate a standard "listsdm" listing of(A)SDM dataset contents.

sdm (str) The path to the (A)SDM dataset to parse

file (stream-like object, such as an opened file) Where to print the human-readable listing. If unspecified, results go to `sys.stdout`.

Returns A dictionary of information about the dataset. Contents not yet documented.

Example:

```
from pwkit.environments.casa import tasks
tasks.listsdm('myalmaa.asdm')
```

This code based on CASA's *task_listsdm.py*, with this version info:

```
# v1.0: 2010.12.07, M. Krauss
# v1.1: 2011.02.23, M. Krauss: added functionality for ALMA data
#
# Original code based on readscans.py, courtesy S. Meyers
```

mfsclean

pwkit.environments.casa.tasks.**mfsclean**(*cfg*)

The mfsclean task.

cfg A *MfscleanConfig* object.

This function runs the mfsclean task. *For documentation of the general functionality of this task and the parameters it takes*, see the documentation for the *MfscleanConfig* object below. Example:

```
from pwkit.environments.casa import tasks

cfg = tasks.MfscleanConfig()
cfg.vis = 'mydataset.ms'
# ... set other parameters ...
tasks.mfsclean(cfg)
```

This task may also be invoked through the command line, as `casatask mfsclean`. Run `casatask mfsclean --help` to see another version of the documentation provided below.

class pwkit.environments.casa.tasks.**MfscleanConfig**

This is a configuration object for the mfsclean task. This object contains no methods. Rather it contains placeholders (and default values) for parameters that can be passed to *mfsclean()*.

The following documentation is written to target the **command-line** version of this task, which may be invoked as `casatask mfsclean`. "Keywords" refer attributes of this structure, "comma-separated lists" should become Python lists, and so on.

Drive the CASA imager with a very restricted set of options.

For W-projection, set ftmachine='wproject' and wprojplanes=64(or so).

vis= Input visibility MS

imbase= Base name of output files. We create files named "imbaseEXT" where EXT is all of "mask", "modelTT", "imageTT", "residualTT", and "psfTT", and TT is empty if nterms = 1, and "ttN." otherwise.

cell = 1 [arcsec] ftmachine = 'ft' or 'wproject' gain = 0.1 imsize = 256,256 mask = (blank) or path of CASA-format region text file niter = 500 nterms = 1 phasecenter = (blank) or 'J2000 12h34m56.7 -12d34m56.7' reffreq = 0 [GHz] stokes = I threshold = 0 [mJy] weighting = 'briggs'(robust=0.5) or 'natural' wprojplanes = 1

Standard data selection keywords This task can filter input data using any of the following keywords, specified as in the standard CASA interface: `antenna`, `array`, `correlation`, `field`, `intent`, `observation`, `scan`, `spw`, `taql`, `timerange`, `uvrange`.

loglevel= Level of detail from CASA logging system. Default value is `warn`. Allowed values are: `severe`, `warn`, `info`, `info1`, `info2`, `info3`, `info4`, `info5`, `debug1`, `debug2`, `debugging`.

mjd2date

pwkit.environments.casa.tasks.**mjd2date**(*mjd*, *precision=3*)

Convert an MJD to a data string in the format used by CASA.

mjd (numeric) An MJD value in the UTC timescale.

precision (integer, default 3) The number of digits of decimal precision in the seconds portion of the returned string

Returns A string representing the input argument in CASA format: `YYYY/MM/DD/HH:MM:SS.SSS`.

Example:

```
from pwkit.environment.casa import tasks
print(tasks.mjd2date(55555))
# yields '2010/12/25/00:00:00.000'
```

mstransform

pwkit.environments.casa.tasks.**mstransform**(*cfg*)

The `mstransform` task.

cfg A *MstransformConfig* object.

This function runs the `mstransform` task. *For documentation of the general functionality of this task and the parameters it takes*, see the documentation for the *MstransformConfig* object below. Example:

```
from pwkit.environments.casa import tasks

cfg = tasks.MstransformConfig()
cfg.vis = 'mydataset.ms'
# ... set other parameters ...
tasks.mstransform(cfg)
```

This task may also be invoked through the command line, as `casatask mstransform`. Run `casatask mstransform --help` to see another version of the documentation provided below.

class pwkit.environments.casa.tasks.**MstransformConfig**

This is a configuration object for the `mstransform` task. This object contains no methods. Rather it contains placeholders (and default values) for parameters that can be passed to *mstransform()*.

The following documentation is written to target the **command-line** version of this task, which may be invoked as `casatask mstransform`. "Keywords" refer attributes of this structure, "comma-separated lists" should become Python lists, and so on.

vis= Input visibility MS

out= Output visibility MS

datacolumn=corrected The data column on which to operate. Comma-separated list of: `data`, `model`, `corrected`, `float_data`, `lag_data`, `all`

realmodelcol=False If true, turn a virtual model column into a real one.

keepflags=True If false, discard completely-flagged rows.

usewtspectrum=False If true, fill in a WEIGHT_SPECTRUM column in the output data set.

combinespws=False If true, combine spectral windows

chanaverage=False If true, average the data in frequency. NOT WIRED UP.

hanning=False If true, Hanning smooth the data spectrally to remove Gibbs ringing.

regridms=False If true, put the data on a new spectral window structure or reference frame.

timebin=<seconds> If specified, time-average the visibilities with the specified binning.

timespan=<undefined> Allow averaging to span over potential discontinuities in the data set. Comma-separated list of options; allowed values are: `scan`, `state`

Standard data selection keywords This task can filter input data using any of the following keywords, specified as in the standard CASA interface: `antenna`, `array`, `correlation`, `field`, `intent`, `observation`, `scan`, `spw`, `taql`, `timerange`, `uvrange`.

loglevel= Level of detail from CASA logging system. Default value is `warn`. Allowed values are: `severe`, `warn`, `info`, `info1`, `info2`, `info3`, `info4`, `info5`, `debug1`, `debug2`, `debugging`.

plotants

`pwkit.environments.casa.tasks.`**`plotants`**(*vis*, *figfile*)

Plot the physical layout of the antennas described in the MS.

vis (str) Path to the input dataset

figfile (str) Path to the output image file.

The output image format will be inferred from the extension of *figfile*. Example:

```
from pwkit.environments.casa import tasks
tasks.plotants('dataset.ms', 'antennas.png')
```

plotcal

`pwkit.environments.casa.tasks.`**`plotcal`**(*cfg*)

The `plotcal` task.

cfg A *`PlotcalConfig`* object.

This function runs the `plotcal` task. *For documentation of the general functionality of this task and the parameters it takes*, see the documentation for the *`PlotcalConfig`* object below. Example:

```
from pwkit.environments.casa import tasks

cfg = tasks.PlotcalConfig()
cfg.vis = 'mydataset.ms'
# ... set other parameters ...
tasks.plotcal(cfg)
```

This task may also be invoked through the command line, as `casatask plotcal`. Run `casatask plotcal --help` to see another version of the documentation provided below.

`class` `pwkit.environments.casa.tasks.`**`PlotcalConfig`**

This is a configuration object for the `plotcal` task. This object contains no methods. Rather it contains placeholders (and default values) for parameters that can be passed to *`plotcal()`*.

The following documentation is written to target the **command-line** version of this task, which may be invoked as `casatask plotcal`. "Keywords" refer attributes of this structure, "comma-separated lists" should become Python lists, and so on.

Plot values from a calibration dataset in any of a variety of ways.

caltable= The calibration MS to plot

xaxis= amp antenna chan freq imag phase real snr time

yaxis= amp antenna imag phase real snr

iteration= antenna field spw time

Supported data selection keywords

Limited data selection is supported. Allowed keywords are `antenna`, `field`, `poln`, `spw`, and `timerange`. The `poln` keyword may take on the values `RL`, `R`, `L`, `XY`, `X`, `Y`, and `/`.

Plot appearance options

To be documented. These keywords control the plot appearance: `plotsymbol`, `plotcolor`, `fontsize`, `figfile`.

loglevel= Level of detail from CASA logging system. Default value is `warn`. Allowed values are: `severe`, `warn`, `info`, `info1`, `info2`, `info3`, `info4`, `info5`, `debug1`, `debug2`, `debugging`.

setjy

pwkit.environments.casa.tasks.**setjy**(*cfg*)

The `setjy` task.

cfg A *SetjyConfig* object.

This function runs the `setjy` task. *For documentation of the general functionality of this task and the parameters it takes*, see the documentation for the *SetjyConfig* object below. Example:

```
from pwkit.environments.casa import tasks

cfg = tasks.SetjyConfig()
cfg.vis = 'mydataset.ms'
# ... set other parameters ...
tasks.setjy(cfg)
```

This task may also be invoked through the command line, as `casatask setjy`. Run `casatask setjy --help` to see another version of the documentation provided below.

class pwkit.environments.casa.tasks.**SetjyConfig**

This is a configuration object for the `setjy` task. This object contains no methods. Rather it contains placeholders (and default values) for parameters that can be passed to *setjy()*.

The following documentation is written to target the **command-line** version of this task, which may be invoked as `casatask setjy`. "Keywords" refer attributes of this structure, "comma-separated lists" should become Python lists, and so on.

Insert model data into a measurement set. We force usescratch=False and scalebychan=True. You probably want to specify "field".

fluxdensity= Up to four comma-separated numbers giving Stokes IQUV intensities in Jy. Default values are [-1, 0, 0, 0]. If the Stokes I intensity is negative (i.e., the default), a "sensible default" will be used: detailed spectral models if the source is known (see "standard"), or 1 otherwise. If it is zero and "modimage" is used, the flux density of the model image is used. The built-in standards do NOT have polarimetric information, so for pol cal you do need to manually specify the flux density information – or see the program "casatask polmodel".

modimage= An image to use as the basis for the source's spatial structure and, potentialy, flux density (if fluxdensity=0). Only usable for Stokes I. If the verbatim value of "modimage" can't be opened as a path, it is assumed to be relative to the CASA data directory; a typical value might be "nrao/VLA/CalModels/3C286_C.im".

spindex= If using `fluxdensity`, these specify the spectral dependence of the values, such that `S = fluxdensity * (freq/reffreq)**spindex`. Reffreq is in GHz. Default values are 0 and 1, giving no spectral dependence.

reffreq= See `spindex`.

standard='Perley-Butler 2013' Acceptable values are: Baars, Perley 90, Perley-Taylor 95, Perley-Taylor 99, Perley-Butler 2010, Perley-Butler 2013. You can specify the solar-system standard "Butler-JPL-Horizons 2012", but doing so farms out the work to a stock CASA installation.

Supported data selection keywords

Only a subset of the standard data selection keywords are supported: `field`, `observation`, `scan`, `spw`, `timerange`..

loglevel= Level of detail from CASA logging system. Default value is `warn`. Allowed values are: `severe`, `warn`, `info`, `info1`, `info2`, `info3`, `info4`, `info5`, `debug1`, `debug2`, `debugging`.

split

`pwkit.environments.casa.tasks.`**`split`**`(`*`cfg`*`)`

The `split` task.

cfg A *`SplitConfig`* object.

This function runs the `split` task. *For documentation of the general functionality of this task and the parameters it takes*, see the documentation for the *`SplitConfig`* object below. Example:

```
from pwkit.environments.casa import tasks

cfg = tasks.SplitConfig()
cfg.vis = 'mydataset.ms'
# ... set other parameters ...
tasks.split(cfg)
```

This task may also be invoked through the command line, as `casatask split`. Run `casatask split --help` to see another version of the documentation provided below.

`class` `pwkit.environments.casa.tasks.`**`SplitConfig`**

This is a configuration object for the `split` task. This object contains no methods. Rather it contains placeholders (and default values) for parameters that can be passed to *`split()`*.

The following documentation is written to target the **command-line** version of this task, which may be invoked as `casatask split`. "Keywords" refer attributes of this structure, "comma-separated lists" should become Python lists, and so on.

timebin= Time-average data into bins of "timebin" seconds; defaults to no averaging

step= Frequency-average data in bins of "step" channels; defaults to no averaging

col=all Extract the column "col" as the DATA column. If "all", copy all available columns without renaming. Possible values: `all`, `DATA`, `MODEL_DATA`, `CORRECTED_DATA`, `FLOAT_DATA`, `LAG_DATA`.

combine=[col1,col2,...] When time-averaging, don't start a new bin when the specified columns change. Acceptable column names: `scan`, `state`.

Standard data selection keywords This task can filter input data using any of the following keywords, specified as in the standard CASA interface: `antenna`, `array`, `correlation`, `field`, `intent`, `observation`, `scan`, `spw`, `taql`, `timerange`, `uvrange`.

loglevel= Level of detail from CASA logging system. Default value is `warn`. Allowed values are: `severe`, `warn`, `info`, `info1`, `info2`, `info3`, `info4`, `info5`, `debug1`, `debug2`, `debugging`.

tsysplot

`pwkit.environments.casa.tasks.`**`tsysplot`**`(`*`cfg`*`)`

The `tsysplot` task.

cfg A *`TsysplotConfig`* object.

This function runs the `tsysplot` task. *For documentation of the general functionality of this task and the parameters it takes*, see the documentation for the *TsysplotConfig* object below. Example:

```
from pwkit.environments.casa import tasks

cfg = tasks.TsysplotConfig()
cfg.vis = 'mydataset.ms'
# ... set other parameters ...
tasks.tsysplot(cfg)
```

This task may also be invoked through the command line, as `casatask tsysplot`. Run `casatask tsysplot --help` to see another version of the documentation provided below.

class pwkit.environments.casa.tasks.**TsysplotConfig**

This is a configuration object for the `tsysplot` task. This object contains no methods. Rather it contains placeholders (and default values) for parameters that can be passed to *tsysplot()*.

The following documentation is written to target the **command-line** version of this task, which may be invoked as `casatask tsysplot`. "Keywords" refer attributes of this structure, "comma-separated lists" should become Python lists, and so on.

Plot a system temperature(Tsys) calibration table.

caltable=MS The input calibration Measurement Set

dest=PATH If specified, plots are saved to this file – the format is inferred from the extension, which must allow multiple pages to be saved. If unspecified, the plots are displayed using a Gtk3 backend.

dims=WIDTH,HEIGHT If saving to a file, the dimensions of a each page. These are in points for vector formats(PDF, PS) and pixels for bitmaps(PNG). Defaults to 1000, 600.

margins=TOP,RIGHT,LEFT,BOTTOM If saving to a file, the plot margins in the same units as the dims. The default is 4 on every side.

loglevel= Level of detail from CASA logging system. Default value is `warn`. Allowed values are: `severe`, `warn`, `info`, `info1`, `info2`, `info3`, `info4`, `info5`, `debug1`, `debug2`, `debugging`.

uvsub

pwkit.environments.casa.tasks.**uvsub**(*cfg*)

The `uvsub` task.

cfg A *UvsubConfig* object.

This function runs the `uvsub` task. *For documentation of the general functionality of this task and the parameters it takes*, see the documentation for the *UvsubConfig* object below. Example:

```
from pwkit.environments.casa import tasks

cfg = tasks.UvsubConfig()
cfg.vis = 'mydataset.ms'
# ... set other parameters ...
tasks.uvsub(cfg)
```

This task may also be invoked through the command line, as `casatask uvsub`. Run `casatask uvsub --help` to see another version of the documentation provided below.

class pwkit.environments.casa.tasks.**UvsubConfig**

This is a configuration object for the `uvsub` task. This object contains no methods. Rather it contains placeholders (and default values) for parameters that can be passed to *uvsub()*.

The following documentation is written to target the **command-line** version of this task, which may be invoked as `casatask uvsub`. "Keywords" refer attributes of this structure, "comma-separated lists" should become Python lists, and so on.

Set the CORRECTED_DATA column to the difference of DATA and MODEL_DATA.

vis= The input data set.

reverse= Boolean, default false, which means to set CORRECTED = DATA - MODEL. If true, CORRECTED = DATA + MODEL.

Standard data selection keywords This task can filter input data using any of the following keywords, specified as in the standard CASA interface: `antenna`, `array`, `correlation`, `field`, `intent`, `observation`, `scan`, `spw`, `taql`, `timerange`, `uvrange`.

loglevel= Level of detail from CASA logging system. Default value is `warn`. Allowed values are: `severe`, `warn`, `info`, `info1`, `info2`, `info3`, `info4`, `info5`, `debug1`, `debug2`, `debugging`.

xyphplot

`pwkit.environments.casa.tasks.`**`xyphplot`**`(`*`cfg`*`)`

The `xyphplot` task.

cfg A *`XyphplotConfig`* object.

This function runs the `xyphplot` task. *For documentation of the general functionality of this task and the parameters it takes*, see the documentation for the *`XyphplotConfig`* object below. Example:

```
from pwkit.environments.casa import tasks

cfg = tasks.XyphplotConfig()
cfg.vis = 'mydataset.ms'
# ... set other parameters ...
tasks.xyphplot(cfg)
```

This task may also be invoked through the command line, as `casatask xyphplot`. Run `casatask xyphplot --help` to see another version of the documentation provided below.

`class` `pwkit.environments.casa.tasks.`**`XyphplotConfig`**

This is a configuration object for the `xyphplot` task. This object contains no methods. Rather it contains placeholders (and default values) for parameters that can be passed to *`xyphplot()`*.

The following documentation is written to target the **command-line** version of this task, which may be invoked as `casatask xyphplot`. "Keywords" refer attributes of this structure, "comma-separated lists" should become Python lists, and so on.

Plot a frequency-dependent X/Y phase calibration table.

caltable=MS The input calibration Measurement Set

dest=PATH If specified, plots are saved to this file – the format is inferred from the extension, which must allow multiple pages to be saved. If unspecified, the plots are displayed using a Gtk3 backend.

dims=WIDTH,HEIGHT If saving to a file, the dimensions of a each page. These are in points for vector formats(PDF, PS) and pixels for bitmaps(PNG). Defaults to 1000, 600.

margins=TOP,RIGHT,LEFT,BOTTOM If saving to a file, the plot margins in the same units as the dims. The default is 4 on every side.

loglevel= Level of detail from CASA logging system. Default value is `warn`. Allowed values are: `severe`, `warn`, `info`, `info1`, `info2`, `info3`, `info4`, `info5`, `debug1`, `debug2`, `debugging`.

CASA Tools and Utilities (`pwkit.environments.casa.util`)

This module provides low-level tools and utilities for interacting with the `casac` module provided by CASA.

This module provides:

- *The tools object*
- *Useful Constants*
- *Useful Functions*

The `tools` object

`pwkit.environments.casa.util.`**`tools`**

This object is a singleton instance of a hidden class that assists in the creation of CASA "tools" objects. For instance, you can create and use a standard CASA "tool" for reading and manipulating data tables with code like this:

```
from pwkit.environments.casa import util
tb = util.tools.table()
tb.open('myfile.ms')
tb.close()
```

Documentation for the individual CASA "tools" is beyond the scope of *pwkit* ... although maybe it will be added, since the documentation provided by CASA is pretty weak.

Here's a list of CASA tool names. They can all be created in the same way: by calling the function `tools.<toolname>()`. This will work even for any tools not appearing in this list, so long as they're provided by the underlying CASA libraries:

- agentflagger
- atmosphere
- calanalysis
- calibrater
- calplot
- componentlist
- coordsys
- deconvolver
- fitter
- flagger
- functional
- image
- imagepol
- imager
- logsink
- measures
- msmetadata

- ms
- msplot
- mstransformer
- plotms
- regionmanager
- simulator
- spectralline
- quanta
- table
- tableplot
- utils
- vlafiller
- vpmanager

Useful Constants

The following useful constants are provided:

pwkit.environments.casa.util.**INVERSE_C_SM**
: The inverse of the speed of light, *c*, measured in seconds per meter. This is useful for converting between wavelength and light travel time.

pwkit.environments.casa.util.**INVERSE_C_NSM**
: The inverse of the speed of light, *c*, measured in nanoseconds per meter. This is useful for converting between wavelength and light travel time.

pwkit.environments.casa.util.**pol_names**
: A dictionary mapping CASA polarization codes to their textual names. For instance, `pol_names[9]` is `"XX"` and `pol_names[7]` is `"LR"`.

pwkit.environments.casa.util.**pol_to_miriad**
: A dictionary mapping CASA polarization codes to MIRIAD polarization codes, such that:

```
miriad_pol_code = pol_to_miriad[casa_pol_code]
```

CASA defines many more polarization codes than MIRIAD, although it is unclear whether CASA's additional ones are ever used in practice. Trying to map a code without a MIRIAD equivalent will result in a `KeyError` as you might expect.

pwkit.environments.casa.util.**pol_is_intensity**
: A dictionary mapping CASA polarization codes to booleans indicating whether the polarization is of "intensity" type. "Intensity-type" polarizations cannot have negative values; they are II, RR, LL, XX, YY, PP, and QQ.

pwkit.environments.casa.util.**msselect_keys**
: A `set` of the keys supported by the CASA "MS-select" subsystem.

Useful Functions

sanitize_unicode(item)	Safely pass string values to the CASA tools.
datadir(*subdirs)	Get a path within the CASA data directory.
logger([filter])	Set up CASA to write log messages to standard output.
forkandlog(function[, filter, debug])	Fork a child process and read its CASA log output.

pwkit.environments.casa.util.**sanitize_unicode**(*item*)

Safely pass string values to the CASA tools.

item A value to be passed to a CASA tool.

In Python 2, the bindings to CASA tasks expect to receive all string values as binary data (`str`) and not Unicode. But *pwkit* often uses the `from __future__ import unicode_literals` statement to prepare for Python 3 compatibility, and other Python modules are getting better about using Unicode consistently, so more and more module code ends up using Unicode strings in cases where they might get exposed to CASA. Doing so will lead to errors.

This helper function converts Unicode into UTF-8 encoded bytes for arguments that you might pass to a CASA tool. It will leave non-strings unchanged and recursively transform collections, so you can safely use it just about anywhere.

I usually import this as just `b` and write `tool.method(b(arg))`, in analogy with the `b''` byte string syntax. This leads to code such as:

```
from pwkit.environments.casa.util import tools, sanitize_unicode as b

tb = tools.table()
path = u'data.ms'
tb.open(path) # => raises exception
tb.open(b(path)) # => works
```

pwkit.environments.casa.util.**datadir**(**subdirs*)

Get a path within the CASA data directory.

subdirs Extra elements to append to the returned path.

This function locates the directory where CASA resource data files (tables of time offsets, calibrator models, etc.) are stored. If called with no arguments, it simply returns that path. If arguments are provided, they are appended to the returned path using `os.path.join()`, making it easy to construct the names of specific data files. For instance:

```
from pwkit.environments.casa import util

cal_image_path = util.datadir('nrao', 'VLA', 'CalModels', '3C286_C.im')
tb = util.tools.image()
tb.open(cal_image_path)
```

pwkit.environments.casa.util.**logger**(*filter='WARN'*)

Set up CASA to write log messages to standard output.

filter The log level filter: less urgent messages will not be shown. Valid values are strings: "DEBUG1", "INFO5", ... "INFO1", "INFO", "WARN", "SEVERE".

This function creates and returns a CASA "log sink" object that is configured to write to standard output. The default CASA implementation would *always* create a file named `casapy.log` in the current directory; this function safely prevents such a file from being left around. This is particularly important if you don't have write permissions to the current directory.

pwkit.environments.casa.util.**forkandlog**(*function*, *filter='INFO5'*, *debug=False*)

Fork a child process and read its CASA log output.

function A function to run in the child process

filter The CASA log level filter to apply in the child process: less urgent messages will not be shown. Valid values are strings: "DEBUG1", "INFO5", ... "INFO1", "INFO", "WARN", "SEVERE".

debug If true, the standard output and error of the child process are *not* redirected to /dev/null.

Some CASA tools produce important results that are *only* provided via log messages. This is a problem for automation, since there's no way for Python code to intercept those log messages and extract the results of interest. This function provides a framework for working around this limitation: by forking a child process and sending its log output to a pipe, the parent process can capture the log messages.

This function is a generator. It yields lines from the child process' CASA log output.

Because the child process is a fork of the parent, it inherits a complete clone of the parent's state at the time of forking. That means that the *function* argument you pass it can do just about anything you'd do in a regular program.

The child process' standard output and error streams are redirected to `/dev/null` unless the *debug* argument is true. Note that the CASA log output is redirected to a pipe that is neither of these streams. So, if the function raises an unhandled Python exception, the Python traceback will not pollute the CASA log output. But, by the same token, the calling program will not be able to detect that the exception occurred except by its impact on the expected log output.

CASA: DFT Dynamic Spectra (`pwkit.environments.casa.dftdynspec`)

This module provides code to extract dynamic spectra from CASA Measurement Sets. CASA doesn't have a task that does this.

The function `dftdynspec()` computes the dynamic spectrum of a point source using a discrete Fourier transform of its visibilities. The function `dftdynspec_cli()` provides a hook to launch the computation from a command-line program.

You can launch a computation from the command line using the command `casatask dftdynspec`.

Due to limitations in the documentation system we're using, the options to the dynamic spectrum computation are not documented here. You can read about them by running `casatask dftdynspec --help`.

The `Loader` class

Unlike the other DFT tasks, `dftdynspec()` produces output that is not easily represented as a table. It is saved to disk as a set of Numpy arrays. The *Loader* class provides a convenient mechanism for loading an output data set.

To load and manipulate data, create a *Loader* instance and then access the various arrays described below:

```
from pwkit.environments.casa.dftdynspec import Loader
path = 'mydataset.npy' # this gets customized
ds = Loader(path)
print('Maximum real part:', ds.reals.max())
```

class pwkit.environments.casa.dftdynspec.**Loader**(*path*)

Read in a dynamic-spectrum file produced by the *dftdynspec* task.

Constructor arguments

path The path of the file to read.

Members

counts = None
A 2D array recording the number of visibilities that went into each average. Shape is *(mjds.size, freqs.size)*.

freqs = None
A 1D sorted array of the frequencies of the data samples, measured in GHz.

imags = None
A 2D array of the imaginary parts of the averaged visibilities. Shape is *(mjds.size, freqs.size)*.

mjds = None
A 1D sorted array of the MJDs of the data samples.

n_freqs
The size of the frequency axis of the data arrays; an integer.

n_mjds
The size of the MJD axis of the data arrays; an integer.

reals = None
A 2D array of the real parts of the averaged visibilities. Shape is *(mjds.size, freqs.size)*.

u_imags = None
A 2D array of the estimated uncertainties on the imaginary parts of the averaged visibilities. Shape is *(mjds.size, freqs.size)*.

u_reals = None
A 2D array of the estimated uncertainties on the real parts of the averaged visibilities. Shape is *(mjds.size, freqs.size)*.

Compact-source photometry with discrete Fourier transformations (`pwkit.environments.casa.dftphotom`)

This module implements an algorithm to compute light curves for point sources in interferometric visibility data. CASA doesn't have a task to do this.

The algorithm is accessible from the command line as `casatask dftphotom`, but it can also be invoked from within Python. For help on usage from the command line, run `casatask dftphotom --help`. The command's help text will also describe some of the parameters below in more detail than is currently found here.

Usage from Within Python

Basic usage from within Python looks like this:

```
from pwkit.astutil import parsehours, parsedeglat
from pwkit.environments.casa import dftphotom

# Here's a sample way to specify the coordinates to
# use; anything that produces J2000 RA/Dec in radians
# will work:
ra = parsehours('17:45:00') # result is in radians
dec = parsedeglat('-23:00:00') # result is in radians

cfg = dftphotom.Config()
cfg.vis = 'path/to/vis/data.ms'
cfg.format = dftphotom.PandasOutputFormat()
cfg.outstream = open('mydata.txt', 'w')
cfg.rephase = (ra, dec)
```

(continues on next page)

(continued from previous page)

```
dftphotom.dftphotom(cfg)
```

The main algorithm is implemented in the *dftphotom()* function. All of the algorithm parameters are passed to the function via a *Config* structure. You can create one *Config* and call *dftphotom()* with it repeatedly, altering the parameters each time if you have a series of related computations to run.

API Reference

class pwkit.environments.casa.dftphotom.**Config**

vis = KeywordOptions(required=True, subval=<class 'str'>)
The path to the visibility MeasurementSet to process. No default; you must specify a value before calling *dftphotom()*.

datacol = 'data'
A string specifying which visibility data column to process: `data`, `corrected_data`, or `model_data`. Default `'data'`.

believeweights = False
Whether to trust that the data-weight information in the MS accurately describe the noise in their corresponding visibilities. Default False.

outstream
A file-like object into which the output table of data will be written. No default in the Python interface.

datascale = 1000000.0
The amount by which to scale the computed values before emitting them as text. The default is 1e6, which means that the output will be in microJanskys if the underlying data are calibrated to Jansky units.

format
An instance of a class that will format the algorithm outputs into text. Either *HumaneOutputFormat* or *PandasOutputFormat*.

rephase
A coordinate tuple `(ra, dec)`, giving a location towards which to rephase the visibility data. The inputs are in radians. If left as `None`, the visibilities will not be rephased.

Generic CASA data-selection options

array = <class 'str'>

baseline = <class 'str'>

field = <class 'str'>

observation = <class 'str'>

polarization = 'RR,LL'

scan = <class 'str'>

scanintent = <class 'str'>

spw = <class 'str'>

taql = <class 'str'>

time = <class 'str'>

uvdist = <class 'str'>

Generic CASA task options

loglevel = 'warn'

pwkit.environments.casa.dftphotom.**dftphotom**(*cfg*)
Run the discrete-Fourier-transform photometry algorithm.

See the module-level documentation and the output of `casatask dftphotom --help` for help. All of the algorithm configuration is specified in the *cfg* argument, which is an instance of *Config*.

pwkit.environments.casa.dftphotom.**dftphotom_cli**(*argv*)
Command-line access to the *dftphotom()* algorithm.

This function implements the behavior of the command-line `casatask dftphotom` tool, wrapped up into a single callable function. The argument *argv* is a list of command-line arguments, in Unix style where the zeroth item is the name of the command.

class pwkit.environments.casa.dftphotom.**HumaneOutputFormat**

class pwkit.environments.casa.dftphotom.**PandasOutputFormat**

Structured scripting within `casapy` (`pwkit.environments.casa.scripting`)

pwkit.environments.casa.scripting - scripted invocation of casapy.

The "casapy" program is extremely resistant to encapsulated scripting – it pops up GUI windows and child processes, leaves log files around, provides a non-vanilla Python environment, and so on. However, sometimes scripting CASA is what we need to do. This tool enables that.

We provide a single-purpose CLI tool for this functionality, so that you can write standalone scripts with a hashbang line of "#! /usr/bin/env pkcasascript" – hashbang lines support only one extra command-line argument, so if we're using "env" we can't take a multitool approach.

class pwkit.environments.casa.scripting.**CasapyScript**(*script*, *raise_on_error=True*, ***kwargs*)
Context manager for launching a script in the casapy environment. This involves creating a temporary wrapper and then using the CasaEnvironment to run it in a temporary directory.

When this context manager is entered, the script is launched and the calling process waits until it finishes. This object is returned. The *with* statement body is then executed so that information can be extracted from the results of the casapy invocation. When the context manager is exited, the casapy files are (usually) cleaned up.

Attributes:

args the arguments to passed to the script.

env the CasaEnvironment used to launch the casapy process.

exitcode the exit code of the casapy process. 0 is success. 127 indicates an intentional error exit by the script; additional diagnostics don't need printing and the work directory doesn't need preservation. Negative values indicate death from a signal.

proc the *subprocess.Popen* instance of casapy; inside the context manager body it's already exited.

rmtree boolean; whether to delete the working tree upon context manager exit.

script the path to the script to be invoked.

workdir the working directory in which casapy was started.

wrapped the path to the wrapper script run inside casapy.

There is a very large overhead to running casapy scripts. The outer Python code sleeps for at least 5 seconds to allow various cleanups to happen.

Merging spectral windows in visibility data (`pwkit.environments.casa.spwglue`)

pwkit.environments.casa.spwglue - merge spectral windows in a MeasurementSet

I find that merging windows in this way offers a lot of advantages. This procesing step is very slow, however.

class pwkit.environments.casa.spwglue.**Progress**
This could be split out; it's useful.

class pwkit.environments.casa.spwglue.**Config**

hackfield
alias of builtins.int

meanbp
alias of builtins.str

7.1.3 Using CASA in the `pwkit.environments` framework

The module *pwkit.environments* implements a system for running sub-programs that depend on large, external software environments such as CASA. It provides a command-line tool, `pkenvtool`, that you can use to run code in a controlled CASA environment.

Some of the *tasks* provided by *pwkit* rely on this framework to implement their functionality — in these cases, the value that *pwkit* is providing is that it lets you access complex CASA functionality through a simple function call in a standard Python environment, rather than requiring manual invocation in a `casapy` shell.

In order to use these tasks or the CASA features of the `pkenvtool` program, you must tell the *pwkit.environments* system where your CASA installation may be found. To do this, just export an environment variable named `$PWKIT_CASA` that stores the path to the CASA installation root. In other words, the file `$PWKIT_CASA/bin/casa` should exist. (Well, the code also checks for `$PWKIT_CASA/bin/casapy` to try to be compatible with older CASA versions.) The environments system will take care of the rest.

Note: does this work on 32-bit systems? Does this work on Macs?

7.1.4 CASA installation notes

Download tarball as linked from here. The tarball unpacks to some versioned subdirectory. The names and version codes are highly variable and annoying.

7.2 HEASoft (`pwkit.environments.heasoft`)

This module provides an encapsulated scheme for running HEASoft tools within the *pwkit.environments* framework.

This module sets things up such that parameter files for HEASoft tasks ("pfiles") land in the directory `~/.local/share/hea-pfiles/`.

7.2.1 Using HEASoft in the `pwkit.environments` framework

The module *`pwkit.environments`* implements a system for running sub-programs that depend on large, external software environments such as HEASoft. It provides a command-line tool, `pkenvtool`, that you can use to run HEASoft code in a controlled environment.

In order to use this module, you must tell the *`pwkit.environments`* system where your HEASoft installation may be found. To do this, just export an environment variable named `$PWKIT_HEASOFT` that stores the path to the *platform-specific* subdirectory of your HEASoft installation. In other words, the file `$PWKIT_HEASOFT/headas-init.sh` should exist. On a Linux system the value of `$PWKIT_HEASOFT` might end with something like `x86_64-unknown-linux-gnu-libc2.23`. Once you've correctly set this environment variable, the environments system will take care of the rest.

7.2.2 HEAsoft installation notes

The following examples assume version 6.21 for concreteness. Substitute your actual version as needed, of course.

Installation of HEASoft from source is strongly recommended. Download the source code from a URL like this one. The HEASoft website lets you customize the tarball, but it's probably easiest just to do the full install every time. The tarball unpacks into a directory named like `heasoft-6.21/...` so you can safely `curl|tar` in your source-code directory.

To build, then run something like:

```
$ cd heasoft-6.21/BUILD_DIR
$ ./configure --prefix=/a/heasoft/6.21
$ make # note: not parallel-friendly
$ make install
```

You then need to fetch the `CALDB` data files into the HEASoft installation directory:

```
$ cd /a/heasoft/6.21
$ wget http://heasarc.gsfc.nasa.gov/FTP/caldb/software/tools/caldb.config
$ wget http://heasarc.gsfc.nasa.gov/FTP/caldb/software/tools/alias_config.fits
```

7.3 SAS (`pwkit.environments.sas`)

sas - running software in the SAS environment

To use, export an environment variable $PWKIT_SAS pointing to the SAS installation root. The files $PWKIT_SAS/RELEASE and $PWKIT_SAS/setsas.sh should exist. The "current calibration files" (CCF) should be accessible as $PWKIT_SAS/ccf/; a symlink may make sense if multiple SAS versions are going to be used.

SAS is unusual because you need to set up some magic environment variables specific to the dataset that you're working with. There is also default preparation to be run on each dataset before anything useful can be done.

7.3.1 Unpacking data sets

Data sets are downloaded as tar.gz files. Those unpack to a few files in '.' including a .TAR file, which should be unpacked too. That unpacks to a bunch of data files in '.' as well.

7.3.2 SAS installation notes

Download tarball from, e.g.,

ftp://legacy.gsfc.nasa.gov/xmm/software/sas/14.0.0/64/Linux/Fedora20/

Tarball unpacks installation script and data into '.', and the installation script sets up a SAS install in a versioned subdirectory of '.', so curl|tar should be run from something like /a/sas:

```
$ ./install.sh
```

The CCF are like CALDB and need to be rsynced – see the update-ccf subcommand.

7.3.3 ODF data format notes

ODF files all have names in the format RRRR_NNNNNNNNNN_IIUEEECCMMM.ZZZ where:

RRRR revolution (orbit) number

NNNNNNNNNN obs ID

II The instrument:

- **OM** optical monitor
- **R1** RGS (reflection grating spectrometer) unit 1
- **R2** RGS 2
- **M1** EPIC (imaging camera) MOS 1 detector
- **M2** EPIC (imaging camera) MOS 2 detector
- **PN** EPIC (imaging camera) PN detector
- **RM** EPIC radiation monitor
- **SC** spacecraft

U Scheduling status of exposure:

- **S** scheduled
- **U** unscheduled
- **X** N/A

EEE exposure number

CC CCD/OM-window ID

MMM data type of file (many; not listed here)

ZZZ file extension

See the `make-*-aliases` commands for tools that generate symlinks with saner names.

7.3.4 More detailed documentation

Interacting with SAS data sets (`pwkit.environments.sas.data`)

pwkit.environments.sas.data - loading up SAS data sets

Data sets

BaseSASData(path[, mjd0, t0])	
Events(path[, mjd0, t0])	
GTIData(path[, mjd0, t0])	
Lightcurve(path[, mjd0, t0])	
RegionData(path[, mjd0, t0])	

class pwkit.environments.sas.data.**BaseSASData**(*path*, *mjd0=None*, *t0=None*)

class pwkit.environments.sas.data.**Events**(*path*, *mjd0=None*, *t0=None*)

class pwkit.environments.sas.data.**GTIData**(*path*, *mjd0=None*, *t0=None*)

class pwkit.environments.sas.data.**Lightcurve**(*path*, *mjd0=None*, *t0=None*)

class pwkit.environments.sas.data.**RegionData**(*path*, *mjd0=None*, *t0=None*)

7.4 CIAO (pwkit.environments.ciao)

ciao - running software in the CIAO environment

To use, export an environment variable $PWKIT_CIAO pointing to the CIAO installation root.

7.4.1 Unpacking data sets

Data sets are provided as tar files. They unpack to a directory named by the "obsid" which contains an `oif.fits` file and `primary` and `secondary` subdirectories.

7.4.2 CIAO installation notes

Download installer script from http://cxc.harvard.edu/ciao/download/. Select some kind of parent directory like `/soft/ciao` for both downloading tarballs and installing CIAO itself. This may also download and install the large "caldb" data set. All of the files will end up in a subdirectory such as `/soft/ciao/ciao-4.8`.

class pwkit.environments.ciao.**CiaoTool**

invoke_command(*cmd*, *args*, ***kwargs*)
This function mainly exists to be overridden by subclasses.

CHAPTER 8

Tools for writing command-line programs

This documentation has a lot of stubs.

8.1 Utilities for command-line programs (`pwkit.cli`)

pwkit.cli - miscellaneous utilities for command-line programs.

Functions:

backtrace_on_usr1 - Make it so that a Python backtrace is printed on SIGUSR1. check_usage - Print usage and exit if –help is in argv. die - Print an error and exit. fork_detached_process - Fork a detached process. pop_option - Check for a single command-line option. propagate_sigint - Ensure that calling shells know when we die from SIGINT. show_usage - Print a usage message. unicode_stdio - Ensure that sys.std{in,out,err} accept unicode strings. warn - Print a warning. wrong_usage - Print an error about wrong usage and the usage help.

Context managers:

print_tracebacks - Catch exceptions and print tracebacks without reraising them.

Submodules:

multitool - Framework for command-line programs with sub-commands.

`pwkit.cli.`**`check_usage`**(*docstring*, *argv=None*, *usageifnoargs=False*)

Check if the program has been run with a –help argument; if so, print usage information and exit.

Parameters

- **`docstring`** (*`str`*) – the program help text
- **`argv`** – the program arguments; taken as `sys.argv` if given as `None` (the default). (Note that this implies `argv[0]` should be the program name and not the first option.)
- **`usageifnoargs`** (*`bool`*) – if `True`, usage information will be printed and the program will exit if no command-line arguments are passed. If “long”, print long usasge. Default is `False`.

This function is intended for small programs launched from the command line. The intention is for the program help information to be written in its docstring, and then for the preamble to contain something like:

```
"""myprogram - this is all the usage help you get"""
import sys
... # other setup
check_usage (__doc__)
... # go on with business
```

If it is determined that usage information should be shown, *show_usage()* is called and the program exits.

See also *wrong_usage()*.

pwkit.cli.**die**(*fmt*, **args*)

Raise a SystemExit exception with a formatted error message.

Parameters

- **fmt** (*str*) – a format string
- **args** – arguments to the format string

If *args* is empty, a SystemExit exception is raised with the argument `'error: ' + str (fmt)`. Otherwise, the string component is `fmt % args`. If uncaught, the interpreter exits with an error code and prints the exception argument.

Example:

```
if ndim != 3:
   die ('require exactly 3 dimensions, not %d', ndim)
```

pwkit.cli.**fork_detached_process**()

Fork this process, creating a subprocess detached from the current context.

Returns a *pwkit.Holder* instance with information about what happened. Its fields are:

whoami A string, either "original" or "forked" depending on which process we are.

pipe An open binary file descriptor. It is readable by the original process and writable by the forked one. This can be used to pass information from the forked process to the one that launched it.

forkedpid The PID of the forked process. Note that this process is *not* a child of the original one, so waitpid() and friends may not be used on it.

Example:

```
from pwkit import cli

info = cli.fork_detached_process ()
if info.whoami == 'original':
    message = info.pipe.readline ().decode ('utf-8')
    if not len (message):
        cli.die ('forked process (PID %d) appears to have died', info.forkedpid)
    info.pipe.close ()
    print ('forked process said:', message)
else:
    info.pipe.write ('hello world'.encode ('utf-8'))
    info.pipe.close ()
```

As always, the *vital* thing to understand is that immediately after a call to this function, you have **two** nearly-identical but **entirely independent** programs that are now both running simultaneously. Until you execute some

kind of `if` statement, the only difference between the two processes is the value of the `info.whoami` field and whether `info.pipe` is readable or writeable.

This function uses `os.fork()` twice and also calls `os.setsid()` in between the two invocations, which creates new session and process groups for the forked subprocess. It does *not* perform other operations that you might want, such as changing the current directory, dropping privileges, closing file descriptors, and so on. For more discussion of best practices when it comes to "daemonizing" processes, see (stalled) PEP 3143.

pwkit.cli.**pop_option**(*ident*, *argv=None*)

A lame routine for grabbing command-line arguments. Returns a boolean indicating whether the option was present. If it was, it's removed from the argument string. Because of the lame behavior, options can't be combined, and non-boolean options aren't supported. Operates on sys.argv by default.

Note that this will proceed merrily if argv[0] matches your option.

class pwkit.cli.**print_tracebacks**(*types=(<class 'Exception'>,)*, *header=None*, *file=None*)

Context manager that catches exceptions and prints their tracebacks without reraising them. Intended for robust programs that want to continue execution even if something bad happens; this provides the infrastructure to swallow exceptions while still preserving exception information for later debugging.

You can specify which exception classes to catch with the *types* keyword argument to the constructor. The *header* keyword will be printed if specified; this could be used to add contextual information. The *file* keyword specifies the destination for the printed output; default is sys.stderr.

Instances preserve the exception information in the fields 'etype', 'evalue', and 'etb' if your program in fact wants to do something with the information. One basic use would be checking whether an exception did, in fact, occur.

pwkit.cli.**show_usage**(*docstring*, *short*, *stream*, *exitcode*)

Print program usage information and exit.

Parameters **docstring** (*str*) – the program help text

This function just prints *docstring* and exits. In most cases, the function *check_usage()* should be used: it automatically checks `sys.argv` for a sole "-h" or "–help" argument and invokes this function.

This function is provided in case there are instances where the user should get a friendly usage message that *check_usage()* doesn't catch. It can be contrasted with *wrong_usage()*, which prints a terser usage message and exits with an error code.

pwkit.cli.**unicode_stdio**()

Make sure that the standard I/O streams accept Unicode.

In Python 2, the standard I/O streams accept bytes, not Unicode characters. This means that in principle every Unicode string that we want to output should be encoded to utf-8 before print()ing. But Python 2.X has a hack where, if the output is a terminal, it will automatically encode your strings, using UTF-8 in most cases.

BUT this hack doesn't kick in if you pipe your program's output to another program. So it's easy to write a tool that works fine in most cases but then blows up when you log its output to a file.

The proper solution is just to do the encoding right. This function sets things up to do this in the most sensible way I can devise, if we're running on Python 2. This approach sets up compatibility with Python 3, which has the stdio streams be in text mode rather than bytes mode to begin with.

Basically, every command-line Python program should call this right at startup. I'm tempted to just invoke this code whenever this module is imported since I foresee many accidentally omissions of the call.

pwkit.cli.**wrong_usage**(*docstring*, **rest*)

Print a message indicating invalid command-line arguments and exit with an error code.

Parameters

- **docstring** (*str*) – the program help text

- **rest** – an optional specific error message

This function is intended for small programs launched from the command line. The intention is for the program help information to be written in its docstring, and then for argument checking to look something like this:

```
"""mytask <input> <output>

Do something to the input to create the output.
"""
...
import sys
... # other setup
check_usage (__doc__)
... # more setup
if len (sys.argv) != 3:
   wrong_usage (__doc__, "expect exactly 2 arguments, not %d",
                len (sys.argv))
```

When called, an error message is printed along with the *first stanza* of *docstring*. The program then exits with an error code and a suggestion to run the program with a –help argument to see more detailed usage information. The "first stanza" of *docstring* is defined as everything up until the first blank line, ignoring any leading blank lines.

The optional message in *rest* is treated as follows. If *rest* is empty, the error message "invalid command-line arguments" is printed. If it is a single item, the stringification of that item is printed. If it is more than one item, the first item is treated as a format string, and it is percent-formatted with the remaining values. See the above example.

See also *check_usage()* and *show_usage()*.

8.2 Parsing keyword-style program arguments (pwkit.kwargv)

The *pwkit.kwargv* module provides a framework for parsing keyword-style arguments to command-line programs. It's designed so that you can easily make a routine with complex, structured configuration parameters that can also be driven from the command line.

Keywords are defined by declaring a subclass of the *ParseKeywords* class with fields corresponding to the support keywords:

```
from pwkit.kwargv import ParseKeywords, Custom

class MyConfig(ParseKeywords):
    foo = 1
    bar = str
    multi = [int]
    extra = Custom(float, required=True)

    @Custom(str)
    def declination(value):
        from pwkit.astutil import parsedeglat
        return parsedeglat(value)
```

Instantiating the subclass fills in all defaults. Calling the *ParseKeywords.parse()* method parses a list of strings (defaulting to `sys.argv[1:]`) and updates the instance's properties. This framework is designed so that you can provide complex configuration to an algorithm either programmatically, or on the command line. A typical use would be:

```
from pwkit.kwargv import ParseKeywords, Custom

class MyConfig(ParseKeywords):
    niter = 1
    input = str
    scales = [int]
    # ...

def my_complex_algorithm(cfg):
   from pwkit.io import Path
   data = Path(cfg.input).read_fits()

   for i in range(cfg.niter):
       # ....

def call_algorithm_in_code():
    cfg = MyConfig()
    cfg.input = 'testfile.fits'
    # ...
    my_complex_algorithm(cfg)

if __name__ == '__main__':
    cfg = MyConfig().parse()
    my_complex_algorithm(cfg)
```

You could then execute the module as a program and specify arguments in the form `./program niter=5 input=otherfile.fits`.

8.2.1 Keyword Specification Format

Arguments are specified in the following ways:

- `foo = 1` defines a keyword with a default value, type inferred as `int`. Likewise for `str`, `bool`, `float`.
- `bar = str` defines an string keyword with default value of None. Likewise for `int`, `bool`, `float`.
- `multi = [int]` parses as a list of integers of any length, defaulting to the empty list `[]` (I call these "flexible" lists.). List items are separated by commas on the command line.
- `other = [3.0, int]` parses as a 2-element list, defaulting to `[3.0, None]`. If one value is given, the first array item is parsed, and the second is left as its default. (I call these "fixed" lists.)
- `extra = Custom(float, required=True)` parses like `float` and then customizes keyword properties. Supported properties are the attributes of the *KeywordInfo* class.
- Use *Custom* as a decorator (`@Custom`) on a function `foo` defines a keyword `foo` that's parsed according to the *Custom* specification, then has its value fixed up by calling the `foo()` function after the basic parsing. That is, the final value is `foo (intermediate_value)`. A common pattern is to use a fixup function for a fixed list where the first few values are mandatory (see *KeywordInfo.minvals* below) but later values can be guessed or defaulted.

See the *KeywordInfo* documentation for specification of additional keyword properties that may be specified. The `Custom` name is simply an alias for *KeywordInfo*.

pwkit.kwargv.**Custom**
alias of pwkit.kwargv.KeywordOptions

exception pwkit.kwargv.**KwargvError**(*fmt*, **args*)
Raised when invalid arguments have been provided.

exception pwkit.kwargv.**ParseError**(*fmt*, **args*)
Raised when the structure of the arguments appears legitimate, but a particular value cannot be parsed into its expected type.

class pwkit.kwargv.**KeywordInfo**
Properties that a keyword argument may have.

default = None
The default value for the keyword if it's left unspecified.

fixupfunc = None
If not `None`, the final value of the keyword is set to the return value of `fixupfunc(intermediate_value)`.

maxvals = None
The maximum number of values allowed. This only applies for flexible lists; fixed lists have predetermined sizes.

minvals = 0
The minimum number of values allowed in a flexible list, *if the keyword is specified at all*. If you want `minvals = 1`, use `required = True`.

parser = None
A callable used to convert the argument text to a Python value. This attribute is assigned automatically upon setup.

printexc = False
Print the exception as normal if there's an exception when parsing the keyword value. Otherwise there's just a message along the lines of "cannot parse value <val> for keyword <kw>".

repeatable = False
If true, the keyword value(s) will always be contained in a list. If they keyword is specified multiple times (i.e. `./program kw=1 kw=2`), the list will have multiple items (`cfg.kw = [1, 2]`). If the keyword is list-valued, using this will result in a list of lists.

required = False
Whether an error should be raised if the keyword is not seen while parsing.

scale = None
If not `None`, multiply numeric values by this number after parsing.

sep = ','
The textual separator between items for list-valued keywords.

uiname = None
The name of the keyword as parsed from the command-line. For instance, `some_value = Custom(int, uiname="some-value")` will result in a keyword that the user sets by calling `./program some-value=3`. This provides a mechanism to support keyword names that are not legal Python identifiers.

class pwkit.kwargv.**ParseKeywords**
The template class for defining your keyword arguments. A subclass of *pwkit.Holder*. Declare attributes in a subclass following the scheme described above, then call the *ParseKeywords.parse()* method.

parse(*args=None*)
Parse textual keywords as described by this class's attributes, and update this instance's attributes with the parsed values. *args* is a list of strings; if `None`, it defaults to `sys.argv[1:]`. Returns *self* for convenience. Raises *KwargvError* if invalid keywords are encountered.

See also *ParseKeywords.parse_or_die()*.

parse_or_die(*args=None*)
Like *ParseKeywords.parse()*, but calls `pkwit.cli.die()` if a *KwargvError* is raised, printing the exception text. Returns *self* for convenience.

`pwkit.kwargv.`**basic**(*args=None*)
Parse the string list *args* as a set of keyword arguments in a very simple-minded way, splitting on equals signs. Returns a *pwkit.Holder* instance with attributes set to strings. The form `+foo` is mapped to setting `foo = True` on the *pwkit.Holder* instance. If *args* is `None`, `sys.argv[1:]` is used. Raises *KwargvError* on invalid arguments (i.e., ones without an equals sign or a leading plus sign).

8.3 Command-line programs with sub-commands (pwkit.cli.multitool)

pwkit.cli.multitool - Framework for command-line tools with sub-commands

This module provides a framework for quickly creating command-line programs that have multiple independent sub-commands (similar to the way Git's interface works).

Classes:

Command A command supported by the tool.

DelegatingCommand A command that delegates to named sub-commands.

HelpCommand A command that prints the help for other commands.

Multitool The tool itself.

UsageError Raised if illegal command-line arguments are used.

Functions:

invoke_tool Run as a tool and exit.

Standard usage:

```
class MyCommand (multitool.Command):
  name = 'info'
  summary = 'Do something useful.'

  def invoke (self, args, **kwargs):
    print ('hello')

class MyTool (multitool.MultiTool):
  cli_name = 'mytool'
  summary = 'Do several useful things.'

HelpCommand = multitool.HelpCommand # optional

def commandline ():
  multitool.invoke_tool (globals ())
```

`pwkit.cli.multitool.`**invoke_tool**(*namespace*, *tool_class=None*)
Invoke a tool and exit.

namespace is a namespace-type dict from which the tool is initialized. It should contain exactly one value that is a *Multitool* subclass, and this subclass will be instantiated and populated (see *Multitool.populate()*) using the other items in the namespace. Instances and subclasses of *Command* will therefore be registered with the *Multitool*. The tool is then invoked.

pwkit.cli.propagate_sigint() and *pwkit.cli.unicode_stdio()* are called at the start of this function. It should therefore be only called immediately upon startup of the Python interpreter.

This function always exits with an exception. The exception will be SystemExit (0) in case of success.

The intended invocation is *invoke_tool (globals ())* in some module that defines a *Multitool* subclass and multiple *Command* subclasses.

If *tool_class* is not None, this is used as the tool class rather than searching *namespace*, potentially avoiding problems with modules containing multiple *Multitool* implementations.

class pwkit.cli.multitool.**Command**
A command in a multifunctional CLI tool.

For historical reasons, this class defaults to a homebrew argument parsing system. Use *ArgparsingCommand* for a better system based on the *argparse* module.

Attributes:

argspec One-line string summarizing the command-line arguments that should be passed to this command.

help_if_no_args If True, usage help will automatically be displayed if no command-line arguments are given.

more_help Additional help text to be displayed below the summary (optional).

name The command's name, as should be specified at the CLI.

summary A one-line summary of this command's functionality.

Functions:

invoke(self, args, **kwargs) Execute this command.

'name' must be set; other attributes are optional, although at least 'summary' and 'argspec' should be set. 'invoke()' must be implemented.

invoke(*args*, ***kwargs*)
Invoke this command. 'args' is a list of the remaining command-line arguments. 'kwargs' contains at least 'argv0', which is the equivalent of, well, *argv[0]* for this command; 'tool', the originating Multitool instance; and 'parent', the parent DelegatingCommand instance. Other kwargs may be added in an application-specific manner. Basic processing of '–help' will already have been done if invoked through invoke_with_usage().

invoke_with_usage(*args*, ***kwargs*)
Invoke the command with standardized usage-help processing. Same calling convention as *Command.invoke()*.

class pwkit.cli.multitool.**DelegatingCommand**(*populate_from_self=True*)
A command that delegates to sub-commands.

Attributes:

cmd_desc The noun used to desribe the sub-commands.

usage_tmpl A formatting template for long tool usage. The default is almost surely acceptable.

Functions:

register Register a new sub-command.

populate Register many sub-commands automatically.

invoke(*args*, ***kwargs*)
Invoke this command. 'args' is a list of the remaining command-line arguments. 'kwargs' contains at least 'argv0', which is the equivalent of, well, *argv[0]* for this command; 'tool', the originating Multitool instance; and 'parent', the parent DelegatingCommand instance. Other kwargs may be added in an

application-specific manner. Basic processing of '–help' will already have been done if invoked through invoke_with_usage().

invoke_command(*cmd*, *args*, ***kwargs*)
This function mainly exists to be overridden by subclasses.

populate(*values*)
Register multiple new commands by investigating the iterable *values*. For each item in *values*, instances of *Command* are registered, and subclasses of *Command* are instantiated (with no arguments passed to the constructor) and registered. Other kinds of values are ignored. Returns 'self'.

register(*cmd*)
Register a new command with the tool. 'cmd' is expected to be an instance of *Command*, although here only the *cmd.name* attribute is investigated. Multiple commands with the same name are not allowed to be registered. Returns 'self'.

class pwkit.cli.multitool.**HelpCommand**

invoke(*args*, *parent=None*, *parent_kwargs=None*, ***kwargs*)
Invoke this command. 'args' is a list of the remaining command-line arguments. 'kwargs' contains at least 'argv0', which is the equivalent of, well, *argv[0]* for this command; 'tool', the originating Multitool instance; and 'parent', the parent DelegatingCommand instance. Other kwargs may be added in an application-specific manner. Basic processing of '–help' will already have been done if invoked through invoke_with_usage().

class pwkit.cli.multitool.**Multitool**
A command-line tool with multiple sub-commands.

Attributes:

cli_name - The usual name of this tool on the command line. more_help - Additional help text. summary - A one-line summary of this tool's functionality.

Functions:

commandline - Execute a command as if invoked from the command-line. register - Register a new command. populate - Register many commands automatically.

commandline(*argv*)
Run as if invoked from the command line. 'argv' is a Unix-style list of arguments, where the zeroth item is the program name (which is ignored here). Usage help is printed if deemed appropriate (e.g., no arguments are given). This function always terminates with an exception, with the exception being a SystemExit(0) in case of success.

Note that we don't actually use *argv[0]* to set *argv0* because it will generally be the full path to the script name, which is unattractive.

exception pwkit.cli.multitool.**UsageError**(*fmt*, **args*)
Raised if illegal command-line arguments are used in a Multitool program.

CHAPTER 9

Behind-the-scenes infrastructure

This documentation has a lot of stubs.

9.1 Interfacing with other software environments (`pwkit.environments`)

pwkit.environments - working with external software environments

Classes:

Environment - base class for launching programs in an external environment.

Submodules:

heasoft - HEAsoft sas - SAS

Functions:

prepend_environ_path - Prepend into a $PATH in an environment dict. prepend_path - Prepend text into a $PATH-like environment variable. user_data_path - Generate paths for storing miscellaneous user data.

Standard usage is to create an *Environment* instance, then use its *launch(argv, ...)* method to run programs in the specified environment. *launch()* returns a *subprocess.Popen* instance that can be used in the standard ways.

`pwkit.environments.`**`prepend_environ_path`**(*env*, *name*, *text*, *pathsep=':'*)

Prepend *text* into a $PATH-like environment variable. *env* is a dictionary of environment variables and *name* is the variable name. *pathsep* is the character separating path elements, defaulting to *os.pathsep*. The variable will be created if it is not already in *env*. Returns *env*.

Example:

```
prepend_environ_path(env, 'PATH', '/mypackage/bin')
```

The *name* and *text* arguments should be *str* objects; that is, bytes in Python 2 and Unicode in Python 3. Literal strings will be OK unless you use the `from __future__ import unicode_literals` feature.

pwkit.environments.**prepend_path**(*orig*, *text*, *pathsep=':'*)

Returns a $PATH-like environment variable with *text* prepended. *orig* is the original variable value, or None. *pathsep* is the character separating path elements, defaulting to *os.pathsep*.

Example:

newpath = cli.prepend_path(oldpath, '/mypackage/bin')

See also *prepend_environ_path*.

9.2 Helper for decorators on class methods (pwkit.method_decorator)

Python decorator that knows the class the decorated method is bound to.

Please see full description here: https://github.com/denis-ryzhkov/method_decorator/blob/master/README.md

method_decorator version 0.1.3 Copyright (C) 2013 by Denis Ryzhkov <denisr@denisr.com> MIT License, see http://opensource.org/licenses/MIT

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

D

E

F

G

H

I

J

K

L

M

R

S

T

U

V

W

X