# PALFA Pipeline2.0 Documentation

Patrick Lazarus, Alex Samoilov, ...
plazar@physics.mcgill.ca, alex@sequencefactory.com

March 9, 2011

**Abstract**

# 1 Installation

## 1.1 Dependencies

The PALFA pipeline has various dependencies. All of the following packages are required to run the pipeline.

- PRESTO (https://github.com/scottransom/presto)
- psrfits_utils (be sure to pull Kevin Stovall's version of psrfits_utils. It contains the merging code required to analyse PALFA Mock spectrometer data. https://github.com/kstovall/psrfits_utils.git)
- numpy (http://numpy.scipy.org/)
- M2Crypto (http://pypi.python.org/pypi/M2Crypto)
- PBSQuery (if PBS is being used - http://subtrac.sara.nl/oss/pbs_python/wiki/TorqueInst
- pyfits (http://www.stsci.edu/resources/software_hardware/pyfits)
- pyodbc (http://pypi.python.org/pypi/pyodbc/)
- suds (http://pypi.python.org/pypi/suds)

*NOTE: Some of these dependencies have requirements of their own. For example, PRESTO requires cfitsio, TEMPO, numpy, scipy, etc. Be sure to follow the installation instructions for each packages.*

## 1.2 Getting Started

Here we will present basic step-by-step instructions for setting up the pipeline.

**Step 1** Create a directory where you want the pipeline to be installed.

**Step 2** Download the pipeline source files. This should be done by cloning the git repository on github.com: https://github.com/plazar/pipeline2.0.

```
$ git clone git://github.com/plazar/pipeline2.0.git
```

*NOTE: This will create a sub-directory called "pipeline2.0".*

**Step 3** Add the pipeline's lib/python directory to your PYTHONPATH environment variable.

**Step 4** Create configuration files using the examples provided. Modify the settings so they are appropriate for your system. It is possible to check the sanity of your configurations by calling a configuration file with the python interpreter. An error message will be displayed if a configuration type or value is invalid.

**Step 5** Set up the database. This is done using the `create_database.py` script.

**Step 6** Perform tests. In the `test/` directory there are some tests that can be run to see if you can connect to the common DB (`commondb_test.py`), e-mail system (`MailerTest.py`), and FTP server (`cornellftp_test.py`). To perform the tests call each script with the python interpreter.

**Step 7** Start the downloader using `StartDownloader.py`.

*NOTE: The downloader does not need to be run on the same computer as the job pooler. However, the directory where the downloader saves files must be accessible from the computer where the job pooler is being run.*

Alternatively, it is possible to add files to the job-tracker database without using the downloader. To do this use `add_files.py`. Be sure to set the `delete_rawdata` configuration to `False` if you do not want raw data files to be deleted when the pipeline no longer requires them.

**Step 8** Start the job pooler using `StartJobPool.py`. The job pooler will start submitting jobs when data files are finished downloading.

**Step 9** Start the uploader using `StartJobUploader.py`. The uploader will upload results to the common database when jobs are successfully processed.

# 2 The Pipeline

An overview of the pipeline.

## 2.1 Features

- Sanity check of configurations
- Dynamic zapping
- Automatic download of data files
- File size checks when downloading
- Notification of errors by email
- Automatic (configurable) retry of failed jobs
- Results are uploaded as a single transaction, so no partial uploads when errors occur
- Automatic (configurable) deletion of data files (upon success or terminal failure)

## 2.2 Components

### 2.2.1 Job-tracking Database

Each job run through the pipeline passes through multiple stages before it is completed. To track the state of each job, and log its history a SQLite3 database is used. The database is simply a (readable/writable) binary file formatted according to the SQLite3 format. The database file can be access using SQL statements using python's sqlite3 module, which is part of python's standard library, as of python version 2.5.

Relevant links:

- http://docs.python.org/library/sqlite3.html
- http://www.sqlite.org/

The job-tracker database governs all aspects of the pipeline. It maintains lists of requests for data, downloads, jobs, files associated with each job, processing attempts, and upload attempts.

Details for each table in the database are presented in Section 3.4.

### 2.2.2 Downloader

### 2.2.3 Job Pooler

### 2.2.4 Uploader

#### Headers

#### Candidates

**Diagnostics** Diagnostics stored in the common database are split into two types: Numeric values and binary data. Some of the "binary data" diagnostics are actually plain text files.

The following per-beam diagnostics are uploaded to the common database:

- *RFI mask percentage* (Numeric value)
  Percentage of data masked due to RFI.
- *Num cands folded* (Numeric Value)
  The number of candidates folded.

- *Num cands produced* (Numeric Value)
  The total number of candidates produced, including those with sigma lower than the folding threshold.
- *Min sigma folded* (Numeric value)
  The smallest sigma value of all folded candidates from this beam.
- *Num cands above threshold* (Numeric value)
  The number of candidates produced (but not necessarily folded) that are above the desired sigma threshold.
- *RFIfind png* (Binary data)
  Output image produced by rfifind in png format.
- *Accelcands list* (Binary data)
  The combined and sifted list of candidates produced by accelsearch.
- *Zaplist used* (Binary data)
  The list of frequencies and ranges zapped from the power spectrum before searching this beam.

## 2.3 Objects Tracked

**Restores**

**Downloads**

**Jobs**

**Uploads**

# 3 Reference

## 3.1 Executables

`StartDownloader.py` A script to start the downloader background process.
usage: `StartDownloader.py`

`StartJobPool.py` A script to start the job pooler background process.
usage: `StartJobPool.py`

`StartJobUploader.py` A script to start the results uploader background process.
usage: `StartJobUploader.py`

## 3.2 Generic Queue Manager Interface

The Pipeline interacts with the system's queue manager using an abstract interface. This generic interface can be implemented with the specific queue manager installed on the system. The system-specific queue manager class must be derived from the `PipelineQueueManager` class. Each of the following methods of `PipelineQueueManager` must be implemented in the derived class:

**submit** Submits a job to queue manager with files list and output directory. Must return a unique string identifier for the submitted job. Raises ValueError if job was not submitted by queue manager.

**is_running** Given a unique identifier for a job, must return True or False whether the job is running or not, respectively.

**delete** Given a unique identifier for a job, remove the corresponding job from the queue. Returns True if the job was delete, False if an error occured.

**status** Return the status of the queue, i.e. the number of jobs running and the number of jobs queued.

**had_errors** Given a unique identifier for a job, return True, or False whether the job terminated with an error, or not, respectively.

**get_errors** Given a unique identifier for a job, return the contents of the error log. If there were not errors return an empty string.

## 3.3 Configurations

The pipeline requires a series of configurations to be set in order to run. The configurations for different aspects of the pipeline are divided into the different files. The configuration files can be found in the pipeline's lib/python/config/ directory.

A sanity check of configurations is performed each time a configuration file is imported. If there are errors (e.g. a setting isn't provided, a directory doesn't exist, etc.) a message is printed to the terminal, and an exception is raised causing the program to exit. It is possible to run the check the sanity of a configuration file by running it. For example, checking the sanity of `basic.py`:

```
python basic.py
```

If the settings are sane there will be no error message is output. *NOTE: it may be necessary to check the sanity of the processing configurations on the worker nodes.*

Each of required settings in each of the configration files are listed below:

### 3.3.1 basic.py

Basic pipeline parameters are found in `basic.py`.

**institution** Institution where the pipeline is being run. This is the value that will get reported in the common database.

**pipeline** The type of pipeline being run. This should be left as "PRESTO".

**survey** The name of the survey. This should be left as "PALFA2.0".

**pipelinedir** The path of the base directory containing the pipeline2.0 code.

**psrfits_utilsdir** The path of the directory containing the installation of `psrfits_utils` that is used by the pipeline.

**delete_rawdata** A boolean value that determines if raw data is deleted when results for a job are successfully uploaded to the common DB, or if the maximum number of attempts for a job is reached. *Set this value to False if you do not want the local copy of your data deleted.*

**coords_table** The path to the `PALFA_coords_table.txt` file, which contains correct coordinates for WAPP data files.

### 3.3.2 background.py

General settings for the three background scripts are found in `background.py`

**screen_output** A boolean value that should be set to True if you want log information copied to the terminal.

**jobtracker_db** The path to the SQLite job-tracker database.

**sleep** The number of seconds to sleep between iterations of the background scripts' loops.

### 3.3.3 commondb.py

The information needed to connect to the common database is in `commondb.py`.

**username** The common database user name.

**password** The password used to access the common database.

**host** The computer through which a connection to the common database will be established. This should be left at "arecibosql.tc.cornell.edu".

### 3.3.4 download.py

Informatation necessary to connect to the FTP server at Cornell and use the data-restore web applications are set in `download.py`, so are settings for controlling the behaviour of the downloader.

**api_serice_url** This is the URL of the data-restore web apps. This should be set to "http://arecibo.tc.cornell.edu/palfadataapi/dataflow.asmx?WSDL".

**api_username** This is your user name used to connect to the data-restore web apps.

**api_password** This is the password for connecting to the data-restore web apps.

**ftp_host** This is the name of the FTP server at Cornell. It should be left at "arecibo.tc.cornell.edu".

**ftp_port** This is the port to use for FTP. It should be 31001.

**ftp_username** This is the user name to connect when FTP'ing. Currently all downloaders are using a shared account, so this should have the value "palfadata".

**ftp_password** This is the password of the FTP account.

**temp** This is the directory where data are downloaded to.

**space_to_use** This is the space available for downloaded files, measured in bytes.

**numdownloads** This is the number of downloads that can be run in parallel. Currently the downloader is not threaded, so this setting is ignored.

**numrestores** This is the number of requests for data that can be active at any time. Since the downloader is not threaded only 1 request can be processed at a time, so this setting is also ignore.

**numretries** This is the number of times a download will be attempted before giving up and moving on to another file.

**log_file_path** The path of the file where the downloader logs information.

### 3.3.5 email.py

The pipeline can be configured to send emails when failures occur. The relevant settings are in `email.py`.

**enabled** A boolean value that enables mailing. If False emails will not be sent.

**smtp_host** Emails are sent using the Simple Mail Transfer Protocol (SMTP). This is the host of the SMTP server to use.

**smtp_username** The user account to connect to the SMTP server.

**smtp_password** The password to connect to the SMTP server.

**recipient** The email address that will be sent notifications.

**sender** The email address that should appear in the "From" field of the email.

**send_on_failures** A boolean value that determines if emails should be sent when a job fails.

**send_on_terminal_failures** A boolean value that determines if emails should be sent on terminal failures (i.e. job will not be re-tried). Note: if "`send_on_failures`" is True emails will be sent on terminal failures regardless of the value of "`send_on_terminal_failures`.

### 3.3.6 jobpooler.py

The setting that determine the behaviour of the Job Pooler are in `jobpooler.py`.

**base_results_directory** The path of where results from processing are stored. Note: the processing of each data set is in a separate subdirectory.

**max_jobs_running** The maximum number of jobs running at any given time.

**max_jobs_queue** The maximum number of jobs that are allowed to be queued at any given time. This must be at least 1, otherwise the Job Pooler will not submit jobs for fear of them being queued.

**max_attempts** The maximum number of times a job is submitted before being abandoned.

**queue_manager** An instance of a sub-class of `PipelineQueueManager`. This is the interface between the pipeline and the queue manager.

### 3.3.7 processing.py

Setting required during processing (i.e. on the worker nodes) are provided in `processing.py`.

**base_working_directory** The directory on the worker node where temporary working and results directories will be placed.

**default_zaplist** The path of the zaplist to use when no MJD-specific zaplist can be found.

**zaplistdir** The directory containing MJD-specific zaplists.

### 3.3.8 searching.py

The search parameters of the pipeline are located in `searching.py`. To ensure searching is done uniformly across all processing sites most of these values shouldn't be modified.

**use_subbands** A boolean value that determines if we'll dedisperse and fold using subbands. In general, it is a very good idea to use them if there is enough scratch space on the machines that are processing (∼30GB/beam processed).

**fold_rawdata** A boolean value that determines if folding should be done from the raw FITS files (as opposed to subbands or dedispersed FITS files).

**datatype_flag** This is the flag provided to PRESTO programs to indicated the data type. This should be "-psrfits".

**rfifind_chunk_time** The duration of each chunk in `rfifind`.

**singlepulse_threshold** The threshold SNR for singlepulse candidate determination.

**singlepulse_plot_SNR** The threshold SNR for displaying singlepulse candidates on plots.

**singlepulse_maxwidth** The maximum pulse width (in seconds) for singlepulse searching.

**to_prepfold_sigma** The minimum incoherent sum significance required to fold a candidate.

**max_cands_to_fold** The maximum number of candidates to fold per beam.

**numhits_to_fold** The minimum number of DMs at which a periodicity must be found to be folded.

**low_DM_cutoff** The lowest DM to consider when choosing candidates to fold.

**lo_accel_numharm** The maximum number of harmonics to sum in low acceleration searching.

**lo_accel_sigma** The minimum gaussian significance threshold in low acceleration searching.

**lo_accel_zmax** The maximum z-value (in bins) used in low acceleration searching.

**lo_accel_flo** The maximum frequency searched in low acceleration searching.

**hi_accel_numharm** The maximum number of harmonics to sum in high acceleration searching.

**hi_accel_sigma** The minimum gaussian significance threshold in high acceleration searching.

**hi_accel_zmax** The maximum z-value (in bins) used in high acceleration searching.

**hi_accel_flo** The maximum frequency searched in high acceleration searching.

**low_T_to_search** The shortest observation that will be searched (in seconds).

**sifting_sigma_threshold** The minimum incoherent power sigma to consider when sifting candidates. This should be left at `to_prepfold_sigma`−1.0.

**sifting_c_pow_threshold** The coherent power threshold used when sifting.

**sifting_r_err** The Fourier bin tolerance to consider two candidates as being at the same frequency.

**sifting_short_period** The shortest candidate period to consider (in seconds).

**sifting_long_period** The longest cndidate period to consider (in seconds).

**sifting_harm_pow_cutoff** The power required in at least one harmonic.

### 3.3.9 upload.py

Configurations used by the uploader are set in `upload.py`.

**version_num** The version number of the current version of the pipeline (including versions of `PRESTO` and `psrfits_utils`). The preferred format is:

`PRESTO:{git-hash};PIPELINE:{git-hash};PSRFITS_UTILS{git-hash}`.

It is **strongly** recommended that the git-hash values are retrieved dynamically, so they are always up-to date. However, if this is not possible you are able to hard code values of the git-hashes, but they should be changed every time the versions of the software used are modified.

## 3.4 Job-Tracker Database Details

The job-tracker database contains six tables: `requests`, `files`, `download_attempts`, `jobs`, `job_files`, and `job_submits`. Each of these tables is meant to track a particular object (e.g. jobs), or events (e.g. job submissions to the queue).

A description of each database table will now be presented:

### 3.4.1 `requests`

The `requests` table tracks requests for data restores. An entry in the table is inserted when a request is made. Each entry contains the following data:

**id** (UNIQUE INTEGER) A unique identifier, automatically assigned by the database upon creation of an entry.

**size** (INTEGER) Total size of the files restored, in bytes.

**guid** (TEXT) A globally unique identifier for the restore. This serves as the directory name on the FTP server where the restored files will be placed.

**status** (TEXT) The current status of the request.

**details** (TEXT) Details about the request.

**created_at** (TEXT) Date and time when the request entry was created.

**updated_at** (TEXT) Date and time when the request entry was last updated.

### 3.4.2 files

The `files` table tracks each file available to the pipeline. A separate entry is inserted for each file. Each entry contains the following data:

**id** (UNIQUE INTEGER) A unique identifier, automatically assigned by the database upon creation of an entry.

**filename** (TEXT) The name of the file.

**remote_filename** (TEXT) The name and path of the downloaded file on the FTP server.

**request_id** (INTEGER) The identifier of the entry in the `requests` table that restored this file.

**status** (TEXT) The current status of the file.

**details** (TEXT) Details about the file.

**created_at** (TEXT) Date and time when the file entry was created.

**updated_at** (TEXT) Date and time when the file entry was last updated.

### 3.4.3 download_attempts

Every time one an attempt to download one of the files listed in the `files` table an entry is created in the `download_attempts` table. Each download attempt entry has the following data:

**id** (UNIQUE INTEGER) A unique identifier, automatically assigned by the database upon creation of an entry.

**request_id** (INTEGER) The identifier of the entry in the `files` table that this download attempt corresponds to.

**status** (TEXT) The current status of the download attempt.

**details** (TEXT) Details about the current status of the download attempt.

**created_at** (TEXT) Date and time when the download attempt entry was created.

**updated_at** (TEXT) Date and time when the download attempt entry was last updated.

### 3.4.4 jobs

Each beam-worth of data gets submitted to be processed as a single job. These jobs are tracked in the `jobs` table, and have the following data:

**id** (UNIQUE INTEGER) A unique identifier, automatically assigned by the database upon creation of an entry.

**status** (TEXT) The current status of the job.

**details** (TEXT) Details about the current status of the job.

**created_at** (TEXT) Date and time when the job entry was created.

**updated_at** (TEXT) Date and time when the job entry was last updated.

### 3.4.5 job_files

Because a job can have multiple files a mapping between jobs and files is required. The `job_files` table serves this purpose. Each row in the table has the following data:

**id** (UNIQUE INTEGER) A unique identifier, automatically assigned by the database upon creation of an entry.

**file_id** (INTEGER) The indentifier of an entry in the `files` table.

**job_id** (INTEGER) The identifier of an entry in the `jobs` table that the file referred to by `file_id` belongs to.

**created_at** (TEXT) Date and time when the entry was created.

**updated_at** (TEXT) Date and time when the entry was last updated.

### 3.4.6 job_submits

Each submission of a job to the queue to be processed is tracked in the `job_submits` table. Each job submission has the following information:

**id** (UNIQUE INTEGER) A unique identifier, automatically assigned by the database upon creation of an entry.

**queue_id** (TEXT) The unique identifier assigned to the job by the queue manager when it is submitted. This identified is used to check if the submission is still being processed.

**job_id** (INTEGER) The identifier of the job (from the `jobs` table) that was submitted for processing.

**output_dir** (TEXT) The directory where the results of processing will be output.

**status** (TEXT) The current status of the job.

**details** (TEXT) Details about the current status of the job.

**created_at** (TEXT) Date and time when the entry was created.

**updated_at** (TEXT) Date and time when the entry was last updated.