UNIVERSITY OF HAWAII AT MĀNOA
Institute for Astrononmy

Pan-STARRS Project Management System

Pan-STARRS PS-1 Image Processing Pipeline Modules
Supplementary Design Requirements

Grant Award No.	**: F29601-02-1-0268**
Prepared For	**: Pan-STARRS PMO**
Prepared By	**: Paul Price, Eugene Magnier**

Document No.	**: PSDC-430-012**
Document Date	**: January 22, 2006**
Revision	**: 11**

DISTRIBUTION STATEMENT
Approved for Public Release – Distribution is Unlimited

©Institute for Astronomy, University of Hawaii
2680 Woodlawn Drive, Honolulu, Hawaii 96822
An Equal Opportunity/Affirmative Action Institution

Submitted By:

[Insert Signature Block of Authorized Developer Representative] Date

Approved By:

[Insert Signature Block of Customer Developer Representative] Date

Revision History

Revision Number	Release Date	Description
DR	2004 Jun 7	Draft
00	2004 Aug 16	final for cycle 3
01	2004 Oct 12	draft for cycle 4
02	2004 Nov 30	final for cycle 4
03	2005 Jan 21	draft for cycle 5
04	2005 Feb 14	final for cycle 5
05	2005 Mar 21	draft for cycle 6
06	2005 Apr 27	final for cycle 6
07	2005 Jul 15	final for cycle 7
08	2005 Sep 13	final for cycle 8
09	2005 Oct 18	final for cycle 9
10	2005 Dec 06	draft for cycle 10
11	2006 Jan 22	revisions for CDR

Referenced Documents

Internal Documents

Reference	Title
PSCD-230-001	PS-1 Design Reference Mission
PSDC-430-004	Pan-STARRS PS-1 IPP C Code Conventions
PSDC-430-005	Pan-STARRS PS-1 IPP Software Requirements Specification
PSDC-430-006	Pan-STARRS PS-1 IPP Algorithm Design Document
PSDC-430-011	Pan-STARRS PS-1 IPP System/Subsystem Design Description

External Documents

Reference	Title
Posix Standard	Open Group Based Specifications Issue 6, IEEE Std 1003.1, 2003

Contents

List of Figures

1 Introduction

This document describes the Pan-STARRS Image Processing Pipeline (IPP) data analysis Modules. The Modules use the functionality of the Pan-STARRS Library (PSLib) to perform more complex tasks, especially tasks which require assumptions of astronomical analysis or the data organization. Within the IPP, the Modules are tied together into programs which perform complete data analysis tasks (an "analysis stage"). The modules may be tied together within a C framework or using a high-level scripting language. Bindings of the Modules are made available to the scripting language using the program SWIG.

In order to preserve name space, globally-visible structures and functions shall be prefixed with `pm`, for "Pan-STARRS Modules".

2 Runtime Configuration Data

PSLib defines a `psMetadata` structure which can carry labeled data of arbitrary types. The associated functions implemented by PSLib consist of tools to manipulate and extract data from `psMetadata` collections. A particular application of the `psMetadata` structure within PSLib is to carry the data from a FITS header. Other general-purpose information is also carried with the structure. Functions are available to fill a `psMetadata` collection from a text-based configuration file using a human-readable syntax, and to fill a `psMetadata` collection from a properly formatted XML document.

In the IPP Modules, we use `psMetadata` collections to carry run-time configuration data used by the data analysis modules. Below, in the discussion of the various modules, this configuration information is defined by specifying the name of the data item of interest, the conceptual meaning of that data item, and the allowed values for the data item. In this section, we discuss top-level concepts related to the configuration information, including the sources of the run-time configuration data and special operations used to extract information from the configuration system.

2.1 Configuration Data Sources

All modules need to load some configuration information defining parameters which may be configured at run-time. We break these parameters down into three levels:

- Options for the particular site installation of the pipeline: the *site*;
- Options specifying the instrument setup, and in particular the format of the FITS file: the *camera*; and
- Options specifying the particular parameter choices that affect the details of an analysis: the *recipe*.

Note that these are arranged in an hierarchical order, with the site configuration being the most general, and the recipe configuration the most specific. For example, not all sites will have to deal with all cameras, and different cameras may require different recipes at different times according to their particular quirks, analysis experimentations, or their evolution.

Each of the levels will have a metadata configuration file. In the case of the site configuration, the filename shall be that specified by the `-site` option on the command line if provided, the environment variable `PS_SITE`, if defined, or `~/.ipprc` otherwise. The camera configuration shall be specified by the `-camera` option on the command line if provided, or shall be inferred from a FITS header (more detail below). The recipe configuration shall be specified by the `-recipe` option on the command line if provided, or from the the camera configuration (more detail below).

2.2 Configuration Files

2.2.1 Site Configuration

The site configuration file must contain the following:

- The database configuration:
  - `DBSERVER` of type `STR`: The database host name for `psDBInit`.
  - `DBUSER` of type `STR`: The database user name for `psDBInit`.
  - `DBPASSWORD` of type `STR`: The corresponding database password for `psDBInit`.
- `CAMERAS` of type `METADATA`: A list of instruments that the system understands. Cameras are specified as separate metadata entries, with the name of the camera as the key, and the filename of the camera configuration file (of type `STR`) as the data.

and may also contain the following psLib configuration options:

- `TIME` of type `STR`: The time configuration file (for `psTimeInitialize`).
- `LOGLEVEL` of type `S32`: The log level for `psLogSetLevel`.
- `LOGFORMAT` of type `STR`: The log format for `psLogSetFormat`.
- `LOGDEST` of type `STR`: The log destination for `psLogSetDestination`.
- `TRACE` of type `METADATA`: A list of components with the desired trace level (of type `S32`) for each.

No doubt there is a need for better security than storing the database password directly in the file, but we push this problem onto the stack for now. (TBD)

We will add other data sources in the future, e.g., file paths, configuration for Nebulous and DVO, etc. (TBD) .

An example site configuration file:

```
### Example .ipprc file

### Database configuration
DBSERVER        STR     ippdb.ifa.hawaii.edu    # Database host name (for psDBInit)
DBUSER          STR     ipp                     # Database user name (for psDBInit)
DBPASSWORD      STR     password                # Database password (for psDBInit)

### Setups for each camera system
CAMERAS         METADATA
        MEGACAM_RAW     STR     megacam_raw.config
        MEGACAM_SPLICE  STR     megacam_splice.config
        GPC1_RAW        STR     gpc1_raw.config
        LRIS_BLUE       STR     lris_blue.config
        LRIS_RED        STR     lris_red.config
END

### psLib setup
TIME            STR     time.config             # Time configuration file
LOGLEVEL        S32     3                       # Logging level; 3=INFO
LOGFORMAT       STR     THLNM                   # Log format
```

```
LOGDEST         STR     STDOUT                  # Log destination
TRACE           METADATA                        # Trace levels
        psLib.math.psPolynomial         S32     6
        psLib.image.psImageConvolve     S32     2
END
```

2.2.2 Camera Configuration

The camera configuration is somewhat complicated and involved, since it must not only specify how to translate the pixels from a FITS file into a focal plane hierarchy (§3), but it must also specify how to derive the various values the IPP needs (§2.3). Moreover, it must be able to do these for the great variety of cameras in use in the astronomical community.

Example camera configuration files are included in an appendix, but below we explain the components.

2.2.2.1 FITS File to Focal Plane Hierarchy

The Focal Plane hierarchy (`pmFPA, pmChip, pmCell, pmReadout`) is explained in more detail in §3. The top level, an FPA contains one or more chips, which correspond to a contiguous piece of silicon. A chip contains one or more cells, which correspond to a single amplifier. A cell contains one or more readouts, which correspond to individual reads of the detector.

The FITS data storage formation is a standard in the astronomical community for storing astronomical images. A FITS file consists of an arbitrary number of coupled human readable `ASCII` header segments and binary data segments. The headers describe the format and layout of the data segments. The first of these groups is traditionally called the 'primary header unit' (PHU) and the rest are referred to as 'extensions'. The header segments may contain extensive documentary information related to the interpretation of the data. Although the FITS format defines a standard representation of the data, the header metadata is not so consistently defined within the astronomical community. Also, the flexibility of the data format means that different representations are possible for the same fundamental collection of data. The tools presented in this section provide a method to define and constrain the wide range of possible FITS representations of astronomical images.

Within the FITS data representation, there are various choices which can and have been made for the placement of the pixels in the file. In the simplest case, the camera consists of a single chip consisting of a single cell always read with a single readout. In this case, the image data could be written as part of the primary data unit. In a more complex case with multiple chips and multiple cells, the data may be organized in several ways. The data may be distributed into multiple files or in multiple FITS data extensions. A single camera image may be written as a collection of files for individual chips with separate extensions for each cell (CFH12K.split, GPC). Another camera may write a single file with multiple extensions for each cell (Megacam.raw), or multiple extensions per chip, with each cell representing portions of the chip image (Megacam.splice, CFHT-IR).

In all of these representations, there are only two basic distinctions in how the pixel data is stored: what level in the hierarchy the entire FITS file corresponds to (FPA, chip, or cell), and what level the extensions correspond to (chip, cell or no extensions at all). Knowing these, and having a list of the components, we can construct the focal plane hierarchy.

Note that a single data extension, consisting of a uniform grid of pixels, can only naturally represent a cell or a chip. In order to represent the entire focal plane array as a single grid, some artificial choices would be made to fill-in or ignore the gaps between chips and their relative rotations. Within our framework, a complete focal plane mosaic of multiple chips could be represented as a single extension by treating the collection of pixels as if they were from a single chip.

To define the hierarchy, we specify the following keywords:

- PHU of type STR: May be one of FPA, CHIP, or CHIP. This specifies the focal plane level of the Primary Header Unit, and hence the entire FITS file (the 'class' of the file) .
- EXTENSIONS of type STR: May be one of CHIP, CELL or NONE, though not of a level higher than that specified by the PHU. This specifies what each extension represents.
- CONTENTS which may be of type METADATA or STR, depending upon the PHU and EXTENSIONS, specifies what the contents of the FITS file are:
  - PHU=FPA, EXTENSIONS=CHIP: Type METADATA with the component keywords being the extension names and the values the names of the cells, separated by commas or whitespace.
  - PHU=FPA, EXTENSIONS=CELL: Type METADATA with the component keywords being the extension names and the values the chip name and the cell type, separated by a colon.
  - PHU=FPA, EXTENSIONS=NONE: Type METADATA with the component keywords being the chip names and the values the names of the cells, separated by commas or whitespace.
  - PHU=CHIP, EXTENSIONS=CELL: Type METADATA with the component keywords being the extension names and the values the corresponding cell type.
  - PHU=CHIP, EXTENSIONS=NONE: Type STR with the value being the cell types separated by commas or whitespace.
- CELLS of type METADATA with the component keywords being the cell names or types, each of type METADATA. Within each cell should be specified various PS concept values appropriate for each cell.

An example:

```
# How to read this data
PHU             STR     FPA     # The FITS file represents an entire FPA
EXTENSIONS      STR     CELL    # The extensions represent cells

# What's in the FITS file?
CONTENTS        METADATA
        # Extension name, chip name:type
        amp00   STR     ccd00:left
        amp01   STR     ccd00:right
        amp02   STR     ccd01:left
        amp03   STR     ccd01:right
        amp04   STR     ccd02:left
END

# Specify the cell data
CELLS   METADATA
        left    METADATA        # Left amplifier
                CELL.BIASSEC            STR     BIASSEC
                CELL.TRIMSEC            STR     DATASEC
                CELL.PARITY             S32     1
        END
        right   METADATA        # Right amplifier
                # This cell is read out in the opposite direction
                CELL.BIASSEC            STR     BIASSEC
                CELL.TRIMSEC            STR     [1025:2048,1:2048]
                CELL.PARITY             S32     -1
        END
END
```

Observe how the CONTENTS specifies the extension name, which we know from the EXTENSIONS is a cell, and that each extension is associated with a chip, and has a cell type.

2.2.2.2 Deriving concept values

The PS concepts are described in more detail in §2.3. Basically, astronomical cameras generally store the important details ("concepts") in different ways. This is generally manifested in the choice of different FITS header keywords to describe the same concept, but one can also imagine deriving values from a database or a known default.

We therefore specify the following keywords:

- `TRANSLATION` of type `METADATA` is a translation table for understanding PS concepts in terms of FITS headers. The PS concept (keyword) is derived from the FITS header given in the value.
- `DATABASE` of type `METADATA` is a formula for obtaining a PS concept from the database. Each component is of a user-specified type containing `TABLE`, `COLUMN`, `GIVENDBCOL` and `GIVENPS`. The idea is that to obtain the value of a PS concept, one refers to a particular `COLUMN` in a particular `TABLE`, where the value of certain PS concepts (`GIVENPS`; multiple values separated by a comma or semicolon) match certain database columns (`GIVENDBCOL`; multiple values separated by a comma or semicolon).
- `DEFAULTS` of type `METADATA` is a set of default values of PS concepts for the camera. The PS concept (keyword) is assigned the value. There is also limited dependency allowed; see §2.3.

An example:

```
# How to translate PS concepts into FITS headers
TRANSLATION     METADATA
        FPA.NAME        STR     EXPNUM
        FPA.AIRMASS     STR     AIRMASS
        FPA.FILTER      STR     FILTER
        FPA.POSANGLE    STR     ROTANGLE
        FPA.RA          STR     RA
        FPA.DEC         STR     DEC
        FPA.RADECSYS    STR     RADECSYS
        FPA.MJD         STR     MJD-OBS
        CELL.EXPOSURE   STR     EXPTIME
        CELL.DARKTIME   STR     DARKTIME
        CELL.XBIN       STR     CCDBIN1
        CELL.YBIN       STR     CCDBIN2
        CELL.SATURATION STR     SATURATE
END

# Default PS concepts that may be specified by value
DEFAULTS        METADATA
        CELL.BAD                S32     0
        CELL.PARITY.DEPEND      STR     CHIP.NAME
        CELL.PARITY    METADATA
                amp00   S32     1
                amp01   S32     -1
                amp02   S32     1
                amp03   S32     -1
        END
END

# How to translate PS concepts into database lookups
DATABASE        METADATA
        TYPE            dbEntry         TABLE           COLUMN          GIVENDBCOL      GIVENPS
        CELL.GAIN       dbEntry         Camera          gain            chipId,cellId   CHIP.NAME,CELL.NAME
        CELL.READNOISE  dbEntry         Camera          readNoise       chipId,cellId   CHIP.NAME,CELL.NAME
END
```

The `.DEPEND` entry in the `DEFAULTS` will be explained in §2.3.

2.2.2.3 Indentification by rule

The function `pmConfigCameraFromHeader` requires that the camera configuration also contains a rule on how to recognise that a FITS header comes from that camera.

We therefore specify another keyword: `RULE` of type `METADATA`: Contains a list of FITS headers keywords and values (of the appropriate type) against which actual headers are compared to determine if it matches the camera type.

An example is:

```
# How to identify this type
RULE    METADATA
        TELESCOP        STR     CFHT 3.6m
        DETECTOR        STR     MegaCam
        EXTEND          BOOL    T
        NEXTEND         S32     72
END
```

2.2.2.4 Recipes

The camera configuration file must also contain filenames for the recipe configuration files. We include `RECIPES` of type `METADATA` with component keywords being the various recipe names and the values (of type `STR`) the corresponding recipe configuration filename.

An example:

```
# Recipes for LRIS
RECIPES METADATA
        PHASE1          STR     lris_phase1.config
        PHASE2          STR     lris_phase2.config
PHASE4          STR     lris_phase4.config
END
```

2.2.3 Recipe Configuration

The contents of the recipe configuration file are dependent upon the particular module, and hence are not specified here at this time. (TBD)

2.3 PS Concepts

Each image has associated with it what we will call *concepts* (for want of a better word). These are values corresponding to general quantities and qualities necessary to understand and interpret the data such as airmass, date, read noise and filter. The values of each of the below concepts shall be determined when the FPA is read into memory (via `pmFPARead`), and stored at the appropriate level in the focal plane hierarchy.

Below is a list of concepts that the IPP requires, with the expected type and a short description.

- `FPA.AIRMASS` (F32): Airmass at which the observation is made (boresight).
- `FPA.FILTER` (STR): Filter used in observation

- `FPA.POSANGLE` (F32): Position angle for camera
- `FPA.RA` (F64): Right Ascension of boresight in radians
- `FPA.DEC` (F64): Declination of boresight in radians
- `FPA.RADECSYS` (STR): System of RA,Dec (e.g., J2000 or ICRS)
- `FPA.NAME` (STR): An identifier (e.g., observation number) for the FPA instance
- `CHIP.NAME` (STR): The name of the chip (unique within the FPA) — set at FITS read
- `CELL.NAME` (STR): The name of the cell (unique within the parent chip) — set at FITS read
- `CELL.TIME` (`psTime`): Time of observation start
- `CELL.READDIR` (S32): Read direction: line (1) or column (2)
- `CELL.BIASSEC` (STR): Overscan region(s)
- `CELL.TRIMSEC` (STR): Trim region
- `CELL.GAIN` (F32): CCD gain (e/ADU)
- `CELL.READNOISE` (F32): CCD read noise (e)
- `CELL.SATURATION` (F32): CCD saturation point (ADU)
- `CELL.BAD` (F32): CCD bad pixel point (ADU)
- `CELL.XBIN` (S32): CCD binning in x
- `CELL.YBIN` (S32): CCD binning in y
- `CELL.XPARITY` (S32): Direction of CCD readout in x relative to the rest of the chip
- `CELL.YPARITY` (S32): Direction of CCD readout in y relative to the rest of the chip
- `CELL.EXPOSURE` (F32): Exposure time of image (sec)
- `CELL.DARKTIME` (F32): Dark time for image (sec)

Note that CELL.EXPOSURE, CELL.DARKTIME and CELL.TIME should actually be specified at the readout level. However, at this present time, we're not sure how these should be specified, and so we move them up to the cell level and assume that all readouts are of the same exposure and dark time. (TBD)

For different camera systems, these concepts are not always known by the same name, nor are they generally obtained in the same manner, and so their source or value must be specified in the camera configuration file. The value of a concept shall be found by searching in the following order:

- The cell data from the `CELLS` metadata in the camera configuration.
- The FITS header via the `TRANSLATION` table.
- The `DATABASE` lookup.
- The `DEFAULTS` value.

After ingest (performed by `pmFPARead`, the user may safely assume that all of the above concepts exist (defined at the appropriate level), is of the specified type and in the specified format.

2.3.1 Dependencies for defaults

In the DEFAULTS table in the camera configuration, we allow the specification of the concept with an additional suffix, DEPEND. The value (of type STR) of the CONCEPT.DEPEND is the name of a concept on which the first concept depends. For example, it might depend on the chip name. Then the first concept becomes of type METADATA, with the component keywords being the value of the second concept (on which the first depends). To avoid infinite recursion, no further dependency is permitted. We also allow an entry CONCEPT.DEFAULT specifiying the default value of the concept if a match is not made with the dependcency list. An example of the dependency:

```
# Default PS concepts that may be specified by value
DEFAULTS        METADATA
        CELL.GAIN.DEPEND     STR     CHIP.NAME
        CELL.GAIN.DEFAULT    STR     1.0
        CELL.GAIN    METADATA
                ccd00   F32     1.2
                ccd01   F32     3.4
                ccd02   F32     5.6
        END
END
```

2.3.2 FORMATS

Because of the variety of methods for specifying these concepts (especially in FITS headers), we must also specify additional information in the camera configuration that specifies how to interpret the data provided. These are provided in an entry FORMATS (of type METADATA) in the camera configuration. Within the FORMATS metadata, there is a string for each of the concepts that requires a format to be specified.

2.3.2.1 CELL.TIME

The time at which the shutter opens is represented in a variety of ways in FITS files, so care must be taken to specify what the format is in the file under consideration. Permitted values of CELL.TIME.FORMAT are:

- JD: The value pointed to by CELL.TIME is to be interpreted as a Julian Date.
- MJD: The value pointed to by CELL.TIME is to be interpreted as a Modified Julian Date.
- ISO: The value pointed to by CELL.TIME is to be interpreted as an ISO date-time (yyyy-mm-ddThh:mm:ss.ss).
- SEPARATE: The date and time are specified separately, and the CELL.TIME contains the headers for the date and the time separated by whitespace or a comma. Then it is necessary to add additional qualifiers to specify the formats of these:
  - PRE2000: The year is in the old style two-digit format popular before the year 2000, and it should be assumed that the date is in the twentieth century.
  - BACKWARDS: The date is in the format dd-mm-yyyy or dd/mm/yyyy.
  - SOD: The time is specified as seconds-of-day.

Note that the FITS standard is that the time in the header refers to the *start* of the observation.

the PRE2000 and BACKWARDS qualifiers should be replace with explicit format definitions in the form YYYY/MM/DD (TBD)

In the future, we might add additional qualifiers that calculate the start time of the observation based on someone foolishly putting the end- or mid-time in the header. (TBD)

Should we move CELL.TIMESYS into the format as well? (TBD)

2.3.2.2 FPA.RA and FPA.DEC

The RA and Declination of the boresight might be specified in a few ways. We need to specify both how the value is interpreted and the units. `FPA.RA.FORMAT` and `FPA.DEC.FORMAT` should be one of the following:

- `HOURS`: The value pointed to by the concept should be interpreted as being in hours.
- `DEGREES`: The value pointed to by the concept should be interpreted as being in degrees.
- `RADIANS`: The value pointed to by the concept should be interpreted as being in radians.

How the value is interpreted can be determined from the type of the header: if it is of type `STR`, then we can reasonably assume that it is in sexagesimal format with colons or spaces as separators; and if it is of type `F32` (or `F64`), then we can assume that it is in decimal format.

2.3.3 Implicit format information

While details like the units of the right ascension in the header must be specified explicitly, some other details can be determined from implicit information.

- `FPA.RA` and `FPA.DEC`: if the value on ingest is of type `STRING`, then it may be interpreted as sexagesimal notation, "`dd:mm:ss.ss`", or "`dd:mm.mmm`". A space may be used instead of a colon to separate the values. Otherwise, if the value is of a numerical type (`F32` or `F64`), then that is the appropriate value.
- `CELL.XBIN` and `CELL.YBIN`: if the value on ingest is of type `STRING`, then it may be interpreted as "`x,y`", where `x` is the binning in x, and `y` is the binning in y. A space may be used instead of a comma, and there may even be a space before or after the comma (or both). Otherwise, if the value is of a numerical type (`S32`, etc), then that is the appropriate value.
- `CELL.BIASSEC` and `CELL.TRIMSEC`: These values on ingest should always be of type `STRING`. If they contain a square bracket, then they may be interpreted as a list of standard region specifications, "`[x0:x1,y0:y1];[x2:x3,y2:y3];...`", where the semi-colon may be replaced by spaces. Otherwise, the string may be interpreted as a FITS header (or headers, separated by spaces, commas or semi-colons) that contains the appropriate values.

the use of implicit interpretation of formats should be discouraged: format interpretation guides should be provided (TBD)

2.4 Configuration APIs

```
bool pmConfigRead(psMetadata **site, psMetadata **camera, psMetadata **recipe,
                  int *argc, char **argv, const char *recipeName);
psMetadata *pmConfigCameraFromHeader(const psMetadata *site, const psMetadata *header);
psMetadata *pmConfigRecipeFromCamera(const psMetadata *camera, const char *recipeName);
```

`pmConfigRead` shall load the `site` configuration (according to the above rule for determining the source). The `camera` configuration shall also be loaded if it is specified on the command line (`argc, argv`); otherwise it shall be set to `NULL`. The `recipe` shall also be loaded from the command line (if specified) or, if the camera configuration has been loaded, from the camera configuration and recipe specification therein (see below). In dealing with the command line parameters, the functions shall use the appropriate functions in psLib to retrieve and remove the relevant options from the argument list; this simplifies assignment of the mandatory arguments, since all the optional command line arguments are removed leaving only the mandatory arguments. The following psLib setups shall also be performed if they are specified in the site configuration:

- the function shall call `psTimeInitialize` with the configuration file specified by `TIME`.
- the function shall call `psLogSetLevel` with the logging level specified by `LOGLEVEL`.
- the function shall call `psLogSetFormat` with the log format specified by `LOGFORMAT`.
- the function shall call `psTraceSetLevel` with the component names and trace levels specified by the `TRACE`.

Note that additional log/trace command-line options may be specified and interpretted using the `psArgumentVerbosity` function from psLib. These options should (in the case of logging) override the configuration-supplied information or (in the case of tracing) supplement it.

`pmConfigCameraFromHeader` shall load the `camera` configuration based on the contents of the FITS `header`, using the list of known cameras contained in the `site` configuration. If more than one camera matches the FITS header, a warning shall be generated and the first matching camera returned.

`pmConfigRecipeFromCamera` shall load the `recipe` configuration based on the `recipeName` and the list of known recipes contained in the `camera` configuration (details below).

```
bool pmConfigValidateCamera(const psMetadata *camera, const psMetadata *header);
```

This function, used by `pmConfigCameraFromHeader`, shall return `true` if the FITS `header` matches the rule contained in the `camera` configuration (see §2.2.2.3); otherwise it shall return `false`.

```
psDB *pmConfigDB(psMetadata *site);
```

`pmConfigDB` shall use the `site` configuration data to open a database handle. **This is fairly straightforward at the moment, but will change when we beef up security. (TBD)**

2.4.1 Example usage

The following is provided as an example of how the above functions are envisioned in use.

```
int main(int argc, char *argv[])
{
    // Parse other command-line arguments here
    psMetadata *site = NULL;            // Site configuration
    psMetadata *camera = NULL;          // Camera configuration
    psMetadata *recipe = NULL;          // Recipe configuration
    if (! pmConfigRead(&site, &camera, &recipe, &argc, argv, "moduleName")) {
        psLogMsg("moduleName", PS_LOG_ERROR, "Can't find site configuration!\n");
        exit(EXIT_FAILURE);
    }
    // Parse other command-line arguments here

    // The command-line argument list now contains only mandatory arguments
    // Assume the first of these is an input image
    char *imageName = argv[1];          // Name of FITS file
    psFits *imageFH = psFitsOpen(imageName, "r"); // File handle for FITS file
    if (! imageFH) {
        psLogMsg("moduleName", PS_LOG_ERROR, "Can't open input image %s\n", imageName);
        exit(EXIT_FAILURE);
    }
    psMetadata *header = psFitsReadHeader(NULL, imageFH); // FITS header

    if (!camera && !(camera = pmConfigCameraFromHeader(site, header))) {
        psLogMsg("moduleName", PS_LOG_ERROR, "Can't find camera configuration!\n");
        exit(EXIT_FAILURE);
    }

    if (! recipe && !(recipe = pmConfigRecipeFromCamera(camera, "moduleName"))) {
        psLogMsg("moduleName", PS_LOG_ERROR, "Can't find recipe configuration!\n");
        exit(EXIT_FAILURE);
    }

    // Now go on and do stuff
    ....
}
```

3 Focal Plane

3.1 Overview

In PSLib, we have defined a basic container for a single 2D collection of pixels (`psImage`), along with basic operations to manipulate the image pixels. For astronomical applications, this data structure is insufficient for two reasons. First, it does not provide sufficient additional metadata to describe the data in detail. Second, astronomy applications frequently involve multiple, related images. For Pan-STARRS, and for general astronomical applications, we require a richer collection of data structures which describe a very general image concept. We have defined several layers in the hierarchy which are necessary to describe the image data which will be produced by the Pan-STARRS GigaPixel Cameras as well as other standard astronomical images.

A simple 2D image is the basic data unit for much of astronomical imaging: if we consider various optical and IR array cameras, a single readout of the detector produces a collection of pixel measurements which is well represented as a single 2D image. We define our lowest-level astronomical image structure, `pmReadout`, to contain the pixels produced by a single readout of the detector, along with metadata needed to define that readout: the origin and binning of the image

relative to the original detector pixels explicitly in the structure, and pointers to the general metadata and derived objects, if any.

A single detector may be read multiple times in sequence. For example, infrared detectors frequently produce an image immediately after the detector is reset followed by an image after the basic exposure is complete. Both readouts correspond to the same pixels, though the binning or rastering may be different between the two readouts. Another example is the video sequence produced by the Pan-STARRS GigaPixel Camera guide cells, each of which represents a series of many images from a subraster of pixels in the detector readout portion. The second level of our image container hierarchy, `pmCell`, consists of a collection of readouts from a single detector.

In the Pan-STARRS GigaPixel camera, the basic readout region is a fraction of the full imaging area of a single CCD chip. The chip is divided into 64 cells, any fraction of which may have been readout for a given exposure. In other cameras, such as Megacam at CFHT, the individual CCDs have multiple amplifiers addressing contiguous portions of the detector. In such cameras, each amplifier produces a separate collection of pixels. In the third level of our image container hierarchy, the data structure `pmChip` represents a collection of different cells.

The top level of our image container hierarchy is a complete focal plane array (`pmFPA`). This structure represents the collection of chips in the camera, all of which are read out in a given exposure.

For example, take a mosaic camera consisting of eight $2k \times 4k$ CCDs, each of which is read out through two amplifiers. Then there would be sixteen cells in total, each of which is presumably $1k \times 4k$. There would be eight chips, each consisting of two cells, and the focal plane consists of these eight chips.

As another example, consider an observation by PS-1. The focal plane would consist of 60 chips, each of which consist of 64 cells (or less; a few cells may be dead). Some cells (those containing guide stars for the orthogonal transfer) will contain multiple readouts.

These data structures represent containers with which to carry around the collection of related image data. There is no requirement on the functions or the structures that each instance of one of these data structures represent the physical hardware. For example, it is not necessary that an instance of `pmFPA` always carry the data for all 60 GigaPixel Camera OTAs. The usage of these structures is such that all astronomical operations which apply to a CCD image should be performed on an instance of `pmFPA`. If a particular circumstance only requires a single 2D image, then that is represented by an instance of `pmFPA` with one `pmChip`, which in turn has one `pmCell`, which in turn has one `pmReadout`.

The data structures defined below provide two additional features beyond the hierarchy of relationships. First, each level of the hierarchy includes hooks for carrying metadata to provide the PS concepts and analysis metadata that would be appropriate for that level. The functions within PSLib do not specify the contents of those metadata containers.

While the `psMetadata` pointers provide a mechanism to carry generic information about the image, the hierarchy of data structures also provides an explicit set of information defining the geometrical relationships between the levels of the hierarchy. Two types of information are provided. In the first case, basic offsets (and in the case of the readouts, binning and flips) are defined to specify the location of a given `pmCell` with respect to its containing `pmChip` in the assumption that the pixels in the entire focal plane array are laid out on a uniform grid. This is a crude approximation, and cannot be assumed for careful astrometric analysis, but it can be used as a starting point or to place the the pixels in a test image. For higher precision, detailed astrometric transformations between one frame and the next are also provided.

In the future, it may be worthwhile to migrate all of these additional pieces to the psMetadata since there is no pressing need to have them visible in the data structures (TBD)

3.2 Image Data Container Hierarchy

Here we specify the contents of the focal plane hierarchy: `pmReadout`, `pmCell`, `pmChip` and `pmFPA`. Many of the components of these are similar. All but the `pmFPA` contain offsets from the level above (`col0,row0`), and a link to the parent. All but the `pmReadout` contain a private pointer to FITS data and more detailed astrometric transforms. Each contains an `analysis` metadata container which is intended to store results of analyses (e.g., the r.m.s. of the overscan fit).

At what stage are the offsets (`col0,row0`) set, and how are they known? (TBD)

3.2.1 A Readout

A readout is the result of a single read of a cell (or a portion thereof). It contains the offset from the lower-left corner of the chip, in the case that the CCD was windowed, as well as the binning factors and parity (if the binning value is negative, then the parity is reversed). It also contains the pixel data (with corresponding mask and weight), metadata container for the analysis, and a link to the parent.

```
typedef struct {
    // Position on the cell
    int col0;                           // Offset from the left of cell.
    int row0;                           // Offset from the bottom of cell.
    int colBins;                        // Amount of binning in x-dimension and parity (from sign)
    int rowBins;                        // Amount of binning in y-dimension and parity (from sign)
    // Information
    psImage *image;                     // Imaging area of readout
    psImage *mask;                      // Mask for image
    psImage *weight;                    // Weight for image
    psList *bias;                       // List of bias section (sub-)images
    psMetadata *analysis;               // Readout-level analysis metadata
    pmCell *parent;                     // Parent cell
} pmReadout;
```

The constructor for `pmReadout` shall be:

```
pmReadout *pmReadoutAlloc(pmCell *cell);
```

The constructor shall make an empty `pmReadout`. If the parent `cell` is not `NULL`, the `parent` link is made and the readout shall be placed in the parent's array of `readouts`. The metadata containers shall be allocated. All other pointers in the structure shall be initialized to `NULL`.

3.2.2 A Cell

A cell consists of one or more readouts (usually only one except in the case that the cell has been used for fast guiding, or similar situations). It has values which specifies the position of the cell on the chip for rough positioning, along with more precise coordinate transforms from the cell to the chip and, as a convenience, from the cell directly to the focal plane. It is expected that these transforms will consist of two first-order 2D polynomials, simply specifying a translation, rotation and magnification; hence they are easily inverted, and there is no need to add reverse transformations. We also add an additional transformation, which is intended to provide a "quick and dirty" transform from the cell coordinates to the sky; this transformation not guaranteed to be as precise as the "standard" transformation of Cell → Chip → Focal Plane → Tangent Plane → Sky, but will be faster. The cell also contains metadata containers for the concepts and analysis, a link

to the parent, and a container for the FITS data, if that corresponds to this level. A boolean indicates whether the cell is of interest, allowing it to be excluded from analysis.

```
typedef struct {
    // Offset specifying position on chip
    int col0;                           // Offset from the left of chip.
    int row0;                           // Offset from the bottom of chip.
    // Astrometric transformations
    psPlaneTransform* toChip;           // Transformations from cell to chip coordinates
    psPlaneTransform* toFPA;            // Transformations from cell to FPA coordinates
    psPlaneTransform* toSky;            // Transformations from cell to sky coordinates
    // Information
    psMetadata *concepts;               // Cache for PS concepts
    psMetadata *camera;        // Camera information
    psMetadata *analysis;               // Cell-level analysis metadata
    psArray *readouts;                  // The readouts (referred to by number)
    pmChip *parent;                     // Parent chip
    bool valid;                         // Do we bother about reading and working with this cell?
    p_pmHDU *hdu;        // FITS data
} pmCell;
```

The constructor for `pmCell` shall be:

```
pmCell *pmCellAlloc(pmChip *chip, psMetadata *cameraData, psString name);
```

The constructor shall make an empty `pmCell`. If the parent `chip` is not `NULL`, the `parent` link is made and the cell shall be placed in the parent's array of `cells`. The `readouts` array shall be allocated with a zero size, and the metadata containers constructed. The cell's `camera` pointer shall be set to the provided `cameraData`, and the `name` shall be used to set `CELL.NAME` in the `concepts`. All other pointers in the structure shall be initialized to `NULL`.

3.2.3 A Chip

A chip consists of one or more cells (according to the number of amplifiers on the device). The chip contains metadata containers for the concepts and analysis, a link to the parent, and pointers to the pointers to the various FITS data, if that corresponds to this level. For astrometry, in addition to the rough positioning information, it contains a coordinate transform from the chip to the focal plane. It is expected that this transform will consist of two second-order 2D polynomials; hence we think that it is prudent to include a reverse transformation which will be derived from numerically inverting the forward transformation. A boolean indicates whether the chip is of interest, allowing it to be excluded from analysis.

```
typedef struct {
    // Offset specifying position on focal plane
    int col0;                           // Offset from the left of FPA.
    int row0;                           // Offset from the bottom of FPA.
    // Astrometric transformations
    psPlaneTransform* toFPA;            // Transformation from chip to FPA coordinates
    psPlaneTransform* fromFPA;          // Transformation from FPA to chip coordinates
    // Information
    psMetadata *concepts;               // Cache for PS concepts
    psMetadata *analysis;               // Chip-level analysis metadata
    psArray *cells;                     // The cells (referred to by name)
    pmFPA *parent;                      // Parent FPA
    bool valid;                         // Do we bother about reading and working with this chip?
    p_pmHDU *hdu; // FITS data
} pmChip;
```

The constructor for `pmChip` shall be:

```
pmChip *pmChipAlloc(pmFPA *fpa, psString name);
```

The constructor shall make an empty `pmChip`. If the parent `fpa` is not NULL, the `parent` link is made and the chip shall be placed in the parent's array of `chips`. The `cells` array shall be allocated with a zero size, and the metadata containers constructed. The `name` shall be used to set `CHIP.NAME` in the `concepts`. All other pointers in the structure shall be initialized to `NULL`.

3.2.4 A Focal Plane

A focal plane consists of one or more chips (according to the number of pieces of contiguous silicon). It contains metadata containers for the concepts and analysis, a link to the parent, and pointers to the FITS header, if that corresponds to this level (the FPA may be the PHU, but will not ever contain pixels). For astrometry, it contains a transformation from the focal plane to the tangent plane and the fixed pattern residuals. It is expected that the transformation will consist of two 4D polynomials (i.e. a function of two coordinates in position, the magnitude of the object, and the color of the object) in order to correct for optical distortions and the effects of the atmosphere; hence we think that it is prudent to include a reverse transformation which will be derived from numerically inverting the forward transformation.

```
typedef struct {
    // Astrometric transformations
    psPlaneDistort* fromTangentPlane;  // Transformation from tangent plane to focal plane
    psPlaneDistort* toTangentPlane;    // Transformation from focal plane to tangent plane
    psProjection *projection;          // Projection from tangent plane to sky
    // Information
    psMetadata *concepts;              // Cache for PS concepts
    psMetadata *analysis;              // FPA-level analysis metadata
    const psMetadata *camera;          // Camera configuration
    psArray *chips;                    // The chips
    p_pmHDU *hdu;        // FITS data
    psMetadata *phu;        // Primary Header
} pmFPA;
```

The constructor for `pmFPA` shall be:

```
pmFPA *pmFPAAlloc(const psMetadata *camera);
```

The constructor shall make an empty `pmFPA`. The `chips` array shall be allocated with a zero size, the `camera` and `db` pointers set to the values provided, and the `concepts` metadata constructed. All other pointers in the structure shall be initialized to `NULL`.

The inclusion of hierarchical links pointing both down (via the arrays) and up (via the `parent`) could make for difficulties. For this reason, we specify a utility function to manage the collection of upward-pointing links:

```
bool pmFPACheckParents(pmFPA *fpa);
```

This function checks the validity of the `parent` links in the FPA hierarchy. If a `parent` link is not set (or not set correctly), it is corrected, and the function shall return `false`. If all the `parent` pointers were correct, the function shall return `true`.

Each of the levels in the hierarchy have a place to hold a `p_pmHDU`, which is the disk representation of the image:

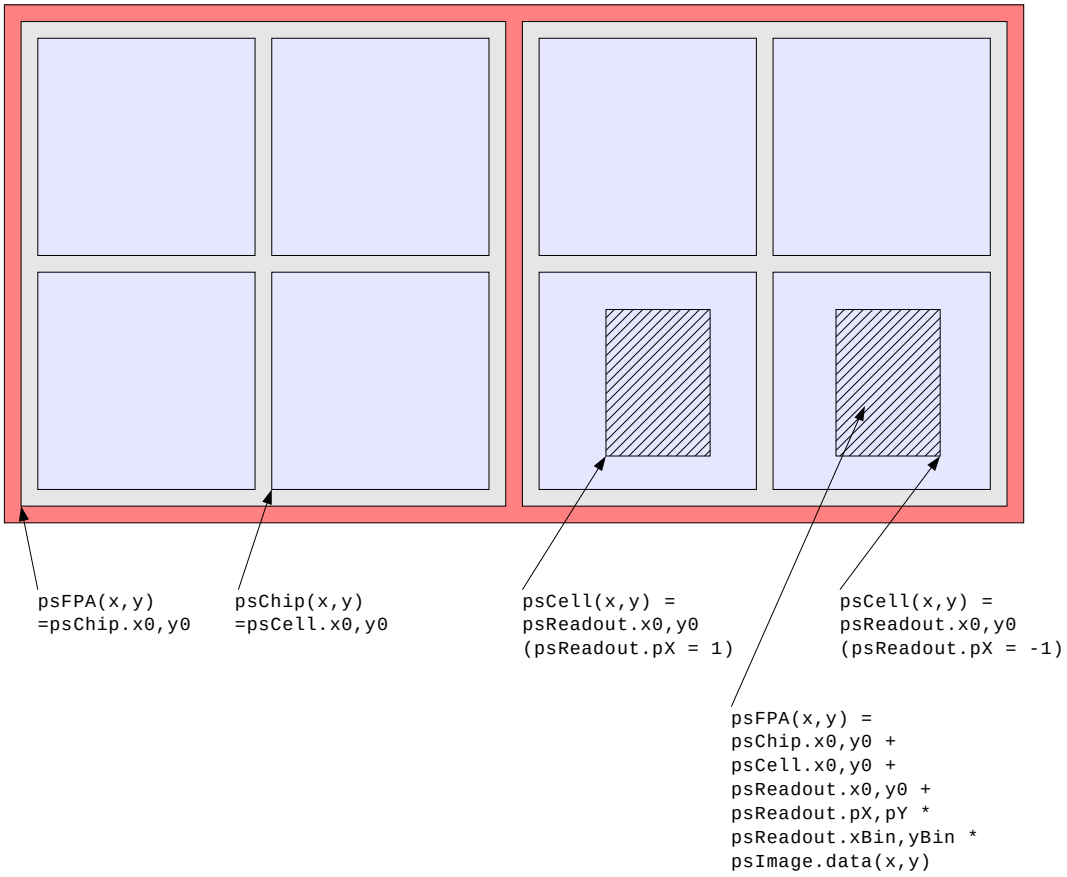

Figure 1: Camera Pixel Layout

```
typedef struct {
    const char *extname;                // Extension name, if it corresponds to this level
    psMetadata *header;                    // The FITS header, if it corresponds to this level
    psArray *images;                     // The pixel data, if it corresponds to this level
    psArray *masks;                     // The mask data, if it corresponds to this level
    psArray *weights;                     // The weight data, if it corresponds to this level
} p_pmHDU;
```

3.3 Detector Coordinate Transformations

These container levels also include in their definition the information needed to transform the coordinates in one of the levels to the coordinate system relevant at the higher levels.

The data structures define the basic coordinate relationships between all of these data elements. A set of offsets for each level in the data hierarchy specifies the location of the particular set of pixels in the next level of the hierarchy. This is illustrated in Figure 1. These offsets may be used to define the complete camera layout in the approximating assumption that all pixels in the camera are laid out on a single linear pixel grid. This approximate is sufficient for many basic operations. For more detail, the precise astrometric relationship between each level of the hierarchy may also be made available in the metadata of the data structures.

In practice, a single readout from a detector may represent only a subset of the complete set of pixels addressed by the *cell*. The readout may also have binning applied in both of the two dimensions. There may also be overscan and pre-scan regions in the set of pixels. Finally, the readout direction is not always the same for all detector amplifiers. As shown in Figure 1,

these different concepts are represented in the data hierarchy. The coordinate of the origin of the data grid for one level of the hierarchy in the grid of the containing hierarchy is defined for each data level. For example, the origin of the coordinates for a single chip are located in the camera grid at `pmChip.cell0,row0`. The `pmReadout` data level has additional information to specify the details of the readout process. The elements `pmReadout.colBins,rowBins` specify the binning factor in the two dimensions, while the sign indicates the parity of the specific readout (readout direction). Note that the value of `pmReadout.col0,row0` must be assigned in such a way that it represents the coordinate of the origin pixel in the actual image: the overscan or pre-scan pixels must be accounted for. Putting all of these element together, we can see that the pixel coordinates in the camera grid may be determined from the pixel coordinates in the image grid from the following relationship:

```
pmFPA(cell,row) = pmChip.cell0,row0 + pmCell.cell0,row0 + pmReadout.cell0,row0 +
                  pmReadout.cellParity,rowParity * pmReadout.cellBins,rowBins *
                  psImage.data(cell,row)
```

3.4 Input/Output of a Focal Plane Hierarchy

We specify two functions to construct a focal plane hierarchy from a camera configuration and read from a FITS file. These two operations are decoupled so that the big investment of memory from the read only occurs when it is necessary.

```
pmFPA *pmFPAConstruct(const psMetadata *camera, psDB *db);
bool pmFPARead(pmFPA *fpa, psFits *fits);
```

`pmFPAConstruct` shall construct a focal plane hierarchy from a `camera` configuration. A `db` handle is also provided so that may be set in the `pmFPA`. The resultant `pmFPA` and its lower-down components shall be ready for to read a FITS file into it by setting the `extname` pointers at the appropriate levels to the appropriate FITS extension name.

`pmFPARead` shall read a `fits` file (the contents of which are described by the previous `camera` configuration) into an extant `fpa`. This involves reading the headers and pixels, as well as ingesting all the concepts.

```
bool pmFPASelectChip(pmFPA *fpa, int chipNum);
int pmFPAExcludeChip(pmFPA *fpa, int chipNum);
```

These functions are provided to set the `valid` booleans within an `fpa` so that only certain chips within the FITS file are read in.

`pmFPASelectChip` shall set `valid` to `true` for the specified chip number (`chipNum`), and all other chips shall have `valid` set to `false`. In the event that the specified chip number does not exist within the `fpa`, the function shall return `false`.

`pmFPAExcludeChip` shall set `valid` to `false` only for the specified chip number (`chipNum`). In the event that the specified chip number does not exist within the `fpa`, the function shall generate a warning, and perform no action. The function shall return the number of chips within the `fpa` that have `valid` set to `true`.

make these functions richer: select by extention, extname, cell, options to invalidate all / validate all, etc (TBD)

```
bool pmFPAMorph(pmFPA *toFPA, pmFPA *fromFPA, bool positionDependent, int chipNum, int cellNum);
```

`pmFPAMorph` shall morph the `fromFPA` focal plane hierarchy to the `toFPA` focal plane hierarchy. This allows us to write the pixels out using a different (though consistent) camera configuration. In the event that the `toFPA` has different levels than the `fromFPA`, only the chip and cell specified by `chipNum` and `cellNum` shall be written; if the levels are the same, these numbers are ignored. This function shall break apart pixel regions or splice them together where required in order to satisfy the demands of the `toFPA`. How the image and overscan regions are spliced together depends on the value of `positionDependent`: If `positionDependent` is `true`, then the overscan regions go to the low end of the spliced image if $i < N/2$, and to the high end of the spliced image if $i >= N/2$, where i (zero-offset) is the cell number, and N is the total number of cells; if `positionDependent` is `false`, then the overscan regions all go to the high end of the spliced image. Care should be taken to check the `CELL.READDIR`, so the orientation of the overscan regions is known. If the bias and trim sections are specified by headers in the `toFPA`, these shall be updated appropriately; otherwise, the function is permitted to fail with a suitable error message, in which case it shall return `false`.

```
bool pmFPAWrite(psFits *fits, pmFPA *fpa);
```

`pmFPAWrite` shall write the focal plane hierarchy, `fpa`, to the specified `fits` file, returning `true` upon success and `false` otherwise. The `fpa` should contain sufficient information with which to write the FITS images.

```
pmFPAWriteMask(psFits *fits, pmFPA *fpa);
```

`pmFPAWriteMask` is very similar to `pmFPAWrite`, but it shall write the `mask` elements of the `pmReadout`s comprising the `fpa`.

```
pmFPAWriteWeight(psFits *fits, pmFPA *fpa);
```

`pmFPAWriteWeight` is very similar to `pmFPAWrite`, but it shall write the `weight` elements of the `pmReadout`s comprising the `fpa`.

4 Astrometry

Astrometry is a basic functionality required for the IPP that will be used repeatedly, both for low-precision (roughly where is my favorite object?) and high-precision (what is the proper motion of this star?). As such, it must be flexible, yet robust.

4.1 Coordinate frames

There are five coordinate frames that we need to worry about for the purposes of astrometry:

- Cell: (x, y) in pixels — raw coordinates;
- Chip: (X, Y) in pixels — the location on the silicon;
- Focal Plane: (p, q) in microns — the location on the focal plane;
- Tangent Plane: (l, m) in arcsec from the telescope boresight; and
- Sky: (RA,Dec) — ICRS.

The following steps are required to convert from the cell coordinates to the sky:

- Cell $\longleftrightarrow$ Chip: two 2D polynomials, $(X, Y) = f(x, y)$;
- Chip $\longleftrightarrow$ FP: two 2D polynomials, $(p, q) = g(X, Y)$;
- FP $\longleftrightarrow$ TP: two 4D polynomials, $(l, m) = h(p, q, m, c)$, where m and c are the magnitude and color of the object, respectively; and
- TP $\longleftrightarrow$ Sky: transformation to the sky using pre-computed coefficients for each pointing.

Note that the transformation between the Focal Plane and the Tangent Plane is a four-dimensional polynomial, in order to account for any possible dependencies in the astrometry on the stellar magnitude and color; the former serves as a check for charge transfer inefficiencies, while the latter will correct chromatic refraction, both through the atmosphere and the corrector lenses.

We require structures to contain each of the above transformations as well as the pixel data.

4.2 Position Finding

We require functions to return the structure containing given coordinates. For example, we want the chip that corresponds to the focal plane coordinates $(p, q) = (-1.234, +5.678)$. These routines handle the one-to-many problem — i.e., for one given focal plane coordinate, there are many chips that this coordinate may be correspond to; these functions will select the correct one.

```
pmCell *pmCellInFPA (const psPlane *coord, const pmFPA *fpa);
pmChip *pmChipInFPA (const psPlane *coord, const pmFPA *fpa);
pmCell *pmCellInChip(const psPlane *coord, const pmChip *chip);
```

4.3 Conversion Functions

We require functions to convert between the various coordinate frames (Section 4.1). The hierarchy of the coordinate frames and the transformations between each are shown in Figure 2. The functions that employ the transformations are shown in Figure 3. In addition to transformations between each adjoining coordinate frame in the hierarchy, we also require higher-level functions to convert between the Cell and Sky coordinate frames; these will simply perform the intermediate steps.

We specify the following functions to convert between coordinates in one type of frame to another type of frame. The first group consist of unambiguous transformations: from the coordinates in a low-level frame to the coordinates in the containing higher-level frame, of which only one exists. In all of these functions, the output coordinate structure may be `NULL` or may be supplied by the calling function. In the former case, the structure must be allocated; in the latter case, the supplied structure must be used.

```
psPlane *pmCoordCellToChip (psPlane *out, const psPlane *in, const pmCell *cell);
% astrometry comes from cell (no need for parent)
```

which converts coordindates `in` on the specified `cell` to the coordinates on the parent chip.

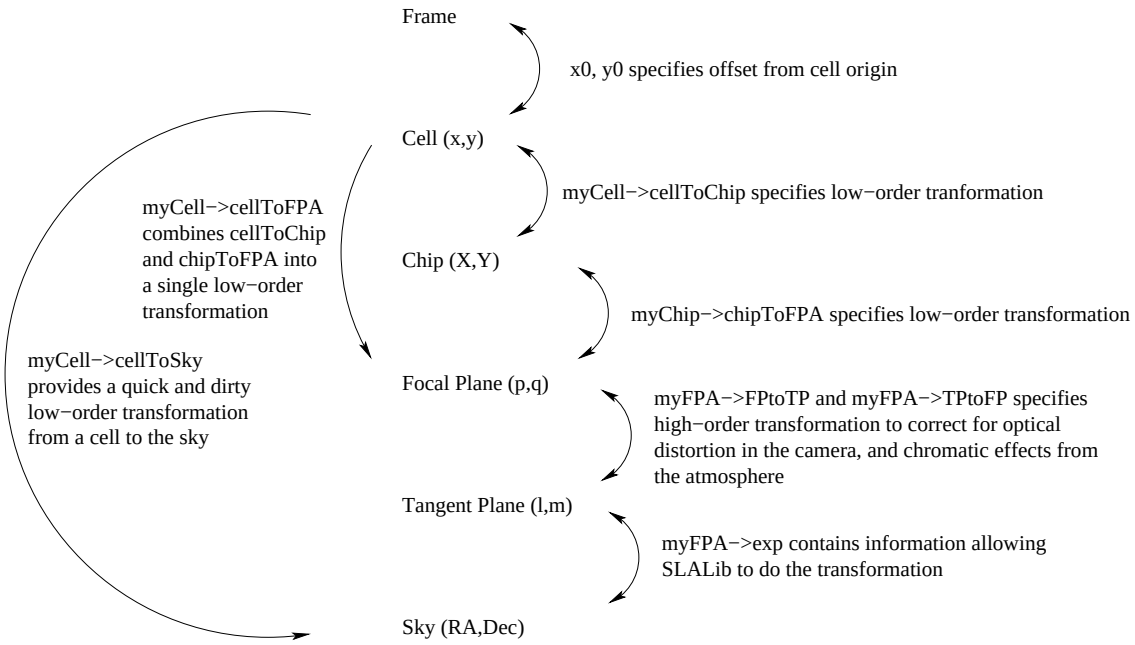

Figure 2: The coordinate systems in the Pan-STARRS IPP, and the relation between each by transformations contained in the appropriate structures.

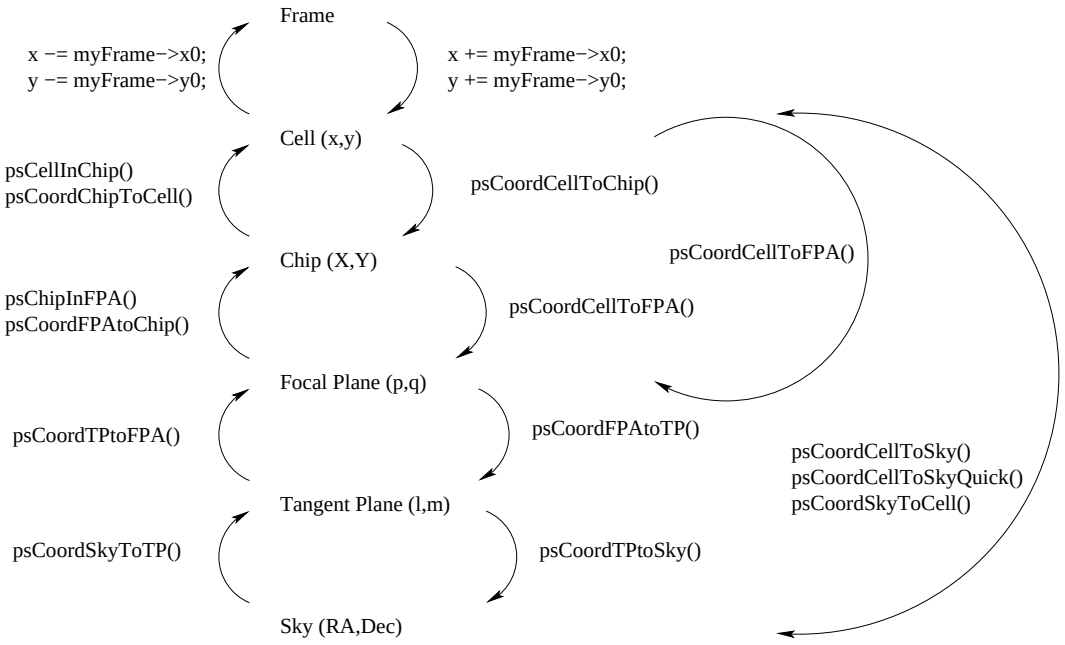

Figure 3: Conversion between coordinate systems by PSLib.

```
psPlane *pmCoordChipToFPA (psPlane *out, const psPlane *in, const pmChip *chip);
% astrometry comes from chip (no need for parent)
```

which converts the coordinates `in` on the specified `chip` to the coordinates on the parent FPA.

```
psPlane *pmCoordFPAToTP(psPlane *out, const psPlane *in, float color, float mag, const pmFPA *fpa);
% astrometry comes from FPA (no need for parent)
```

which converts coordinates `in` on the specified focal plane `fpa` to tangent plane coordinates, applying the appropriate distortion terms. The `color` and magnitude (`mag`) of the source is necessary in order to perform the distortion between the focal plane and the tangent plane.

```
psSphere *pmCoordTPToSky(psSphere *out, const psPlane *in, const psProjection *projection);
```

which converts the tangent plane coordinates `in` to (RA,Dec) on the sky, using the specified `projection`.

```
psPlane *pmCoordCellToFPA(psPlane *out, const psPlane *in, const pmCell *cell);
```

which performs the single-step conversion between Cell coordinates `in` and FPA coordinates.

```
psSphere *pmCoordCellToSky(psSphere *out, const psPlane *in, float color, float mag, const pmCell *cell);
```

which converts coordinates on the specified cell to (RA,Dec). This transformation must be performed using the intermediate stage transformations of Cell to Chip, Chip to FPA, FPA to Tangent Plane, Tangent Plane to Sky. The information needed for each of these transformations is available in the `.parent` elements of `pmCell` and `pmChip`, and the `pmFPA.projection`. The `color` and magnitude (`mag`) of the source is necessary in order to perform the distortion between the focal plane and the tangent plane.

```
psSphere *pmCoordCellToSkyQuick(psSphere *out, const psPlane *in, const pmCell *cell);
```

which uses the 'quick-and-dirty' transformation to convert coordinates on the specified cell to (RA,Dec). This transformation should use the locally linear transformation specified by the element `pmCell.toTP`. Although the accuracy of this transformation is lower than the complete transformation above, the calculation is substantially faster as it only involves linear transformations.

The following functions convert from high-level frames to the coordinates of contained lower-level frames.

```
psPlane *pmCoordSkyToTP(psPlane *out, const psSphere *in, const psProjection *projection);
```

which converts (RA,Dec) coordinates `in` to tangent plane coords using the specified `projection`.

```
psPlane *pmCoordTPToFPA(psPlane *out, const psPlane *in, float color, float mag, const pmFPA *fpa);
```

which converts the tangent plane coordinates `in` to focal plane coordinates. The `color` and magnitude (`mag`) of the source is necessary in order to perform the distortion between the focal plane and the tangent plane.

```
psPlane *pmCoordFPAToChip (psPlane *out, const psPlane *in, const pmChip *chip);
```

which converts the specified FPA coordinates `in` to the coordinates on the given Chip. The specified chip need not contain the input coordinate. To find the chip which contains a particular coordinate, the function `pmChipInFPA`, defined above, should be used.

```
psPlane *pmCoordChipToCell (psPlane *out, const psPlane *in, const pmCell *cell);
```

which converts the specified Chip coordinate `in` to the coordinate on the given Cell. The specified Cell need not contain the input coordinate. To find the cell which contains a particular coordinate, the function `pmCellInChip`, defined above, should be used.

```
psPlane *pmCoordSkyToCell(psPlane *out, const psSphere *in, float color, float mag, pmCell *cell);
```

which directly converts (RA,Dec) `in` to coordinates on the specified cell. The specified cell need not contain the input coordinates. The `color` and magnitude (`mag`) of the source is necessary in order to perform the distortion between the focal plane and the tangent plane.

```
psPlane *pmCoordSkyToCellQuick(psPlane *out, const psSphere *in, pmCell *cell);
```

which directly converts (RA,Dec) `in` to coordinates on the specified cell. The specified cell need not contain the input coordinates. This transformation should use the locally linear transformation specified by the element `pmCell.toTP`. Although the accuracy of this transformation is lower than the complete transformation above, the calculation is substantially faster as it only involves linear transformations.

4.4 FITS World Coordinate System

The FITS World Coordinate System (WCS) headers are commonly employed with astronomical images in order to relate pixels to celestial (or otherwise) coordinates. Since it is a FITS standard, we must be able to read and write from WCS into our internal format. For the time being, we will consider only celestial WCS (i.e., no spectral wavelength calibrations, etc). Because WCS does not support the multiple layers that we have built for Pan-STARRS, we will use a simple internal representation: a transformation, which handles any distortions (i.e., goes directly from the coordinate frame of the image to the tangent plane); and the projection.

```
bool pmAstromReadWCS(psPlaneTransform **transform, // Output transformation
                     psProjection **projection, // Output projection
                     psMetadata *header // Input FITS header
);
bool pmAstromWriteWCS(psMetadata *header, // Output FITS header
                      psPlaneTransform *transform, // Input transformation
                      psProjection *projection // Input projection
);
```

`pmAstromReadWCS` shall parse the specified FITS `header`, returning new instances of the `transform` and `projection` that represent the WCS. The function shall return `true` if it was able to successfully generate the outputs; otherwise it shall return `false`.

`pmAstromWriteWCS` shall add WCS keywords to the supplied FITS `header` that implement the given `transform` and `projection`. The function shall return `true` if it was able to successfully generate the output; otherwise it shall return `false`.

```
bool pmAstrometrySimplify(psPlaneTransform **transform, // Output transformation
                          psProjection **projection, // Output projection
                          pmCell *cell // Cell for which to generate transform and projection
);
```

`pmSimplifyAstrometry` shall take a `cell` and simplify the internal astrometric representation (`cell->toFPA` or equivalent, `cell->parent->parent->toTangentPlane` and `cell->parent->parent->grommit`) to a single `transform` and `projection`. This allows the subsequent use of `pmWriteAstrometry` in the case that we have only the multi-layered Pan-STARRS internal astrometric representation. The function shall return `true` if it was able to successfully generate the output; otherwise it shall return `false`.

4.5 Astrometry Analysis

Astrometry is performed on an astronomical image after a collection of sources in the image have been detected and thier instrumental positions have been measured. In this collection of tools, the coordinates should be measured in the frame of the `pmReadout` portion of the Image Hierarchy. This potentially allows us to measure an astrometric transformation resulting from the transformation to any of the other coordinate system. For example, it might be necessary to determine the coordinates of the `psReadout` pixels relative to the `psCell` (rather than accept the relationship defined by the metadata).

For the moment, we define two layers of astrometric analysis which will be performed on typical mosaic images in the Pan-STARRS IPP: per-chip astrometry and per-mosaic astrometry. In the first case, a collection of detections across a single chip are used to determine a basic, linear transformation to celestial coordinates. This astrometric analysis can be used to determine an initial, approximate astrometry for each chip in a mosaic camera, with accurcy limited by the effects of optical distortion (a few pixels error, typically). In the second case, a collection of detectors on a collection of chips from a mosaic camera are used to measure the position of the telescope boresight, the camera rotation, the impact of distortion from the telescope optics, and the chip-to-focal plane transformation resulting from chip placement errors or possible detector tilts or warps.

The process of performing astrometry involves the following steps:

- Make an initial guess at the celestial coordinates of the raw detections (eg, from metadata or image header information).
- Determine, and load, stars from an astrometric catalog which may potentially correspond to the raw detections.
- Project both raw and reference stars to a common coordinate frame (here we use the Focal Plane as an appropriate coordinate system).
- Identify matches between the raw and reference stars.
- Determine the transformations necessary to relate the coordinates of the two sets of stars.
- Convert the measured transformations into appropriate terms in the astrometric elements of the Image Hierarchy.

In this section, we specify several functions which together make possible the above analysis steps.

4.5.1 Astrometry Objects

We define the following structure to carry the necessary information about each detection.

```
typedef struct {
    psPlane pix;
    psPlane FP;
    psPlane TP;
    psSphere sky;
    double mag;
} pmAstromObj;
```

This structure specifies the coordinate of the detection in each of the four necessary coordinate frames: `pix` defines the position in the `psReadout` frame, `FP` defines the position in the Focal Plane frame, `TP` defines the position in the Tangent Plane frame, `sky` defines the position on the Celestial Sphere. In addition, a measurement of the brightness is given by the element `Mag`. Such a data structure should be used for both the raw and the reference stars. In astrometric processing, the raw detections will be projected using the best available information to each of these coordinate frames from the `pix` coordinates, while the reference detections will be projected to the other frames from the `sky` coordinates.

The raw detections and the reference stars are both projected to a common coordinate frame for analysis. In these modules, we use the Focal Plane for this reference frame. After projection to the common frame, it is necessary to determine the match between corresponding objects in the two lists. In order to match the raw detections to the reference stars, different methods are used depending on the circumstance.

4.5.2 Matching Stars : Close Match

If the two sets of coordinates are expected to agree very well (ie, the current best-guess astrometric solution is quite close to the 'true' astrometric solution), then it is possible to use the simplest matching process: cross-correlation within a fixed radius. The following function accepts two sets of `pmAstromObj` sources and determines the matched objects between the two lists. The input sources must have been projected to the Focal Plane coordinates (`pmAstromObj.FP`), and the supplied `options` entry must contain the desired match radius (keyword: `ASTROM.MATCH.RADIUS`). The output consists an array of `pmAstromMatch` values, defined below.

```
psArray *pmAstromRadiusMatch (psArray *starlist1, psArray *starlist2, psMetadata *options);
```

```
typedef struct {
    int idx1;
    int idx2;
} pmAstromMatch;
```

The `pmAstromMatch` structure defines the cross-correlation between two arrays. An single such data item specifies that item number `pmAstromMatch.idx1` in the first list corresponds to `pmAstromMatch.idx2` in the second list.

4.5.3 Matching Stars : Rough Match

If the two sets of coordinates are not known to agree well, a somewhat different approach is needed. Several algorithms have been defined in the past to correlate two lists with unknown offsets, and potentially unknown relative rotations and

scaling. One well-known method is the triangle-match algorithm which searches for similar triangles observed in the two lists. This algorithm has the advantage of not requiring the rotation or the scale to be well-known in advance. The disadvantage of the triangle match algorithm is that it is necessarily an $O(N^3)$ process since it is necessary to construct a substantial fraction of all possible triangles for both input lists. **we do not define a triangle match algorithm at this time (TBD)** .

If the two sets of coordinates are not known to agree well, but the relative scale and approximate relative rotation is known, then a much faster match can be found using pair-pair displacements. In such a case, the two lists can be considered as having the same coordinate system, with an unknown relative displacement. In this algorithm, all possible pair-wise differences between the source positions in the two lists are constructed and accumulated in a grid of possible offset values. The resulting grid is searched for a cluster representing the offset between the two input lists. This algorithm can only tolerate a small error in the relative scale or the relative rotation of the two coordinate lists. However, this process is naturally $O(N^2)$, and is thus advantageous over triangle matching in some circumstances. This process can be extended to allow a larger uncertainty in the relative rotation by allowing the procedure to scan over a range of rotations. We define the following function to apply this matching algorithm:

```
pmAstromStats pmAstromGridMatch (psArray *raw, psArray *ref, psMetadata *options);
```

The input sources must have been projected to the Focal Plane coordinates (`pmAstromObj.FP`), and the supplied `options` entry must contain the following user-defined parameters:

- `GRID.OFFSET` : maximum allowed displacement in search
- `GRID.SCALE` : grid bin size in focal-plane coordinate units
- `GRID.MIN.ANGLE` : minimum tested relative rotation
- `GRID.MAX.ANGLE` : maximum tested relative rotation
- `GRID.DEL.ANGLE` : relative rotation step size

Note that the angles are defined as clockwise rotations of `raw` relative to `ref`.

This function returns values in the `pmAstromStats` structure, defined as:

```
typedef struct {
    psPlane center;
    psPlane offset;
    double  angle;
    double  minMetric;
    double  minVar;
    int     nMatch;
} pmAstromStats;
```

The elements `angle` and `offset` define the best rotation and offset; the element `nMatch` indicates the number of matched sources which fell within the match bin; the element `minVar` specifies the variance of the sources within the match bin; the element `minMetric` specifies the value of the selection metric for the matched bin. Note that the metric of choice may not necessarily be either the simple number of sources or the varience. We find that a combination based on both which enhances the importance of having a well-populated bin with a minimal variance gives good results: metric $=$ var $\times N^{-4}$. The element `center` defines the center of rotation used for rotating the `raw` stars relative to the `ref` stars.

The result of a pmAstromGridMatch may be used to modify a psPlaneTransform structure map. The result of pmAstromGridMatch can be translated into adjustments of the psPlaneTransform (ie the rotation and offset). This adjustment is made using the function:

```
bool pmAstromGridApply (psPlaneTransform *map, pmAstromStats stats);
```

This function modifies the supplied map entry assuming the adjustments are relative to the provided transformation.

We define two additional functions which are used in pmAstromGridMatch, but which may be useful on their own:

```
pmAstromStats pmAstromGridAngle (psArray *raw, psArray *ref, psMetadata *options);
```

This function is identical to pmAstromGridMatch, but is valid for only a single relative rotation. The input config information need not contain any of the GRID.*.ANGLE entries (they will be ignored).

```
psArray *pmAstromRotateObj (psArray *old, psPlane center, double angle);
```

This function accepts an array of pmAstromObj objects and rotates them by the given angle about the given center coordinate center in the Focal Plane Array coordinates.

4.5.4 Astrometry Fitting Routines

The result of a pmAstromRadiusMatch operation is a list of matched entries between the two input lists. This list may be used to determine a linear fit between the two sets of matched sources. The following function performs this operation:

```
bool pmAstromFitFPA (pmFPA *fpa, psArray *st1, psArray *st2, psArray *match, psMetadata *config);
```

This function accepts the raw and reference source lists and the list of matched entries. It uses the matched list to determine a polynomial transformation between the two coordinate systems. The fitting uses clipping to exclude outliers, likely representing poor matches. The config element must contain the information ASTROM.NSIGMA (specifying the number of sigma used in the clipping) and ASTROM.NCLIP (specifying the number of clipping iterations must be performed). The config element must also specify the order of the polynomial fit (keyword: ASTROM.ORDER). The result of this fit is a set of modifications of the components of the pmFPA.toTangentPlane transformation, and the modifications of the reference coordinate of the projection (pmFPA.projection.R,D) and the projection scale (pmFPA.projection.Xs,Ys). The modifications to pmFPA.toTangentPlane incorporate the rotation component of the linear terms and the higher-order terms of the polynomial fits.

An alternative to fitting the rotation of the FPA relative to the Tangent Plane is to treat the fitted transformation as a measurement of the chip within the FPA. The following function performs this operation in the same way as pmAstromFitFPA:

```
bool pmAstromFitChip (pmFPA *fpa, psArray *st1, psArray *st2, psArray *match, psMetadata *config);
```

This function accepts the raw and reference source lists for a single chip and the list of matched entries. It uses the matched list to determine a polynomial transformation between the two coordinate systems. The fitting uses clipping to exclude outliers, likely representing poor matches. The config element must contain the information ASTROM.NSIGMA (specifying the number of sigma used in the clipping) and ASTROM.NCLIP (specifying the number of clipping iterations

must be performed). The `config` element must also specify the order of the polynomial fit (keyword: `ASTROM.ORDER`). The result of this fit is a set of modifications of the components of the `pmChip.toFPA` transformation.

A mosaic represents a particular set of challenges when determining an astrometric solution. There is substantial degeneracy between the astrometric terms which describe the transformationfrom the chip to the focal plane, and the tranformation from the focal plane to the tangent plane, in the presence of distortion. The degeneracy can be broken by examining the distortion component in its effect on the gradient of the sources position residuals rather than in the source positions themselves. The following function determines the position residual, in the tangent plane, as a function of position in the focal plane, for a collection of raw measurements and matched reference stars. The configuration data must include the bin size over which the gradient is measured (keyword: `ASTROM.GRAD.BOX`). The function returns an array of `pmAstromGradient` structures, defined below.

```
psArray pmAstromMeasureGradients (psArray *starlist1, psArray *starlist2, psArray *match, psMetadata *confi
```

The following data structure carries the information about the residual gradient of source positions in the tangent plane (`pmAstromObj.TP`) as a function of position in the focal plane (`pmAstromObj.FP`).

```
typedef struct {
    psPlane FP;
    psPlane dTPdL;
    psPlane dTPdM;
} pmAstromGradient;
```

The gradient set measured above can be fitted with a pair of 2D polynomials. The resulting fits can then be related back to the implied polynomials which represent the distortion. The following function performs the fit and applies the result to the distortion transformation of the supplied `pmFPA` structure. The configuration variable supplies the polynomial order (keyword: `ASTROM.DISTORT.ORDER`).

```
psArray pmAstromFitDistortion (pmFPA *fpa, psArray *gradients, psMetadata *config);
```

5 Photometry

This section is to be deferred, and for now consists only of place holders, with no functional items. (TBD)

Photometric observations are performed in an instrumental photometric system, and must be related to other photometric systems. We require a data structure which defines a photometric system, as well as a structure to define the transformation between photometric systems.

The photometric system is defined by the psPhotSystem structure. A photometric system is identified by a human-readable `name` (ie, SDSS.g, Landolt92.B, GPC1.OTA32.r). Each photometric system is given a unique identifier `ID`. Observations taken with a specific camera, detector, and filter represent their own photometric system, and it may be necessary to perform transformations between these systems. Photometric systems associated with observations from a specific camera/detector/filter combination can be associated with those components.

```
typedef struct {
    const int ID;                       ///< ID number for this photometric system
    const char *name;                   ///< Name of photometric system
    const char *camera;                 ///< Camera for photometric system
    const char *filter;                 ///< Filter used for photometric system
    const char *detector;               ///< Detector used for photometric system
} psPhotSystem;
```

The following structure defines the transformation between two photometric systems.

```
typedef struct {
    psPhotSystem src;                   ///< Source photometric system
    psPhotSystem dst;                   ///< Destination photometric system
    psPhotSystem pP, pM;                ///< Primary color reference
    psPhotSystem sP, sM;                ///< Secondary color reference
    float pA, sA;                       ///< Color offset for references
    psPolynomial3D transform;           ///< Transformation from source to destination
} psPhotTransform;
```

The transformation between two photometric systems may depend on the airmass of the observation and on the colors of the object of interest. For a specific observation, such a transformations can be defined as a polynomial function of the color of the star and the airmass of the observations. If sufficient data exists, the transformation between the photometric systems may include more than one color, constraining the curvature of the stellar spectral energy distributions. This latter term may be significant for stars which are highly reddened, for example. Derived photometric quantities may have been corrected for airmass variations, in which case only color terms may be measurable. The structure defines the transformation between a source photometric system (`src`) and a target photometric system (`dst`). The photometric system of a primary color is defined by `pP, pM` such that the color is constructed as $pP - pM$. A secondary color is defined by `sP, sM`. For both, a reference color is specified (`pA, sA`): the polynomial transformation terms refer to colors in the form $pP - pM - pA$. The transformation is specified as a 3D polynomial. For a star of magnitude $M_{\rm src}$ in the source photometric system, with additional magnitude information in the other systems $M_{\rm pP}$, $M_{\rm pM}$, $M_{\rm sP}$, $M_{\rm sM}$, observed at an airmass of z, the magnitude of the star in the target system $M_{\rm dst}$ is given by: $M_{\rm dst} = M_{\rm src} + transform(z, M_{\rm pP} - M_{\rm pM} - pA, M_{\rm sP} - M_{\rm sM} - sA)$.

6 Image Detrending

Image Detrending is the image analysis process wherein the instrumental signatures are removed from the individual images. This section discusses the modules used for image detrending. The basic image detrending steps are:

- Subtract bias;
- Correct for non-linearity;
- Flat-field;
- Mask bad pixels;
- Subtract the background;
- Mask cosmic rays;
- Mask optical defects;

6.1 Bias subtraction

The bias subtraction module provides a facility to correct detector images for the electronic pedestal introduced by the readout electronics.

Given an input image and various other parameters, `pmSubtractBias` shall subtract the bias from the image:

```
pmReadout *pmSubtractBias(pmReadout *in, pmOverscanOptions *overscanOpts,
                          psRegion imageRegion, psList *overscanRegions,
                          const pmReadout *bias, const pmReadout *dark);
```

Three types of bias correction may optionally be performed on the input image, `in`. The first is the subtraction of an overscan. Multiple overscan regions may be specified and fit as a function of row (or column). The second is the subtraction of a full-frame bias image. The third is the subtraction of a suitably scaled full-frame dark image.

The input image, `in`, shall have the bias subtracted in-place. The input image may be of type U16, S32, or F32. The region of the input image that shall have the overscan or full-frame subtractions applied is specified by `imageRegion`.

Overscan subtraction is performed if `overscanOpts` is non-`NULL` (see §6.1.1). `overscanRegions` shall be a list of `psRegion`s that specify the regions that comprise the overscans.

A `bias` frame shall be subtracted pixel-by-pixel from the input image if `bias` is non-NULL. If `dark` is non-`NULL`, then the dark image, scaled by the ratio of dark times (from `CELL.DARKTIME`) shall be subtracted pixel-by-pixel from the input image. The full-frame subtractions (both bias and dark) should only be performed on the image region specified by `CELL.TRIMSEC`. Note that the input image, `in`, and the `bias` and `dark` frames need not be the same size, but the function shall use the offsets in the image (`in->x0` and `in->y0`) to determine the appropriate offsets to obtain the correct pixel on the `bias`. In the event that the `bias` image is too small (i.e., pixels on the input image refer to pixels outside the range of the `bias` image), the function shall generate an error. Any pixels masked in the `bias` or `dark` shall also be masked in the output. The bias and dark images may be copied to the same type as the input image if required.

6.1.1 Overscan subtraction

The options for performing the overscan subtraction are bundled in a `pmOverscanOptions`:

```
typedef struct {
    // Inputs
    bool single;               // Reduce all overscan regions to a single value?
    bool scanRows;             // Scan direction was rows? (otherwise columns)
    pmFit fitType;             // Type of fit to overscan
    unsigned int order;        // Order of polynomial, or number of spline pieces
    psStats *stat;             // Statistic to use when reducing the minor direction
    // Outputs
    psPolynomial1D *poly;      // Result of polynomial fit
    psSpline1D *spline;        // Result of spline fit
} pmOverscanOptions;
```

The mode in which the overscan is subtracted is specified by the `single` boolean. If `single` is `true`, then the entire overscan region is reduced to a single value using the `stat`. If `single` is `false`, the overscan shall be reduced along the dimension specified by `scanRows` (rows if `scanRows` is true; otherwise columns).

If the overscan is not defined for each row/column, `pmSubtractBias` shall generate an error if `fitType` is `PM_FIT_NONE`; otherwise, the function shall shall generate a warning and the undefined values shall be interpolated using the provided functional form.

The statistic to use in combining multiple pixels in the prescan/overscan regions is specified by `stat`. `stat` is of type `psStats` instead of simply `psStatsOptions` so that clipping levels may be specified, if desired. In the event that multiple options are specified by `stats`, a warning shall be

generated, and the option with the highest priority shall be used, according to the following priority order: PS_STAT_SAMPLE_MEAN, PS_STAT_SAMPLE_MEDIAN, PS_STAT_CLIPPED_MEAN, PS_STAT_ROBUST_MEAN, PS_STAT_ROBUST_MEDIAN, PS_STAT_ROBUST_MODE.

fitType is an enumerated type which specifies the type of fit to employed on the overscan vector:

```
typedef enum {
    PM_FIT_NONE,                        ///< No fit
    PM_FIT_POLY_ORD,                    ///< Fit ordinary polynomial
    PM_FIT_POLY_CHEBY,                  ///< Fit Chebyshev polynomial
    PM_FIT_SPLINE                       ///< Fit cubic splines
} pmFit;
```

If fitType is PM_FIT_NONE, then the overscan vector is subtracted from the image without fitting. Otherwise, the overscan vector is fit using the specified functional form, the fit is subtracted from the image, and the poly or spline is allocated and updated with the results of the fit.

The allocator for a pmOverscanOptions shall be:

```
pmOverscanOptions *pmOverscanOptionsAlloc(bool single, bool scanRows,
                                          pmFit fitType, unsigned int order,
                                          psStats *stat);
```

6.2 Non-linearity

We here specify two functions to perform the non-linearity correction, since either (or both) might be used to specify the correction.

These operations act only on the region of the readout specified by CELL.TRIMSEC.

The first, pmNonLinearityPolynomial shall correct the input image for non-linearity by replacing the flux in each pixel of the input image, in, with the result of the specified polynomial, coeff, acting on the flux. The API shall be the following:

```
pmReadout *pmNonLinearityPolynomial(pmReadout *in, const psPolynomial1D *coeff);
```

The polynomial coefficients, coeff, will be supplied by the caller, likely from the image metadata.

The second function, pmNonLinearityLookup shall correct the input image for non-linearity by using a lookup table. The API shall be the following:

```
pmReadout *pmNonLinearityLookup(pmReadout *in, const char *filename);
```

For each pixel in the input image, the function shall replace the flux with the corresponding value from the supplied lookup table, specified by the filename. The lookup table file shall consist of two columns of data, the first being the original flux value and the second being the replaced flux value. The file shall be in a format suitable for reading by psLookupTableRead.

Both pmNonLinearityPolynomial and pmNonLinearityLookup shall modify the input image in-place. The input image may be of type U16, S32, or F32.

6.3 Flat-fielding

Given an input image and a flat-field image, `pmFlatField` shall divide the input image by the flat-field image and return it in place, updating the mask contained within the input image as appropriate. The API shall be the following:

```
bool pmFlatField(pmReadout *in, const pmReadout *flat);
```

Note that the input image, `in`, and the flat-field image, `flat`, need not be the same size, since the input image may already have been trimmed (following overscan subtraction), but the function shall use the offsets of the readout (`in->col0, in->row0`) and the image subarray (`in->image->x0, in->image->y0`) to determine the appropriate offsets to obtain the correct detector pixels in the flat-field image. Note that the image offset is relative to its parent, so this offset must be followed to the top level image which is not a child of another image and the offsets summed. The detector pixel coordinates of pixel `x,y` in a top-level image are thus `x + in->image->x0 + in->col0, y + in->image->y0 + in->row0`. In the event that the `flat` image is too small (i.e., pixels on the input image refer to pixels outside the range of the `flat` image), the function shall generate an error.

Pixels which are negative or zero in the `flat` shall be masked in the input image with the value `PM_MASK_FLAT` (see §6.4.1). Negative pixels in the `flat` may be set to zero so that they are treated identically to zeroes. Any pixels masked in the `flat` shall be masked with corresponding values in the `output`.

The function shall not normalize the `flat`; this responsibility is left to the caller. This function is basically equivalent to a divide (with `psImageOp`), but with care for the region that is divided, checking for zero and negative pixels, and copying of the mask from the `flat` to the output.

The images in the input and flat-field readouts must both be of type F32.

This operation acts only on the region of the readout specified by `CELL.TRIMSEC`.

6.4 Masking

6.4.1 Mask values

We define several mask values for use in the detrend processing:

```
/** Mask values */
typedef enum {
    PM_MASK_TRAP       = 0x0001,        ///< The pixel is a charge trap
    PM_MASK_BADCOL     = 0x0002,        ///< The pixel is a bad column
    PM_MASK_SAT        = 0x0004,        ///< The pixel is saturated
    PM_MASK_FLAT       = 0x0008         ///< The pixel is non-positive in the flat-field
} pmMaskValue;
```

Of these, masks for the charge traps need to be grown by the extent of the OT convolution kernel. For other pixel types, orthogonal transfer of the flux in this pixel will not (necessarily) affect the flux in neighbouring pixels.

6.4.2 Bad pixels

Given an input image, `in`, a bad pixel `mask`, a corresponding value in the bad pixel mask to mask in the input image, `maskVal`, a saturation level, and a growing radius, `pmMaskBadPixels` shall mask in the input image those pixels in the bad pixel mask that match the value to mask. The API shall be the following:

```
pmReadout *pmMaskBadPixels(pmReadout *in, const pmReadout *mask, unsigned int maskVal,
                           float sat, unsigned int growVal, int grow);
```

Note that the input image, `in`, is modified in-place. All pixels in the `mask` which satisfy the `maskVal` shall have their corresponding pixels masked in the input image, `in`. All pixels which satisfy the `growVal` shall have their corresponding pixels, along with all pixels within the `grow` radius masked. Pixels which have flux greater than `sat` shall also be masked, and grown by a single pixel (in addition to the `grow` done on the `growVal`).

In the future, may change `grow` to a convolution kernel (TBD) .

Note that the input image, `in`, and the `mask` need not be the same size, since the input image may already have been trimmed (following overscan subtraction), but the function shall use the offsets in the image (`in->x0` and `in->y0`) to determine the appropriate offsets to obtain the correct pixel on the mask. In the event that the `mask` image is too small (i.e., pixels on the input image correspond to pixels outside the range of the `mask` image), the function shall generate an error.

The input image may be of type U16, S32 or F32. The mask image must be of type U8.

This operation acts only on the region of the readout specified by `CELL.TRIMSEC`.

6.5 Subtract sky

This may be deferred. (TBD)

Given an input image, a polynomial or spline specifying the order of a desired fit, a binning factor and statistics to use for the binning, along with a clipping level, `pmSubtractSky` shall fit and subtract a model for the background of the image. The API shall be the following:

```
pmReadout *pmSubtractSky(pmReadout *in, psPolynomial2D *poly, psImage *mask, psU8 maskVal,
                         int binFactor, psStats *stats, float clipSD);
```

Note that the input image, `in`, shall be subtracted in-place. The function shall return the subtracted image, and also update the polynomial, Chebyshev or spline specified by `fitSpec`, to hold the coefficients used in the subtraction.

The polynomial, `poly`, specifies the order of the polynomial, and on return shall contain the coefficients of the fit. If `poly` is `NULL`, then no fit shall be performed, and the function shall generate a warning and return.

When fitting the polynomial, the function shall first bin the input image by `binFactor` in order to reduce the required processing time. In the binning, pixels in the `mask` (if non-`NULL`) which satisfy the `maskVal` shall be excluded. The statistic to use in this binning is specified by `stat`. `stat` is of type `psStats` instead of simply `psStatsOptions` so that clipping levels may be specified, if desired. In the event that multiple options are specified by `stats`, a warning shall be generated, and the option with the highest priority shall be used, according to the following priority order: `PS_STAT_SAMPLE_MEAN`, `PS_STAT_SAMPLE_MEDIAN`, `PS_STAT_CLIPPED_MEAN`, `PS_STAT_ROBUST_MEAN`, `PS_STAT_ROBUST_MEDIAN`, `PS_STAT_ROBUST_MODE`. If the `binFactor` is non-positive, or `stats` is `NULL` or fails to specify an option, a warning shall be generated, and the fit shall be performed on the entire image.

Binned pixels deviating more than `clipSD` standard deviations from the mean of the binned pixels shall be clipped in a single clipping iteration before polynomial fitting. These pixels may be interpolated over, or may be simply ignored in the fitting, according to the choice of algorithm. If the `clipSD` is non-positive, then the function shall generate a warning and not perform any clipping.

The `mask` shall be of type U8, and the input image, `in`, must be of type F32.

This operation acts only on the region of the readout specified by `CELL.TRIMSEC`.

6.6 Paper Trail

The elements of the focal plane hierarchy each contain an `analysis` member, intended to log the results of the detrend tasks. The detrend tasks shall add to the `analysis` members as follows:

- `pmMaskBadPixels`:
  - `MASK.DONE` (STR): The time at which masking was completed.
  - `MASK.SAT` (S32): The number of saturated pixels masked in the image
  - `MASK.SAT.GROW` (S32): The number of additional pixels masked by growing the saturated pixels.
  - `MASK.BAD` (S32): The number of pixels masked in the image
  - `MASK.BAD.GROW` (S32): The number of additional pixels masked by growing the specified bad pixels.
- `pmNonLinearityPolynomial` and `pmNonLinearityLookup`:
  - `NONLIN.DONE` (STR): The time at which the non-linearity correction was completed.
  - `NONLIN.POLY` (STR): The polynomial coefficients used (if applicable).
  - `NONLIN.LOOKUP` (STR): The filename for the lookup table (if applicable).
- `pmSubtractBias`:
  - `BIAS.DONE` (STR): The time at which the bias-subtraction was completed.
  - `BIAS.OVERSCAN.AXIS` (STR): Overscan axis used.
  - `BIAS.OVERSCAN.FIT.TYPE` (STR): Fit type applied to overscan.
  - `BIAS.OVERSCAN.FIT.COEFF` (STR): Coefficients of overscan fit.
  - `BIAS.OVERSCAN.REGION` (STR): Overscan regions (from `x0,y0,numCols,numRows`).
  - `BIAS.OVERSCAN.BIN` (S32): Number of pixels per bin used in overscan.
  - `BIAS.OVERSCAN.MEAN` (F32): The mean of the binned overscan pixels after subtracting the fit.
  - `BIAS.OVERSCAN.SD` (F32): The standard deviation of the binned overscan pixels after subtracting the fit.
- `pmFlatField`:
  - `FLAT.DONE` (STR): The time at which the flat-fielding was completed.
  - `FLAT.BAD` (S32): Number of non-positive flat-field pixels.

To be added by higher-levels:

- `BIAS.NAME` (STR): Name of bias image
- `DARK.NAME` (STR): Name of dark image
- `FLAT.NAME` (STR): Name of flat image
- `MASK.NAME` (STR): Name of mask image

6.7 Detrend Lookups

When it comes time to perform a detrend operation on an image, it is necessary to determine *which* detrend image should be used. The Pan-STARRS Image Processing Pipeline uses the concept of a detrend image database table, or set of tables (part of the Metadata Database), to store the known master detrend images. These tables can be accessed though the basic query functions specified for the master detrend database. To simplify the interaction for the case of the detrend images, the following function allows the user to explicitly search the detrend database table or tables for detrend images which satisfy a set of characteristics.

```
psArray *pmDetrendLookup (psMetadata *constraints, psMetadata *tableDefs);
```

This function accepts a metadata structure which restricts the selected detrend images. This metadata structure may contain any of the following entries:

```
TYPE        type of detrend data (eg, flat, bias)
CAMERA      name of desired camera (eg, GPC, MEGACAM)
CHIP        chip identifier (eg., ccd00)
FILTERNAME  name of specific filter hardware (eg, r.GPC01)
FILTERTYPE  conceptual name of filter (eg., r)
TIME_MIN    lower bound on valid time range
TIME_MAX    upper bound on valid time range
LABEL       match the entry label
RECIPE      recipe used to build detrend image
EXPTIME     exposure time
AIRMASS     airmass
```

Any detrend images which match the provided constraints are returned as an array of `psMetadata` elements corresponding to the columns of the detrend database table. The additional input parameter specifies additional information to define the detrend database tables. This may include the access information (IP, Username, Password), as well as names for the table and the columns which correspond to the constraint names.

7 Detrend Creation

In the detrend creation process, a collection of raw images are combined to produce a clean, high-quality master image for correcting the effect of interest. The input images may potentially be processed and scaled in some way. The resulting output images may be to be re-scaled to have a consistent signal for all chips in the mosaic. The simplest example is the construction of a bias image, in the case where there is signficant 2-D bias structure. In this case, the input raw bias images are probably combined without any additional processing. In another example, flat-field image must be bias-corrected and scaled to a consistent normalization before being combined, and the flat-field images from the different chips must be normalized so that each chip will be flattened consistently across the mosaic. A complex example is the fringe pattern, in which the input images must be bias-corrected and flattened, and the resulting images must be scaled by the amplitude of the fringe pattern on each image, rather than by the average flux level. In this section, we define the tools necessary to perform the detrend creation process.

7.1 Image Stacking

A basic operation in generating the master detrend images is using a stack of many input images of a particular type and combining them, with perhaps some additional scaling, in order to build up signal-to-noise and to reject deviant pixel. For

this, we require a general purpose image combination module. We forsee this module as only acting upon data from the same detector, and so each input image will have the same noise characteristics.

```
typedef struct {
    psStats *stats;                     // Statistics to use in combining pixels
    unsigned int maskVal,               // Mask pixels where mask & maskVal == 1
    float fracHigh;                     // Fraction of high pixels to throw
    float fracLow;                      // Fraction of low pixels to throw
    int nKeep;                          // Number of pixels to be sure to keep
} pmCombineParams;

psImage *
pmReadoutCombine(psImage *output,       // Output image, or NULL
                 const psList *inputs,  // List of input readouts
                 pmCombineParams *params, // Combination parameters
                 const psVector *zero,  // Offsets to apply for each image
                 const psVector *scale, // Scales to apply for each image
                 bool applyZeroScale,   // Are zero and scale for application, or only noise properties?
                 float gain,            // Gain in e/ADU
                 float readnoise        // Read noise in e
                 );
```

`pmReadoutCombine` combines input images pixel by pixel — for each pixel of the output image, a stack of contributing input pixels is formed and combined. Several of its input parameters are lists or vectors, and if these are not all of the same length (or `NULL`), the module shall generate an error and return `NULL`.

If the provided `output` is `NULL`, then the module shall allocate a new image of sufficient size for the input images. If the `output` image is non-`NULL` and is not of sufficient size for the combined image, the module shall generate an error and return `NULL`.

If the `inputs` is `NULL`, the module shall generate an error and return `NULL`. Otherwise, the `inputs` shall be a list of `pmReadout`s. The images contained within the `pmReadout`s need not all be of the same size, but the module shall take into account the offsets (`col0,row0`) from the corner of the detector when comparing pixels, so that it is the same *physical* pixels that are combined.

The parameters used in the combination, including how the pixels are to be combined, and how the rejection is performed is contained within the `params`, which may not be `NULL` (otherwise the module shall generate an error and return `NULL`). We choose to use this structure instead of supplying the values separately in order to keep down the number of parameters to `pmReadoutCombine`; the `pmCombineParams` may be recycled for subsequent calls to `pmReadoutCombine` since the values are not dependent upon the choice of inputs, but merely specify how the combination is to be performed.

The particular statistic specified by `stats` shall be used to combine each stack of pixels from the input images. Only one of the statistics choices may be specified, otherwise the module shall generate an error and return `NULL`.

If the `maskVal` is non-zero, then pixels in the `mask` of each `pmReadout` in the `inputs` which satisfy the `maskVal` shall not have the corresponding pixels placed in the stack for combination.

After masking, but before performing the combination, the highest `fracHigh` fraction and lowest `fracLow` fraction of pixels in the stack are immediately rejected, unless this would leave less than `nKeep` pixels in the stack, in which case no immediate rejection is performed.

If the `zero` vector is non-`NULL` and `applyZeroScale` is `true`, then the appropriate values shall be added to the `inputs` before rejection is performed. If `zero` is non-`NULL` and `applyZeroScale` is false, then the values shall only be used in calculating the Poisson variances.

If the `scale` vector is non-`NULL` and `applyZeroScale` is `true`, then the appropriate values shall multiply the `inputs` before rejection is performed. If `scale` is non-`NULL` and `applyZeroScale` is false, then the values shall only be used in calculating the Poisson variances.

The purpose of `applyZeroScale` is to allow combination of fringe frames, where the frames have been deliberately sky-subtracted and rescaled (to get the fringes amplitudes running from -1 to 1), which actions should not be undone when combining, but yet it is desirable to provide the `zero` and `scale` values so that the correct noise properties are used in the combination.

If the `gain` and `readnoise` are positive and non-negative (respectively), then these shall be used to provide weights for the combination using Poisson statistics (σ_i below).

In summary, pixels corresponding to the same physical pixel are combined, having values $x_i \pm \sigma_i$. In the case that `applyZeroScale` is `true`, then:

$$x_i = s_i f_i + z_i \quad (1)$$

$$\sigma_i = [g x_i + r^2]^{1/2}/g \quad (2)$$

Where f_i is the value of the pixel in image i, s_i is the scale applied to image i, z_i is the zero offset applied to image i, g is the gain, and r is the read noise. If scales are not provided, they are set to unity; if zero offsets are not provided, they are set to zero.

If `applyZeroScale` is `false`, then the values are:

$$x_i = f_i \quad (3)$$

$$\sigma_i = [g(s_i f_i + z_i) + r^2]^{1/2}/g \quad (4)$$

where the same symbols are used as above.

The `inputs, zero` and `scale` may be of U16, S32 and F32 types, and must all be of the same type. The `output` shall be of the same type.

7.2 Fringe Amplitude

Some images contain a signal caused by thin-film interference in the device due to strong emission lines. The resulting instrumental effect consists of a pattern (the “fringe pattern”) of bright and dark bands corresponding to the constructive and destructive interference of the emission lines. In the case that a single emission line causes the line structure, the resulting pattern can be described by two independent parameters: First, the amplitude of the emission line determines the overall amplitude of the pattern. Second, the three-dimensional surface structure of the device determines the shape of the pattern. In a typical situation, the device is illuminated by multiple emission lines, as well as a continuum spectral source, which contributes to the overall light detected by the device without following the fringe pattern. The relative intensities of the continuum background and the fringe pattern depend on the device structure (thickness) and on the ratio of the continuum and line emission fluxes.

A simple approach to the fringe pattern is to subtract a master fringe frame scaled by the amplitude of the fringe pattern. The amplitude of the fringe pattern is used both in the process of constructing the master image and in scaling the master image when it is applied to science image. We thus need a method of measuring the fringe amplitude which is robust in the presence of stars and which is fast. We implement a method used at CFHT in which the fringe pattern is mapped by a series of points pairs which correspond to peaks and valleys of the fringe pattern. We define the following function to measure the global fringe amplitude of an image given a collection of fringe point pairs.

```
stats *pmFringeStats (psArray *fringePoints, psImage *image, psMetadata *config);
```

This function measures the robust median at each of the minimum and maximum coordinates and determines the difference and mean of the two values. The size of the box used to make the measurement at each point is specified by the configuration variable `FRINGE_SQUARE_RADIUS`. From the collection of differences, the robust median is calculated, and returned as part of the fringe statistics. For each fringe point, the values of `delta` and `midValue` are also assigned and available to the user on return.

The `fringePoints` are defined by the following structure:

```
typedef struct {
    double xMin;
    double yMin;
    double xMax;
    double yMax;
    double delta;
    double midValue;
} pmFringePoint;
```

7.3 Flat-field Re-Normalization

Consider a collection of N_i flat-field images obtained with a mosaic camera consisting of N_j chips. Each image is exposed to an illumination source which should be a uniform surface brightness[1] Two factors determine the actual measured flux level (in Digital Numbers) on each of the chips in each image: the gain of each chip (gain_j) and the flux level from the illumination source (source_i). When the images are combined, the input images must be scaled so that the flux levels can be consistently compared. After combining the collection of images, it is necessary to determine an appropriate re-normalization for the resulting flat-field images. In effect, the individual chips must be adjusted so that the master flat-field image has a flux level which varies from chip to chip in proportion to the actual chip gain. In this case, if a uniform illumination source illuminates the mosaic, the resulting flux levels will be corrected by the flat-field to a single, consistent flux level.

In order to determine the correct relative scaling between the devices, it is thus necessary to know the individual chip gains, or at least the gain ratios. A typical technique scaled all chips relative to a reference chip, or by a statistic measured for the complete collection. These techniques fail if the input collection of images does not always consist of the same set of chips; for the GPC on Pan-STARRS, we must expect that individual cells or even chips may be disabled on a frequent basis, so our algorithms must not be limited by the assumption that all chips are available in all images. We therefore define the following algorithm to measure the relative chip gains for a collection of input flat-field images, each with a measured flux $\text{flux}_{i,j}$. We want to solve for the chip gains and the source illumination fluxes which would make the best prediction of the measured input image fluxes:

$$\text{flux}_{i,j}^{\text{pred}} = \text{gain}_j \times \text{source}_i$$

This relationship is easiest to determine if we take the logarithm of both sides of the equation:

$$M_{i,j}^{\text{pred}} = G_j + S_i$$

[1] This is likely a false assumption: the illumination source likely has spatial variations. However, for the purposes of this discussion, it only matters that such spatial variations scale consistently as a function of illumination intensity. The spatial errors are corrected by the photometric flat-field correction technique (eg., Magnier & Cuillandre 2004).

where $M_{i,j}^{\rm pred} = \log(\text{flux}_{i,j}^{\rm pred})$, $G_j = \log(\text{gain}_j)$, and $S_i = \log(\text{source}_i)$. We can then write the chi-square which we want to minimize as:

$$\chi^2 = \sum_{i,j} (M_{i,j}^{\rm obs} - G_j - S_i)^2$$

where we ignore the weights of the different measured flux levels. Taking the derivatives with respect to the parameters of interest (G_j, S_i), and setting them to 0, we determine the following set of equations which must be solved:

$$G_j \times N_i = \sum_i M_{i,j}^{\rm obs} - \sum_i S_i$$

$$S_j \times N_j = \sum_j M_{i,j}^{\rm obs} - \sum_j G_j$$

This set of equations can be solved iteratively, starting from the assumption that all chip gains are 1.0, ($G_j = 0$), or by supplying a guess for the chip gains. The result of this analysis is the measured chip gains and the measured source illumination levels for each of the input flat-field images. The chip gains can then be used to modify the flux levels on the master flat-field images.

We define the following function to perform the analysis discussed above:

```
bool pmFlatNormalization (psVector *sourceFlux, psVector *chipGains, psArray *fluxLevels);
```

The input array `fluxLevels` consists of N_i vectors, one per mosaic image. Each vector consists of N_j elements, each a measurement of the input flat-field image flux levels. All of these vectors must be constructed with the same number of elements, or the function will return an error. If a chip is missing from a particular image, that element should be set to `NaN`. The vector `chipGains` supplies initial guesses for the chip gains. If the vector contains the values 0.0 or `NaN` for any of the elements, the gain is set to the mean of the valid values. If the vector length does not match the number of chips, an warning is raised, all chip gain guesses will be set to 1.0, and the vector length modified to match the number of chips defined by the supplied `fluxLevels`. The `sourceFlux` input vector must be allocated (not `NULL`), but the routine will set the vector length to the number of source images regardless of the initial state of the vector. All vectors used by this function must be of type `PS_DATA_F64`.

8 Objects on Images

8.1 Overview

The process of finding, measuring, and classifying astronomical sources on images is one of the critical tasks of the IPP or any astronomical software system. In this section, we define structures and functions related to the task of source detection and measurement. The elements defined in this section are generally low-level components which can be connected together to construct a complete object measurement suite.

We first define the collection of structures needed to carry information about the detected sources. A major challenge is to define what we mean by an astronomical object in the context of image source detection. An astronomical object may be as simple as a stellar point source, or it may consist of a galaxy which has smooth extended structure; it may consist of an irregular galaxy or galaxy group with substantial and complex sub-structure, or it may consist of complex non-stellar structures such as planetary nebulae, reflection nebulae, outflows and jets.

The simplest objects (ie, stars) can be sufficiently modeled by the point-source function (PSF). More complex objects (such as simple, smooth galaxies), may have approximate analytical models which represent their morphology with more-or-less accuracy. In the extreme cases, the objects are not well modeled at all and must be represented in other ways. Thus, one aspect of our data structures must be elements to specify if an object has been represented by a model, what the model parameters are, and how well it is represented by the model. Another aspect of the data structures must be a representation of the pixels associated with the object so complex structures may be referenced without attempting to supply an analytical model. Finally, it is often useful to allow a single complex model to be represented as a collection of simpler contained structures which may be modeled. Thus, the representation of an object must be capable of identifying children, or substructures, of that object.

Two additional aspects must be considered. First, source detection need not be performed on a single image in isolation: it is necessary for multiple realizations of the same source in multiple images to be measured together (whether or not through simultaneous fitting in multiple bands or via application of the results from one image to another image). Second, it will be necessary to performed object measurements on pixels in which no source is actually detected. For example, this is a convenient way to provide flux upper limits at the locations of known objects.

In the discussion that follows, images are of type F32 and masks are of type U8.

8.2 Structures to Describe Sources

In the object analysis process, we will use specific mask values to mark the image pixels. The following structure defines the relevant mask values.

```
typedef enum {
    PSPHOT_MASK_CLEAR     = 0x00,
    PSPHOT_MASK_INVALID   = 0x01,
    PSPHOT_MASK_SATURATED = 0x02,
    PSPHOT_MASK_MARKED    = 0x08,
} psphotMaskValues;
```

8.2.1 pmSource and pmPeak

We define the following structure to represent a single source detected in a single image.

```
typedef struct {
  pmPeak *peak;            // description of peak pixel
  psImage *pixels;         // rectangular region including object pixels
  psImage *weight;         // Image variance.
  psImage *mask;           // Mask which marks pixels associated with objects.
  pmMoments *moments;      // basic moments measure for the object
  pmModel *modelPSF;       // PSF model parameters and type
  pmModel *modelEXT;       // FLT model parameters and type
  pmSourceType type;       // Best identification of object
  pmSourceMode mode;       // flags describing the model quality
  psArray *blends;         // array of other sources blended with this source
  float apMag;             // measured aperture magnitude for source
  float fitMag;            // measured model magnitude for source
  psRegion region;         // area on image covered by selected pixels
} pmSource;
```

A source has the capacity for several types of measurements. The simplest measurement of a source is the location and flux of the peak pixel associated with the source:

```
typedef struct {
  int x;                  // x-coordinate of peak pixel
  int y;                  // y-coordinate of peak pixel
  float counts;           // value of peak pixel (above sky?)
  pmPeakType class;       // description of peak
} pmPeak;
```

A peak pixel may have several features which may be determined when the peak is found or measured. These are specified by the `pmPeakType` enum. `PM_PEAK_LONE` represents a single pixel which is higher than its 8 immediate neighbors. The `PM_PEAK_EDGE` represents a peak pixel which touching the image edge. The `PM_PEAK_FLAT` represents a peak pixel which has more than a specific number of neighbors at the same value, within some tolerance:

```
typedef enum {
  PM_PEAK_LONE,            // isolated peak
  PM_PEAK_EDGE,            // peak on edge
  PM_PEAK_FLAT             // peak has equal-value neighbors
  PM_PEAK_UNDEF            // Undefined.
} pmPeakType;
```

8.2.2 pmMoments and source description

The pixels which contain the source are specified with the `psImage *pixels` element, a subimage of the image being analysed. Similarly, the `mask` element is a subimage of the corresponding mask image and the `weight` element is a subimage of the corresponding weight image (image varience). Since these are subimages, a collection of many objects may include overlapping pixels; care must be taken that pixel manipulations for one source do not unintentionally interfere with the other source pixels. The `mask` may be used to exclude any pixels which are not considered part of the source. Along with these pixel structures, we include the `psRegion region` element which defines the boundaries of the current associated subimages.

One of the simplest measurements which can be made quickly for an object are the object moments. We specify a structure to carry the moment information for a specific source:

```
typedef struct {
  float x;                // x-coord of centroid
  float y;                // y-coord of centroid
  float Sx;               // x-second moment
  float Sy;               // y-second moment
  float Sxy;              // xy cross moment
  float Sum;              // pixel sum above sky (background)
  float Peak;             // peak counts above sky
  float Sky;              // sky level (background)
  float SN;                   // approx signal-to-noise
  int   nPixels;          // number of pixels used
} pmMoments;
```

A collection of object moment measurements can be used to determine approximate object classes. The key to this analysis is the location and statistics (in the second-moment plane, σ_x vs σ_y) of the group of objects which are likely PSF objects. We define the following structure to identify the location and size of the psf clump in the second-moment plane.

```
typedef struct {
    float X;
```

```
    float dX;
    float Y;
    float dY;
} pmPSFClump;
```

A given source may be identified as most-likely to be one of several source types. The `pmSource` entry `pmSourceType` defines the current best-guess for this source.

```
typedef enum {
    PM_SOURCE_UNKNOWN,                  ///< no guess yet made
    PM_SOURCE_DEFECT,                   ///< a cosmic-ray
    PM_SOURCE_SATURATED,                ///< random saturated pixels
    PM_SOURCE_STAR,                     ///< a good-quality star
    PM_SOURCE_EXTENDED,                 ///< an extended object (eg, galaxy)
} pmSourceType;
```

The related element, `pmSourceMode mode`, holds a collection of flags which are used to indicate the status of the analysis for a source. These are defined below:

```
typedef enum {
    PM_SOURCE_DEFAULT    = 0x0000, ///< no flags are set
    PM_SOURCE_PSFMODEL   = 0x0001, ///< flags refer to the PSF model
    PM_SOURCE_EXTMODEL   = 0x0002, ///< flags refer to the EXT model
    PM_SOURCE_SUBTRACTED = 0x0004, ///< the model has been subtracted from the image
    PM_SOURCE_FITTED     = 0x0008, ///< the source has been fitted with a model
    PM_SOURCE_FAIL       = 0x0010, ///< the model fit failed
    PM_SOURCE_POOR       = 0x0020, ///< the model fit was poor (low S/N, etc)
    PM_SOURCE_PAIR       = 0x0040, ///< the model fit is one of a paired source
    PM_SOURCE_PSFSTAR    = 0x0080, ///< the source was used to construct the image PSF model
    PM_SOURCE_SATSTAR    = 0x0100, ///< the source is saturated
    PM_SOURCE_BLEND      = 0x0200, ///< the source is a blend with another source
    PM_SOURCE_LINEAR     = 0x0400, ///< the source was fitted with the linear PSF model
    PM_SOURCE_TEMPSUB    = 0x0800, ///< the source has been subtracted, but should be replaced
} pmSourceMode;
```

8.2.3 pmModel Source Model and Abstraction

An object's flux distribution may be modeled with some analytical function. The description of the model includes the model parameters and their errors, along with the fit χ^2. The model type is identified by code `type`, dynamically assigned based on the available models (see below). We discuss the details of these models in section A. The model parameters have 4 special elements. The first four elements represent aspects of the source which are not specified by the image PSF, even for point sources. These consist of, in order:

- the local sky
- the object normalization
- the x-coordinate
- the y-coordinate

should be include utility pointers to these parameters so that functions do not need to know the parameter sequence? (TBD)

The structure which carries the information about a given source model is defined below:

```
typedef struct {
  pmModelType type;         // model to be used
  psVector *params;         // parameter values
  psVector *dparams;        // parameter errors
  psF32 chisq;              // fit chisq
  psS32 nDOF;               // number of degrees of freedom
  psS32 nIter;              // number of iterations
  pmModelStatus status;     // fit status
  float radius;                    // fit radius actually used
} pmModel;
```

The `status` element carries the resulting success/failure status of an attempt to fit the model to the source:

```
typedef enum {
    PM_MODEL_UNTRIED,               ///< model fit not yet attempted
    PM_MODEL_SUCCESS,               ///< model fit succeeded
    PM_MODEL_NONCONVERGE,           ///< model fit did not converge
    PM_MODEL_OFFIMAGE,              ///< model fit drove out of range
    PM_MODEL_BADARGS                ///< model fit called with invalid args
} pmModelStatus;
```

We distinguish several ways in which an analytical model may be applied to a source. The PSF model represents the best fit of the image PSF to the specific object. In this case, the PSF-dependent parameters are specified for the object by the PSF, not by the fit. The EXT model represents the best fit of the given model to the object, with all parameters floating in the fit. Such a model would typically be used to represent and extended object, hence the abbreviation EXT. In some circumstances, a source may be fitted with a PSF model in which the position is held fixed, and not allowed to vary in the model fitting process. We identify such a model as FIX. Finally, we allow for the case in which two nearly-merged PSFs are fitted with a single 2-PSF model. We identify such a model as DBL. The `pmSource` structure contains a pointer to both a PSF and an EXT model, allowing any source to carry information about both possible fitting modes **not clear that we actually use this information; we might be better off simply distinguishing with one of the pmSourceMode flags (TBD)** . The value of the model at a specific coordinate can be determined by calling the function:

```
psF32 pmModelEval(pmModel *model, psImage *image, psS32 col, psS32 row);
```

For this function, the values of `col,row` are in the `image` coordinates, which may be a subimage, while the reference coordinate for the model is in the parent image coordinates.

In the `pmSource` structure, the elements `apMag` and `fitMag` are used to carry the measured magnitude of the source determined either from aperture photometry or from the integral of the fitted model function. The element `blends` is used to carry pointers to the collection of sources which were found to be blended with this source. Only the primary source of a blend group carries this information (see Section **??**).

Every model instance belongs to a class of models, defined by the value of the `pmModelType type` entry. Various functions need access to information about each of the models. Some of this information varies from model to model, and may depend on the current parameter values or other data quantities. In order to keep the code from requiring the information about each model to be coded into the low-level fitting routines, we define a collection of functions which allow us to abstract this type of model-dependent information. These generic functions take the model type and return

the corresponding function pointer for the specified model. Each model is defined by creating this collection of specific functions, and placing them in a single file for each model. We define the following structure to carry the collection of information about the models.

```
typedef struct {
    char *name;
    int nParams;
    pmModelFunc          modelFunc;
    pmModelFlux          modelFlux;
    pmModelRadius        modelRadius;
    pmModelLimits        modelLimits;
    pmModelGuessFunc     modelGuessFunc;
    pmModelFromPSFFunc   modelFromPSFFunc;
    pmModelFitStatusFunc modelFitStatusFunc;
} pmModelGroup;
```

Each entry in the `pmModelGroup` defines the information needed by the system to specify a model. The function types define above are

```
typedef psMinimizeLMChi2Func pmModelFunc;
typedef psF64 (*pmModelFlux)(const psVector *params);
typedef psF64 (*pmModelRadius)(const psVector *params, double flux);
typedef bool (*pmModelLimits)(psVector **beta_lim, psVector **params_min, psVector **params_max);
typedef bool (*pmModelGuessFunc)(pmModel *model, pmSource *source);
typedef bool (*pmModelFromPSFFunc)(pmModel *modelPSF, pmModel *modelFLT, pmPSF *psf);
typedef bool (*pmModelFitStatusFunc)(pmModel *model);
```

Each of these functions is found for a given model by calling the corresponding lookup function:

```
pmModelFunc          pmModelFunc_GetFunction (pmModelType type);
pmModelFlux          pmModelFlux_GetFunction (pmModelType type);
pmModelRadius        pmModelRadius_GetFunction (pmModelType type);
pmModelLimits        pmModelLimits_GetFunction (pmModelType type);
pmModelGuessFunc     pmModelGuessFunc_GetFunction (pmModelType type);
pmModelFromPSFFunc   pmModelFromPSFFunc_GetFunction (pmModelType type);
pmModelFitStatusFunc pmModelFitStatusFunc_GetFunction (pmModelType type);
```

`pmModelFunc` is the function used to determine the value of the model at a specific coordinate, and is the one used by `psMinimizeLMChi2`.

`pmModelFlux` returns the total integrated flux for the given input parameters.

`pmModelRadius` returns the scaling radius at which the flux of the model matches the specified flux. This presumes that the model is a function of an elliptical contour.

`pmModelLimits` sets the parameter limit vectors for the function.

`pmModelGuessFunc` generates an initial guess for the model based on the provided source statistics (moments and pixel values as needed).

`pmModelFromPSFFunc` takes as input a representation of the psf and a value for the model and fills in the PSF parameters of the model. The input primarily relies upon the centroid coordinates of the input model, thought the normalization may potentially be used.

`pmModelFitStatusFunc` returns a true or false values based on the success or failure of a model fit. the success is determined by quantities such as the chisq or the signal-to-noise.

In addition, the following functions are useful for interacting with the collection of models:

```
int                 pmModelParameterCount (pmModelType type);
```

This function returns the number of parameters used by the listed function.

```
char               *pmModelGetType (pmModelType type);
pmModelType         pmModelSetType (char *name);
```

These two functions provide translations between the user-space model names and the internal model type codes. The model type codes are not necessarily maintained between compilations of the program; the name should be used to transfer models between programs or systems.

8.2.4 pmGrowthCurve

When the photometry of source is measured in a fixed aperture, there is always a fraction of the source light which falls outside of the aperture. The resulting aperture magnitude is thus larger (ie, fainter) than the actual source. As the aperture is increased, the amount of loss decreases and the measured magnitude increases. This trend is the curve of growth for the source. We use the following structure to carry information about the curve of growth. We use the PSF model to measure the curve of growth for an image.

```
typedef struct {
    psVector *radius;
    psVector *apMag;
    psF32 refRadius;
    psF32 maxRadius;
    psF32 fitMag;
    psF32 apRef;   // apMag[refRadius]
    psF32 apLoss;  // fitMag - apRef
} pmGrowthCurve;
```

In this structure, `radius` is a monotonically increasing sequence of radius values (in pixels). The `apMag` vector contains the measured magnitude at any of these radius: this is the curve-of-growth trend. The remaining entries summaries the relationship: `refRadius` is the global reference radius used for this image; `maxRadius` is the outermost radius at which the curve of growth was measured; `fitMag` is the fitted PSF model magnitude integrated to infinity; `apRef` is the aperture magnitude at the reference radius; `apLoss` is the difference between the aperture magnitude at the reference radius and the fitted model magnitude. A few related functions are specified to interact with the curve of growth:

```
pmGrowthCurve *pmGrowthCurveAlloc (psF32 minRadius, psF32 maxRadius, psF32 dRadius);
```

This function allocates a `pmGrowthCurve` structure and fills in the `radius` vector (see psLib SDRS `psVectorCreate`). It does *not* allocate the `apMag` vector.

```
psF32 pmGrowthCurveCorrect (pmGrowthCurve *growth, psF32 radius);
```

This function accepts a `growth` curve structure and returns the correction between the specified radius and the reference radius ($apMag(refRadius) - apMag(radius)$).

The following two functions are used to search the growth curve to the corresponding radius entry:

```
int psVectorBracket (psVector *index, psF32 key, bool above);
psF32 psVectorInterpolate (psVector *index, psVector *value, psF32 key);
```

8.2.5 Aperture Trends

With PSF model fitting, there is always some discrepancy between the model of the PSF and the actual PSF. As a result, the measured flux from the model will not represent exactly the flux of the source. It is necessary to measure the correction between the model and the actual source flux. One way to perform this measurement is to compare the model flux with the flux measured for bright stars within a fixed aperture. The quantity to be measured is $dA = m_{\rm aperture} - m_{\rm fit}$. In practice, dA exhibits variations as a function of the source position (x, y) and the source flux. The variations as a function of source position can be understood as a change in the PSF model error as a function of position due to the changing shape of the PSF (despite the varying PSF model, it is possible that the fitted model yields positional variations in the residual flux). The variations in dA as a function of magnitude can be understood as the result of a bias in the local background measurement (for the fainter sources) and as a result of non-linearity in the detector setting on the bright end. We use a 4D polynomial to represent these trends. The first two dimensions of the polynomial represent the variation of dA as a function of x, y; we provide helper functions to define 1st and 2nd order polynomials in x, y. The next two dimensions are fitted independently (no cross terms). The first represents the variation as a function of $r^2/flux$, where r is the aperture radius used to measure dA; this is the scaling of a magnitude error in the presence of a constant error in the sky level. The last dimension represents the variation of dA as a function of the stellar flux.

The following forms of the aperture correction model may be selected by the user:

```
typedef enum {
    PM_PSF_NONE,
    PM_PSF_CONSTANT,
    PM_PSF_SKYBIAS,
    PM_PSF_SKYSAT,
    PM_PSF_XY_LIN,
    PM_PSF_XY_QUAD,
    PM_PSF_SKY_XY_LIN,
    PM_PSF_SKYSAT_XY_LIN,
    PM_PSF_ALL
} pmPSF_ApTrendOptions;
```

The following utility function sets the aperture correction model coefficient masks to select the specific desired coefficients:

```
bool pmPSF_MaskApTrend (pmPSF *psf, pmPSF_ApTrendOptions option);
```

8.2.6 pmPSF, pmPSFtry, and PSF model

It is useful to generate a model to define the point-spread-function which describes the flux distribution for unresolved sources in an image. In general, the PSF varies with position in the image. We allow any of the source models defined for the `pmModel` to represent the PSF. For a given source model, the 2D spatial variation of all of the source parameters,

except the first four PSF-independent parameters, are represented as polynomial, stored in a psArray. The structure also contains the aperture correction model (ApTrend) and the curve-of-growth model (growth). The additional elements are: ApResid, the constant term in the aperture correction model; dApResid, the residual scatter for bright sources ($S/N > 100$) after applying the aperture correction; skyBias, the measured average bias in the sky measurement; skySat, the scaling of the flux-dependent portion of the correction.

The other elements of the structure define the quality of the PSF determination.

```
typedef struct {
    pmModelType type;                       ///< PSF Model in use
    psArray *params;                       ///< Model parameters (psPolynomial2D)
    psPolynomial4D *ApTrend;               ///< ApResid vs (x,y,rflux) (rflux = ten(0.4*mInst)
    pmGrowthCurve *growth;                ///< apMag vs Radius
    float ApResid;                        ///< ???
    float dApResid;                        ///< ???
    float skyBias;                        ///< ???
    float skySat;                        ///< ???
    float chisq;                        ///< PSF goodness statistic
    int nPSFstars;                        ///< number of stars used to measure PSF
    int nApResid;                        ///< number of stars used to measure ApResid
} pmPSF;

pmModel *pmModelFromPSF (pmModel *model, pmPSF *psf);
```

This function constructs a pmModel instance based on the pmPSF description of the PSF. The input is a pmModel with at least the values of the centroid coordinates (possibly normalization if this is needed) defined. The values of the PSF-dependent parameters are specified for the specific realization based on the coordinates of the object.

```
bool pmPSFFromModels (pmPSF *psf, psArray *models, psVector *mask);
```

This function takes a collection of pmModel fitted models from across a single image and builds a pmPSF representation of the PSF. The input array of model fits may consist of entries to be ignored (noted by a non-zero mask entry). The analysis of the models fits a 2D polynomial for each parameter to the collection of model parameters as a function of position (and normalization?). In this process, some of the input models may be marked as outliers and excluded from the fit. These elements will be marked with a specific mask value (1 == PSFTRY_MASK_OUTLIER).

We definet he following two functions to convert the PSF model parameters into a collection of elements on a metadata structure, and vice versa. These can be used to read and write PSFs to a file and or a database.

```
psMetadata *pmPSFtoMD (psMetadata *metadata, pmPSF *psf);
pmPSF *pmPSFfromMD (psMetadata *metadata);
```

We have the capability to test several different model functions in an attempt to build an accurate PSF for an image. The complete set of data needed to build and test as specific PSF model is carried by the pmPSFtry structure:

```
typedef struct {
    pmModelType modelType;
    pmPSF      *psf;
    psArray    *sources;      // pointers to the original sources
    psArray    *modelEXT;     // model fits, floating parameters
    psArray    *modelPSF;     // model fits, PSF parameters
```

```
    psVector   *mask;
    psVector   *metric;
    psVector   *fitMag;
} pmPSFtry;
```

This structure contains a pointer to the collection of `sources` which will be used to test the PSF model form. It lists the `pmModelType type` of model being tests, and contains an element to store the resulting `psf` representation. In addition, this structure carries the complete collection of FLT (floating parameter) and PSF (fixed parameter) model fits to each of the sources `modelFLT` and `modelPSF`. It also contains a mask which is set by the model fitting and psf fitting steps. For each model, the value of the quality metric is stored in the vector `metric` and the fitted instrumental magnitude is stored in `fitMag`. The quality metric for the PSF model is the aperture magnitude minus the fitted magnitude for each source.

This collection of aperture residuals is examined in the analysis process, and a linear trend of the residual with the inverse object flux (ie, $10^{0.4*mag}$) is fitted. The result of this fit is a measured sky bias (systematic error in the sky measured by the fits), an effective infinite-magnitude aperture correction (`ApResid`), and the scatter of the aperture correction for the ensemble of PSF stars (`dApResid`). The ultimate metric to intercompare multiple types of PSF models is the value of the aperture correction scatter.

The following functions are used to try out a single PSF model.

```
pmPSFtry *pmPSFtryModel (psArray *sources, char *modelName, float RADIUS);
```

This function takes the input collection of sources and performs a complete analysis to determine a PSF model of the given type (specified by model name). The result is a `pmPSFtry` with the results of the analysis.

```
bool pmPSFtryMetric (pmPSFtry *try, float RADIUS);
```

This function is used to measure the PSF model metric for the set of results contained in the `pmPSFtry` structure.

The following datatype defines the masks used by the `pmPSFtry` analysis to identify sources which should or should not be included in the analysis.

```
enum {
    PSFTRY_MASK_CLEAR    = 0x00,
    PSFTRY_MASK_OUTLIER  = 0x01, // 1: outlier in psf polynomial fit (provided by psPolynomials)
    PSFTRY_MASK_EXT_FAIL = 0x02, // 2: ext model failed to converge
    PSFTRY_MASK_PSF_FAIL = 0x04, // 3: psf model failed to converge
    PSFTRY_MASK_BAD_PHOT = 0x08, // 4: invalid source photometry
    PSFTRY_MASK_ALL      = 0x0f,
} pmPSFtryMaskValues;

typedef enum {
  PM_CONTOUR_CRUDE
} pmContourType;
```

Allocators for the above structures are defined as follows:

```
pmSource   *pmSourceAlloc ();
pmPeak     *pmPeakAlloc (int x, int y, float counts, psPeakType class);
pmMoments  *pmMomentsAlloc ();
pmModel    *pmModelAlloc (pmModelType type);
```

8.3 Basic Object Detection APIs

In this section, we specify a collection of basic functions which operate on images and sources. We define them roughly in order in which we expect to use them in a basic object detection process.

```
psVector *pmFindVectorPeaks(const psVector *vector, float threshold);
```

Find all local peaks in the given vector above the given threshold. A peak is defined as any element with a value greater than its two neighbors and with a value above the threshold. Two types of special cases must be addressed. Equal value elements: If an element has the same value as the following element, it is not considered a peak. If an element has the same value as the preceding element (but not the following), then it is considered a peak. Note that this rule (arbitrarily) identifies flat regions by their trailing edge. Edge cases: At start of the vector, the element must be higher than its neighbor. At the end of the vector, the element must be higher or equal to its neighbor. These two rules again places the peak associated with a flat region which touches the image edge at the image edge. The result of this function is a vector containing the coordinates (element number) of the detected peaks (type `psU32`).

```
psArray *pmFindImagePeaks(const psImage *image, float threshold);
```

Find all local peaks in the given image above the given threshold. This function should find all row peaks using `pmFindVectorPeaks`, then test each row peak and exclude peaks which are not local peaks. A peak is a local peak if it has a higher value than all 8 neighbors. If the peak has the same value as its +y neighbor or +x neighbor, it is NOT a local peak. If any other neighbors have an equal value, the peak is considered a valid peak. Note two points: first, the +x neighbor condition is already enforced by `pmFindVectorPeaks`. Second, these rules have the effect of making flat-topped regions have single peaks at the (+x,+y) corner. When selecting the peaks, their type must also be set. The result of this function is an array of `pmPeak` entries.

```
psArray *pmPeaksSubset(psArray *peaks, float maxvalue, const psRegion valid);
```

Create a new peaks array, removing certain types of peaks from the input array of peaks based on the given criteria. Peaks should be eliminated if they have a peak value above the given maximum value limit or if the fall outside the valid region. The result of the function is a new array with a reduced number of peaks.

```
bool pmSourceDefinePixels(pmSource *mySource,
                          pmReadout *readout,
                          psF32 x,
                          psF32 y,
                          psF32 Radius)

bool pmSourceRedefinePixels(pmSource *mySource,
                            pmReadout *readout,
                            psF32 x,
                            psF32 y,
                            psF32 Radius)
```

The first form defines `psImage` subarrays (pixel, weight, and mask) for the source located at coordinates `x,y` on the image set defined by `readout`. The pixels defined by this operation consist of a square window (of full width $2Radius+1$) centered on the pixel which contains the given coordinate, in the frame of the readout. The window is defined to have

limits which are valid within the boundary of the `readout` image, thus if the radius would fall outside the image pixels, the subimage is truncated to only consist of valid pixels. If `readout->mask` or `readout->weight` are not `NULL`, matching subimages are defined for those images as well. This function fails if no valid pixels can be defined (x or y less than Radius, for example). This function should be used to define a region of interest around a source, including both source and sky pixels. The second form accepts an existing source and redefines the pixels if the requested radius encompasses more pixels than the existing images.

```
pmSource *pmSourceLocalSky(pmSource *source,
                           psStatsOptions statsOptions,
                           float Radius)
```

Measure the local sky in the vicinity of the given `source`. The `Radius` defines the square aperture in which the moments will be measured. This function assumes the source pixels have been defined, and that the value of `Radius` here is smaller than the value of `Radius` used to define the pixels. The annular region not contained within the radius defined here is used to measure the local background in the vicinity of the source. The local background measurement uses the specified statistic passed in via the `statsOptions` entry. This function allocates the `pmMoments` structure. The resulting sky is used to set the value of the `pmMoments.sky` element of the provided `pmSource` structure.

```
bool pmSourceMoments(pmSource *source, float radius);
```

Measure source moments for the given `source`, using the value of `source.moments.sky` provided as the local background value and the peak coordinates as the initial source location. The resulting moment values are applied to the `source.moments` entry, and the source is returned. The moments are measured within the given circular radius of the `source.peak` coordinates. The return value indicates the success (TRUE) of the operation. This function also measures the approximate signal-to-noise ratio of the source (`source.SN`) based on the total number of source counts divided by the square-root of the total source variance, as determined from the weight image.

```
pmPSFClump pmSourcePSFClump(psArray *sources, psMetadata *metadata);
```

We use the source moments to make an initial, approximate source classification, and as part of the information needed to build a PSF model for the image. As long as the PSF shape does not vary excessively across the image, the sources which are represented by a PSF (the start) will have very similar second moments. The function `pmSourcePSFClump` searches a collection of `sources` with measured moments for a group with moments which are all very similar. The function returns a `pmPSFClump` structure, representing the centroid and size of the clump in the σ_x, σ_y second-moment plane.

The goal is to identify and characterize the stellar clump within the σ_x, σ_y plane. To do this, an image is constructed to represent this plane. The units of σ_x and σ_y are in image pixels. A pixel in this analysis image represents 0.1 pixels in the input image. The dimensions of the image need only be 10 pixels. The peak pixel in this image (above a threshold of half of the image maximum) is found. The coordinates of this peak pixel represent the 2D mode of the σ_x, σ_y distribution. The sources with σ_x, σ_y within 0.2 pixels of this value are then used to calculate the median and standard deviation of the σ_x, σ_y values. These resulting values are returned via the `pmPSFClump` structure.

The return value indicates the success (TRUE) of the operation.

limit the S/N of the candidate sources (part of Metadata)? (TBD)

save the clump parameters on the Metadata (TBD)

```
bool pmSourceRoughClass(psArray *sources, psMetadata *metadata, pmPSFClump clump)
```

Based on the specified data values, make a guess at the source classification. The sources are provides as a `psArray` of `pmSource` entries. Definable parameters needed to make the classification are provided to the routine with the `psMetadata` structure. The rules below refer to values which can be extracted from the metadata using the given keywords. Except as noted, the data type for these parameters are `psF32`.

The following rules are used to make the classification. The number of saturated pixels are counted, based on the mask having the `PSPHOT_MASK_SATURATED` bit set. Sources which are greater than 1σ larger than the `pmPSFClump` center in both dimensions and which have more than a single saturated pixel are identified as being a likely saturated star (`type = PM_SOURCE_STAR, mode = PM_SOURCE_SATSTAR`). Sources which are not so large but which have multiple saturated pixels are identified as saturated regions, ie bleed trails or hot columns (`type = PM_SOURCE_SATURATED`).

Sources with

$$\sigma_x < 0.05$$

or

$$\sigma_y < 0.05$$

should be identified as type `PM_SOURCE_DEFECT` (likely cosmic ray pixel).

Sources with

$$\sigma_x > \mathrm{CLUMP}_x + 3\mathrm{CLUMP}_{dx}$$

and

$$\sigma_y > \mathrm{CLUMP}_y + 3\mathrm{CLUMP}_{dy}$$

should be identified as type `PM_SOURCE_EXTENDED`.

All other sources should be identified as type `PM_SOURCE_STAR`. Of these sources, the mode should be set to `PM_SOURCE_PSFSTAR` for any sources with SN greater than `PSF_SN_LIM` which are within 1.5σ of the PSF clump center. These sources are used to determine a guess at the shape of the PSF, based on the collection of σ_x and σ_y values.

8.4 Object Fitting

We need a way to fit a particular functional model to an object. PSLib includes the `psMinimizeLMChi2` and `psMinimizePowell` functions, which form the core of this processes. However, additional support functions and wrapping functions are necessary for the specific case of source fitting. The operations can be broken down into discrete steps:

8.4.1 Identify the pixels of interest

8.4.2 Make a guess at the model parameters. For some models, the parameters may be guessed based on only the moments. For others, additional measurements must be made.

8.4.3 Construct the input vectors from the pixels of interest.

8.4.4 Apply fitting function `psMinimizeLMChi2()`

8.4.5 Construct model image.

8.4.6 Subtract model from image.

```
bool pmSourceModelGuess(pmSource *source, const psImage *image, pmModelType model);
```

Convert available data to an initial guess for the given model. This function allocates a `pmModel` entry for the `pmSource` structure based on the provided model selection. The method of defining the model parameter guesses are determined by using `pmModelGuessFunc_GetFunction` to determine the guess function for the model of interest. The returned function is called and the guess values are used to set the model parameters. The function returns `TRUE` on success or `FALSE` on failure.

```
psArray *pmSourceContour(const pmSource *source, const psImage *image, float level, pmContourType type);
```

Find points in a contour for the given source at the given level. If `type` is `PM_CONTOUR_CRUDE`, the contour is found by starting at the source peak, running along each pixel row until the level is crossed, then interpolating to the level coordinate for that row. This is done for each row, with the starting point determined by the midpoint of the previous row, until the starting point has a value below the contour level. The returned contour consists of two vectors giving the x and y coordinates of the contour levels. This function may be used as part of the model guess inputs.

Other contour types may be specified in the future for more refined contours (TBD)

```
bool pmSourceFitModel(pmSource *source, psImage *image);
```

Fit the requested model to the specified source. The starting guess for the model is given by the input `source.model` parameter values. The pixels of interest are specified by the `source.pixels` and `source.mask` entries. This function calls `psMinimizeLMChi2()` on the image data. The function returns `TRUE` on success or `FALSE` on failure.

```
bool pmModelFitStatus (pmModel *model);
```

This function wraps the call to the model-specific function returned by `pmModelFitStatusFunc_GetFunction`. The model-specific function examines the model parameters, parameter errors, Chisq, S/N, and other parameters available from `model` to decide if the particular fit was successful or not.

```
bool pmSourceAddModel(psImage *image, pmSource *source, bool center, bool sky);
bool pmSourceSubModel(psImage *image, pmSource *source, bool center, bool sky);
```

Add or subtract the given source model flux to/from the provided image. The boolean option `center` selects if the source is re-centered to the image center or if it is placed at its centroid location. The boolean option `sky` selects if the background sky is applied (`TRUE`) or not. The pixel range in the target image is at most the pixel range specified by the `source.pixels` image. The success status is returned.

```
bool pmSourcePhotometry (float *fitMag,  // integrated fit magnitude
                         float *obsMag,  // aperture flux magnitude
                         pmModel *model, // model used for photometry
                         psImage *image, // image pixels to be used
                         psImage *mask   // mask of pixels to ignore
);
```

The function returns both the magnitude of the fit, defined as $-2.5 \log \mathrm{flux}$, where the flux is integrated under the model, theoretically from a radius of 0 to infinity. In practice, we integrate the model beyond 50σ. The aperture magnitude is defined as $-2.5 \log \mathrm{flux}$, where the flux is summed for all pixels which are not excluded by the aperture mask. The model flux is calculated by calling the model-specific function provided by `pmModelFlux_GetFunction`.

```
int pmSourceDophotType (pmSource *source);
```

This function converts the source classification into the closest available approximation to the Dophot classification scheme. The following list gives the correspondence:

```
PM_SOURCE_DEFECT:       8
PM_SOURCE_SATURATED: 8
PM_SOURCE_SATSTAR: 10
PM_SOURCE_PSFSTAR: 1
PM_SOURCE_GOODSTAR: 1
PM_SOURCE_POOR_FIT_PSF: 7
PM_SOURCE_FAIL_FIT_PSF: 4
PM_SOURCE_FAINTSTAR: 4
PM_SOURCE_GALAXY: 2
PM_SOURCE_FAINT_GALAXY: 2
PM_SOURCE_DROP_GALAXY: 2
PM_SOURCE_FAIL_FIT_GAL: 2
PM_SOURCE_POOR_FIT_GAL: 2
PM_SOURCE_OTHER: ?
```

```
int pmSourceSextractType (pmSource *source);
```

This function converts the source classification into the closest available approximation to the Sextractor classification scheme. **the correspondence is not yet defined (TBD)** .

9 Image Combination

The image combination for Pan-STARRS will employ an iterative approach, in order to identify cosmic rays. The first pass involves transforming and combining the input images, and noting pixels which are apparently deviant. These pixels are examined in further detail, before a subset of them are declared to be bad, whereupon these pixels are re-transformed, and the images are combined properly. Here we introduce two functions which will perform the combination and examination steps. Prototype code exists for each of these functions. **For further details, see the document about image combination for Pan-STARRS. (TBD)**

9.1 Combining images

```
psImage *pmCombineImages(psImage *combined, // Combined image
                         psArray **questionablePixels, // Array of rejection masks
                         const psArray *images, // Array of input images
                         const psArray *errors, // Array of input error images
                         const psArray *masks,// Array of input masks
                         unsigned int maskVal, // Mask value
                         const psPixels *pixels, // Pixels to combine
                         int numIter,   // Number of rejection iterations
                         float sigmaClip, // Number of standard deviations at which to reject
                         const psStats *stats // Statistics to use in the combination
                         );
```

`pmCombineImages` shall combine the input `images`, returning the `combined` image and a list of `questionablePixels` in each input image. The array of error images, `errors`, shall be used to calculate the value

in the combined image and the list of questionable pixels, if non-NULL. Pixels whose corresponding value in the array of mask images, masks, matches maskVal shall be masked from the combination. The images, errors and masks arrays, if non-NULL, shall all carry the same number of images; otherwise the function shall generate an error and return NULL. The sizes of all images in the images, errors and masks arrays shall be identical; otherwise the function shall generate an error and return NULL.

If pixels is non-NULL, only those pixels specified shall be combined. The combination consists of numIter iterations in which a stack of pixels is combined using the specified stats. In each iteration, questionable pixels are identified as lying more than sigmaClip standard deviations from the combined value; these pixels are excluded from the stack for the next iteration. The value for the combined image is that produced by the *first* iteration (i.e., with no pixels excluded except those which have their corresponding mask match the maskVal); this allows subsequent calls to the function to only act on a small fraction of the pixels, since questionable pixels identified in the first call of the function will be properly rejected at a later point (see the example, below).

In the event that images or stats are NULL, the function shall generate an error and return NULL.

9.2 Rejecting pixels

```
psArray *pmRejectPixels(const psArray *images, // Array of input images
                        const psArray *masks, // Array of masks for input images
                        const psArray *pixels, // These are the pixels which were rejected in the combinati
                        const psArray *inToOut, // Transformations from input to output system
                        const psArray *outToIn, // Transformations from output to input system
                        float rejThreshold, // Rejection threshold
                        float gradLimit // Gradient limit
                        );
```

This algorithm will change: an addition will be made to avoid masking pixels in the wings of a star when combining images taken in different seeing, and the gradient limit criteria will be changed. (TBD)

pmRejectPixels inspects those questionable pixels identified by pmCombineImages to determine if they are truly discrepant. This inspection is performed in the coordinate frame of the detector, where the pixels haven't been smeared by transformation. Two tests are applied to each of the images:

9.2.1 The list of questionable pixels for an image is converted to an image which is transformed back to the coordinate frame of the detector. Those pixels in the detector frame which have a value exceeding rejThreshold are suspected cosmic rays and subjected to the next test. Depending on the value of the rejThreshold, this test basically amounts to demanding that questionable pixels neighbor each other in the transformed image.

9.2.2 The cores of point sources may mimic a cosmic ray, especially in under-sampled images. To minimize flagging stars as cosmic rays, we determine the gradient around the pixel of interest; if the gradient is large, then the pixel is likely the core of a point source. In order to reliably measure the gradient in the presence of a suspected cosmic ray, we use the companion images — the gradient is the mean gradient at the corresponding position on the other images. In order to calculate the corresponding positions, the inToOut and outToIn transformations are required. If the gradient is less than gradLimit, then the pixel is identified as a cosmic ray.

The function shall return an array of psPixels, one for each of the input images, containing pixels that have been identified as cosmic rays according to the above criteria.

If any of the input pointers are NULL, then the function shall generate an error and return NULL.

9.3 Example

Here is an example of what the image combination routine looks like, demonstrating how the various pieces fit together. The inputs are:

- `psArray *inputs`: Input detector images, each a `psImage` of type `psF32`
- `psArray *inputMask`: Input mask images, each a `psImage` of type `psU8`
- `psArray *inputsErr`: Input error images, each a `psImage` of type `psF32`
- `psPlaneTransform *skyToDetector`: Maps from sky coordinates to detector coordinates, each a `psPlaneTransform`
- `psRegion *combineRegion`: Sky coordinate pixels to combine
- `int numIter`: Number of iterations in combination
- `float rejThreshold`: Threshold for rejection
- `float gradLimit`: Limit for gradient

The output is the combined image.

```
psArray *transformed = psArrayAlloc(nImages); // Array of transformed images
psArray *transformedErr = psArrayAlloc(nImages); // Array of transformed error images
psArray *transformedMask = psArrayAlloc(nImages); // Array of masks for transformed images

for (int i = 0; i < nImages; i++) {
    psPixels *blanks = NULL;        // List of blank pixels
    transformed->data[i] = psImageTransform(NULL, &blanks, inputs->data[i],
                                            inputMask->data[i], inputMaskVal, NAN, skyToDetector,
                                            combineRegion, NULL, PS_INTERPOLATE_BILINEAR);
    transformedErr->data[i] = psImageTransform(NULL, NULL, inputsErr->data[i], inputMask->data[i],
                                               inputMaskVal, NAN, skyToDetector, combineRegion, NULL,
                                               PS_INTERPOLATE_BILINEAR_VARIANCE);
    psImage *skyImage = transformed->data[i]; // Dereference the transformed image
    psRegion *blankRegion = psRegionAlloc(0, 0, skyImage->numCols, skyImage->numRows); // Size of
                                                                                       // transformed
                                                                                       // image
    transformedMask->data[i] = psPixelsToMask(NULL, blanks, *blankRegion, PS_MASK_BLANK);
    psFree(blankRegion);
    psFree(blanks);
}

psArray *rejected = NULL;           // Array of rejected pixel lists
psStats *combineStats = psStatsAlloc(PS_STAT_SAMPLE_MEAN); // Statistic to use in doing the combination
psImage *combined = pmCombineImages(NULL, &rejected, transformed, transformedErr, transformedMask, 0,
                                    NULL, numIter, sigmaClip, combineStats); // Combined image
psArray *bad = pmRejectPixels(inputs, rejected, NULL, skyToDetector, rejThreshold, gradLimit); // Bad pix
psPixels *combinePixels = NULL;     // Pixels to combine
for (int i = 0; i < nImages; i++) {
    psPixels *badSource = psPixelsTransform(NULL, bad->data[i], skyToDetector); // Bad pixels on the input
    psImage *badMask = psPixelsToMask(NULL, badSource, PS_MASK_COSMICRAY); // Mask image for the input
    (void)psBinaryOp(inputMask->data[i], inputMask->data[i], "|", badMask); // Put CRs into original mask
    psFree(badSource);
    psFree(badMask);

    combinePixels = psPixelsConcatenate(redo, bad->data[i]);

    // Update transformed image
```

```
        psPixels *blanks = NULL;        // List of blank pixels
        transformed->data[i] = psImageTransform(transformed->data[i], &blanks, inputs->data[i],
                                                inputMask->data[i], inputMaskVal | PS_MASK_COSMICRAY, NAN,
                                                skyToDetector, combineRegion, bad->data[i],
                                                PS_INTERPOLATE_BILINEAR);
        transformedErr->data[i] = psImageTransform(transformedErr->data[i], NULL, inputsErr->data[i],
                                                   inputMask->data[i], inputMaskVal | PS_MASK_COSMICRAY,
                                                   NAN, skyToDetector, combineRegion, bad->data[i],
                                                   PS_INTERPOLATE_BILINEAR_VARIANCE);
        psImage *skyImage = transformed->data[i]; // Dereference the transformed image
        psRegion *blankRegion = psRegionAlloc(0, 0, skyImage->numCols, skyImage->numRows); // Size of
                                                                                           // transformed
                                                                                           // image
        transformedMask->data[i] = psPixelsToMask(transformedMask->data[i], blanks, *blankRegion,
                                                  PS_MASK_BLANK);
        psFree(blankRegion);
        psFree(blanks);
    }
    psFree(bad);

    // Combine with no rejection
    combined = pmCombineImages(combined, NULL, transformed, transformedErr, transformedMask,
                               PS_MASK_BLANK, combinePixels, 0, 0.0, combineStats);
    psFree(combineStats);
    psFree(combinePixels);
    psFree(transformed);
    psFree(transformedErr);
    psFree(transformedMask);
```

10 Image Subtraction

Image subtraction is arguably the best method of identifying faint variable sources in images with different point-spread functions. It relies on fitting for a convolution kernel that minimizes the residuals in subtracting small regions of the image. The use of a convolution kernel consisting of a linear combination of basis functions allows the problem to be solved with only modest computing power.

10.1 The kernels

We will allow for the use of two convolution kernels. The first is that employed by the popular image subtraction program, ISIS, consisting of Gaussians modified by polynomials:

$$B_{ijk}(u,v) = e^{-(u^2+v^2)/2\sigma_i^2} u^j v^k \tag{5}$$

The second simply consists of delta functions, which we refer to as POIS (Pan-STARRS Optimal Image Subtraction):

$$B_{ij}(u,v) = \delta(u-i)\,\delta(v-j) \tag{6}$$

For further details, see the document about image subtraction for Pan-STARRS. (TBD) The former is widely used, while the second appears to be equally useful and faster, though not as tried and proven.

```
typedef enum {
    PM_SUBTRACTION_KERNEL_POIS,         // POIS kernel --- delta functions
    PM_SUBTRACTION_KERNEL_ISIS          // ISIS kernel --- gaussians modified by polynomials
} pmSubtractionKernelsType;
```

In order to simplify the book-keeping for the kernels, we will define a `pmSubtractionKernels`, which keeps track of the details of the each of the kernel basis functions:

```
typedef struct {
    pmSubtractionKernelType type;       // Type of kernels --- allowing the use of multiple kernels
    int size;                           // Size of kernel in x and y
    int spatialOrder;                   // Maximum order of spatial variations
    psVector *u, *v;                    // Offset (for POIS) or polynomial order (for ISIS)
    psVector *sigma;                    // Width of Gaussian (for ISIS)
    psVector *xOrder, *yOrder;          // Spatial polynomial order (for all)
    int subIndex;                       // Index of kernel to be subtracted to maintain flux conservation
    psArray *preCalc;                   // Array of images containing pre-calculated kernel (to
                                        // accelerate ISIS; don't use for POIS)
} pmSubtractionKernels;
```

This structure caters for both choices of kernel type. For a POIS kernel, the `u` and `v` vectors shall be set to the coordinates for the delta functions for the corresponding kernel. For an ISIS kernel, the `sigma` vector shall be set to the Gaussian widths and the `u` and `v` vectors shall be set to the orders of the modifying polynomials for the corresponding kernel. For both choices of kernel, the `xOrder` and `yOrder` vectors specify the order of the spatial variation.

In order to maintain flux conservation when the kernel is spatially variable, we need to treat one kernel in the set differently. The convolutions for this kernel, identified by the `subIndex`, are calculated in the usual way, while all others have the `subIndex` kernel subtracted from them. For details, see the paper by Alard (2000, A&AS, 144, 363).

Since the ISIS kernels are continuous functions, it is worth pre-calculating them instead of calculating them each time they are required. The `preCalc` array, consisting of `psImage`s is provided for this purpose.

The `pmSubtractionKernels` are generated by the following functions:

```
pmSubtractionKernels *pmSubtractionKernelsAllocPOIS(int size, int spatialOrder);
pmSubtractionKernels *pmSubtractionKernelsAllocISIS(const psVector *sigmas, const psVector *orders,
                                                    int size, int spatialOrder);
```

`pmSubtractionKernelsAllocPOIS` shall generate the `pmSubtractionKernels` suitable for the POIS kernel basis set. This involves setting the `u`, `v`, `xOrder` and `yOrder` to the appropriate values. `size` is the half-size of the kernel, and `spatialOrder` is the maximum spatial order (the spatial variation is $x^i y^j$ with $i + j <$ `spatialOrder`). The `subIndex` is set to the kernel which has `u = 0`, `v = 0`, `xOrder = 0` and `yOrder = 0`. There should be `(2 * size + 1) * (2 * size + 1) * (spatialOrder + 1) * (spatialOrder + 2) / 2` kernels.

`pmSubtractionKernelsAllocISIS` shall generate the `pmSubtractionKernels` suitable for the ISIS kernel basis set. This involves setting the `sigma`, `u`, `v`, `xOrder` and `yOrder` to the appropriate values, as well as generating the `preCalc` images. Note that the `sigma` vector contained within the `pmSubtractionKernels` is not the same as the input `sigmas` vector, but contains repeated entries. `size` is the half-size of the kernel, which specifies the size of the `preCalc` images. The `spatialOrder` is the maximum spatial order (the spatial variation is $x^i y^j$ with $i + j <$ `spatialOrder`). The `subIndex` is set to the kernel which has `u = 0`, `v = 0`, `xOrder = 0` and `yOrder = 0`, for the first of the Gaussian widths in the `sigmas` vector.

10.2 Stamps

Sub-regions on an image which are used to derive the best-fit convolution kernel are referred to as "stamps".

```
typedef struct {
    int x, y;                           // Position
    psImage *matrix;                    // Associated matrix
    psVector *vector;                   // Associated vector
    pmStampStatus status;               // Status of stamp
} pmStamp;
```

A stamp is the region around a central pixel, `x,y`. The `matrix` and `vector` are generated in the process of solving for the best-fit convolution kernel; each of these will likely be of type `psF64` in order to maintain the best possible precision (we will be summing squares). In order to allow us to throw out stamps without having to laboriously recompute the total least-squares matrix and vector, we use a separate matrix and vector for each stamp.

To allow iteration on the choice of stamps, a stamp contains a `status`, an enumerated type:

```
typedef enum {
    PM_STAMP_USED,                      // Use this stamp
    PM_STAMP_REJECTED,                  // This stamp has been rejected
    PM_STAMP_RECALC,                    // Having been reset, this stamp needs to be recalculated
    PM_STAMP_NONE                       // No stamp in this region
} pmStampStatus;

psArray *pmSubtractionFindStamps(psArray *stamps, // Output stamps, or NULL
                                 const psImage *image, // Image for which to find stamps
                                 const psImage *mask, // Mask
                                 unsigned int maskVal, // Value for mask
                                 float threshold, // Threshold for stamps in the image
                                 int xNum, int yNum, // Number of stamps in x and y
                                 int border // Border around image to ignore (should be size of kernel)
                                 );
```

`pmSubtractionFindStamps` returns an array of stamps on the `image` suitable for use in calculating the best-fit convolution kernel. Except for a `border` all the way around, the `image` is broken into `xNum` $\times$ `yNum` rectangles; there will be a stamp within each rectangle. If `stamps` is non-`NULL`, then the function shall only attempt to identify a new stamp in a particular rectangle if the corresponding stamp `status` is `PM_STAMP_REJECTED`.

A stamp shall be recognized as the pixel with the greatest value that does not have the corresponding pixel in the `mask` matching `maskVal`. If the value of the this pixel does not exceed `threshold`, then the stamp `status` shall be marked as `PM_STAMP_NONE`, which means that the stamp will be ignored in future iterations. If a legitimate stamp is found within the region, then its status shall be changed to `PM_STAMP_RECALC`.

10.3 Solving for the kernel

Calculating the best-fit convolution kernel requires solving a matrix equation, the elements of which are obtained by applying the kernel basis functions to the stamps. The final matrix and vector are the sum of the matrices and vectors obtained for each of the individual stamps.

```
bool pmSubtractionCalculateEquation(psArray *stamps, // The stamps for which to calculate the equation
                                    const psImage *reference, // Reference image
                                    const psImage *input, // Input image
                                    const psSubtractionKernels *kernels, // The kernel basis functions
                                    int footprint // Half-size of region over which to calculate equation
                                    );
```

`pmSubtractionCalculateEquation` shall calculate the `matrix` and `vector` for each of the `stamps` which have `status` set to `PM_STAMP_RECALC`. The calculation is made over a region with a half size of `footprint` on the `reference` and `input` images, using each of the `kernels`. In the event that any of the input pointers are `NULL`, the function shall generate an error and return `false`; otherwise, the function shall return `true`.

The vector is:

$$v_i = \sum_{x,y} I(x,y)[R(x,y) \otimes B_i(u,v)]/\sigma(x,y)^2 \tag{7}$$

and the matrix is:

$$M_{ij} = \sum_{x,y} [R(x,y) \otimes B_i(u,v)] \; [R(x,y) \otimes B_j(u,v)] \, /\sigma(x,y)^2 \tag{8}$$

where $I(x,y)$ is the input image, $R(x,y)$ is the reference image, $B_i(u,v)$ is the i-th kernel basis function, $\otimes$ denotes convolution, $\sigma(x,y) = R(x,y)^{1/2}$ is an estimate of the error, and the sum over x, y indicates summing over the stamp regions.

In addition to the each of the `kernels`, an additional parameter for which we must solve is the difference in the background level between the `reference` and `input` images. The appropriate term shall be added to the `matrix` and `vector`.

In order to maintain flux conservation when the kernel is spatially variable, for each of the kernel basis functions apart from the first, the kernel actually employed shall be the first kernel function subtracted from the original kernel function.

Having calculated the matrix equation for a stamp, its `status` is set to `PM_STAMP_USED`.

Since this step is one of the major rate-limiting factors in image subtraction, care should be taken with optimization.

```
psVector *pmSubtractionSolveEquation(psVector *solution,        // Solution vector, or NULL
                                     const psArray *stamps // Array of stamps
                                     );
```

`pmSubtractionSolveEquation` shall solve the matrix equation provided by each of the `stamps`, returning the `solution` vector. This involves summing the `matrix` and `vector` of each of the stamps which have `status` set to `PM_STAMP_USED`, and multiplying the inverse of the matrix by the `vector`. If the `solution` is `NULL`, then the function shall allocate and return a new vector; otherwise, the `solution` vector shall be modified in-place. If `stamps` is `NULL`, then the function shall generate an error and return `NULL`. The type of the `solution` vector should be `psF64`, since the matrix equation involves summing squares.

10.4 Rejection of stamps

```
bool pmSubtractionRejectStamps(psArray *stamps, // Array of stamps to check for rejection
                               psImage *mask, // Mask image
                               unsigned int badStampMaskVal, // Value to use in mask for bad stamp
                               int footprint, // Region to mask if stamp is bad
                               float sigmaRej, // Number of RMS deviations above zero at which to reject
                               const psImage *refImage, // Reference image
                               const psImage *inImage, // Input image
                               const psVector *solution, // Solution vector
                               const pmSubtractionKernels *kernels // Array of kernel parameters
                               );
```

`pmSubtractionRejectStamps` shall apply the `solution` to the `stamps`, rejecting stamps for which the mean square residuals exceed `sigmaRej` RMS deviations from zero. `stamps` which are rejected have their `status` set to `PM_STAMP_REJECTED`, and have pixels within `footprint` of the corresponding position in the `mask` set to `badStampMaskVal` so they will not be used again.

The deviations are calculated through extracting the stamps from the `refImage` and `inImage`, convolving the reference stamp by the best-fit kernel (derived from the `solutions` vector and the `kernels`), subtracting and then dividing by the stamp from the input image, and then squaring to obtain the mean square residual.

10.5 Visualization of kernel

Having solved for the best-fit kernel, it is often useful to visualize it.

```
psImage *pmSubtractionKernelImage(psImage *out, const psVector *solution,
                                  const pmSubtractionKernels *kernels, float x, float y);
```

`pmSubtractionKernelImage` shall create an image of the kernel from the `solution` vector and the `kernels`. The relative position (between -1 and +1) on the image at which to evaluate the kernel (important if the kernel is spatially variable) is specified by `x` and `y`. If `out` is `NULL`, then the function shall allocate a new image of sufficient size (matching the `precalc` images), and return the result; otherwise, `out` shall be modified in-place.

10.6 Example

Here is an example of what the image subtraction routine looks like, demonstrating how the various pieces fit together. The inputs are:

- `psImage *reference`: Reference image
- `psImage *refMask`: Mask for reference image
- `psImage *input`: Input image
- `psImage *inMask`: Mask for input image
- `unsigned int maskVal`: Value to be masked
- `pmSubtractionKernelType kernelType`: Type of kernel to use
- `int kernelHalfSize`: Half the kernel size (full size is `2*kernelHalfSize + 1`)
- `psVector *sigmas`: Widths for the ISIS Gaussians
- `psVector *polyOrders`: Polynomial orders for ISIS Gaussians
- `int spatialOrder`: Maximum spatial order for spatially variable kernel
- `float stampThreshold`: Threshold for finding stamps
- `int nStampsX, nStampsY`: Number of stamps in x and y

- `int stampSize`: Half size of stamp footprint
- `int numIter`: Number of iterations on the stamps
- `float sigmaRej`: Rejection threshold for stamps

The output is the subtracted image and the corresponding mask.

```
// Mask around bad pixels in the reference image.  There are two cases to worry about:
// 1. Bad pixels within the kernel, which will affect the subtracted image
// 2. Bad pixels within the stamp, which affects the calculation of the kernel
psImage *subMask = psImageGrowMask(NULL, refMask, maskVal, kernelHalfSize, PS_MASK_NEAR_BAD);
(void)psImageGrowMask(subMask, refMask, maskVal, stampSize, PS_MASK_BAD_STAMP);
// Add in the mask for the input image.  Don't need to grow this, since it isn't convolved.
(void)psBinaryOp(subMask, subMask, "|", inMask);

// Generate kernel basis functions
psArray *kernels = NULL;            // Array of kernel basis functions
switch (kernelType) {
  case PM_SUBTRACTION_KERNEL_POIS:
    // Create the kernel basis functions
    kernels = pmSubtractionKernelsGeneratePOIS(kernelHalfSize, spatialOrder);
    break;
  case PM_SUBTRACTION_KERNEL_ISIS:
    kernels = pmSubtractionKernelsGenerateISIS(sigmas, polyOrders, kernelHalfSize, spatialOrder);
    break;
  default:
    barf();
}

psArray *stamps = NULL;             // Array of stamps
psVector *kernelCoeffs = NULL;      // Coefficients for the kernels
bool rejected = true;               // Did we reject a stamp in the last iteration?

// Iterate for a solution
for (int iter = 0; iter < numIter && rejected; iter++) {

    // Find stamps
    stamps = pmSubtractionFindStamps(stamps, reference, subMask, maskVal | PS_MASK_BAD_STAMP,
                                     stampThreshold, nStampsX, nStampsY, stampSize, kernelHalfSize);

    // Generate and solve matrix equations
    (void)pmSubtractionCalculateEquation(stamps, reference, input, kernels, stampSize);
    kernelCoeffs = pmSubtractionSolveEquation(kernelCoeffs, stamps);

    // Reject bad stamps
    rejected = pmSubtractionRejectStamps(stamps, subMask, PS_MASK_BAD_STAMP, stampSize, sigmaRej,
                                         reference, input, kernelCoeffs, kernels);
}

// Convolve the reference image
psImage *referenceConvolved = pmSubtractionConvolveImage(NULL, reference, subMask, kernelCoeffs, kernels);
// Subtract
psImage *subtracted = (psImage*)psBinaryOp(NULL, input, "-", referenceConvolved);

// What does the kernel look like?
psImage *kernelImage = pmSubtractionKernelImage(NULL, kernelCoeffs, kernels, 0.0, 0.0);
// Check/save kernel image, print statistics....

psFree(referenceConvolved);
psFree(stamps);
psFree(kernels);
psFree(kernelCoeffs);
```

A Basic Object Models

We specify a variety of basic object models which are required. Details of the model functional forms, parameters, and the derivatives are specified in the ADD.

A.0.1 Real 2D Gaussian

```
float pmMinLM_Gauss2D(psVector *deriv, psVector *params, psVector *x);
```

This function is a two-dimensional Gaussian with an elliptical cross-section and a constant local background.

The initial guess for the Gaussian parameters may be taken from the moments, peak value, and local sky.

A.0.2 Pseudo-Gaussian

```
float pmMinLM_PseudoGauss2D(psVector *deriv, psVector *params, psVector *x);
```

This function is a polynomial approximation of a 2D Gaussian otherwise very similar to the real Gaussian. It is used in place of a real Gaussian for speed.

The initial guess for the Gaussian parameters may be taken from the moments, peak value, and local sky.

A.0.3 Waussian

```
float pmMinLM_Wauss2D(psVector *deriv, psVector *params, psVector *x);
```

The Waussian is a modified polynomial approximation of a 2D Gaussian, with non-linear polynomial terms having variable coefficients, rather than the Taylor series values of 1/2 and 1/6.

A.0.4 Twisted Gaussian

```
float pmMinLM_TwistGauss2D(psVector *deriv, psVector *params, psVector *x);
```

This function describes an object with power-law wings and a flattened core, where the core has a different contour from the wings.

The initial guess for the Gaussian parameters may be taken from the moments, peak value, and local sky.

future galaxy models to be implemented (TBD)

A.0.5 Sersic Galaxy Model

```
float pmMinLM_Sersic(psVector *deriv, psVector *params, psVector *x);
```

A.0.6 Sersic with Core Galaxy Model

```
float pmMinLM_SersicCore(psVector *deriv, psVector *params, psVector *x);
```

A.0.7 Pseudo Sersic Galaxy Model

```
float pmMinLM_PseudoSersic(psVector *deriv, psVector *params, psVector *x);
```

B Example Camera Configuration Files

Some of these don't exactly match the specifications of this document yet, because they have been changed from the prototype, but it is hoped that they will be useful. Questions are welcome. (TBD)

B.1 MegaCam Raw

```
# The raw MegaCam data comes off the telescope with each of the chips stored in extensions of a MEF file.

# How to identify this type
RULE    METADATA
        TELESCOP        STR     CFHT 3.6m
        DETECTOR        STR     MegaCam
        EXTEND          BOOL    T
        NEXTEND         S32     72
END

# How to read this data
PHU             STR     FPA     # The FITS file represents an entire FPA
EXTENSIONS      STR     CELL    # The extensions represent cells

# What's in the FITS file?
CONTENTS        METADATA
        # Extension name, chip name:type
        amp00   STR     ccd00:left
        amp01   STR     ccd00:right
        amp02   STR     ccd01:left
        amp03   STR     ccd01:right
        amp04   STR     ccd02:left
        amp05   STR     ccd02:right
        amp06   STR     ccd03:left
        amp07   STR     ccd03:right
        amp08   STR     ccd04:left
        amp09   STR     ccd04:right
        amp10   STR     ccd05:left
        amp11   STR     ccd05:right
        amp12   STR     ccd06:left
        amp13   STR     ccd06:right
        amp14   STR     ccd07:left
        amp15   STR     ccd07:right
        amp16   STR     ccd08:left
        amp17   STR     ccd08:right
        amp18   STR     ccd09:left
        amp19   STR     ccd09:right
        amp20   STR     ccd10:left
        amp21   STR     ccd10:right
        amp22   STR     ccd11:left
        amp23   STR     ccd11:right
        amp24   STR     ccd12:left
        amp25   STR     ccd12:right
```

```
        amp26   STR     ccd13:left
        amp27   STR     ccd13:right
        amp28   STR     ccd14:left
        amp29   STR     ccd14:right
        amp30   STR     ccd15:left
        amp31   STR     ccd15:right
        amp32   STR     ccd16:left
        amp33   STR     ccd16:right
        amp34   STR     ccd17:left
        amp35   STR     ccd17:right
        amp36   STR     ccd18:left
        amp37   STR     ccd18:right
        amp38   STR     ccd19:left
        amp39   STR     ccd19:right
        amp40   STR     ccd20:left
        amp41   STR     ccd20:right
        amp42   STR     ccd21:left
        amp43   STR     ccd21:right
        amp44   STR     ccd22:left
        amp45   STR     ccd22:right
        amp46   STR     ccd23:left
        amp47   STR     ccd23:right
        amp48   STR     ccd24:left
        amp49   STR     ccd24:right
        amp50   STR     ccd25:left
        amp51   STR     ccd25:right
        amp52   STR     ccd26:left
        amp53   STR     ccd26:right
        amp54   STR     ccd27:left
        amp55   STR     ccd27:right
        amp56   STR     ccd28:left
        amp57   STR     ccd28:right
        amp58   STR     ccd29:left
        amp59   STR     ccd29:right
        amp60   STR     ccd30:left
        amp61   STR     ccd30:right
        amp62   STR     ccd31:left
        amp63   STR     ccd31:right
        amp64   STR     ccd32:left
        amp65   STR     ccd32:right
        amp66   STR     ccd33:left
        amp67   STR     ccd33:right
        amp68   STR     ccd34:left
        amp69   STR     ccd34:right
        amp70   STR     ccd35:left
        amp71   STR     ccd35:right
END

# Specify the cell data
CELLS   METADATA
        left    METADATA        # Left amplifier
                CELL.BIASSEC    STR     HEADER:BIASSEC
                CELL.TRIMSEC    STR     HEADER:DATASEC
                CELL.XPARITY    S32     1       # We could have specified this as a DEFAULT, but this works
        END
        right   METADATA        # Right amplifier
                CELL.BIASSEC    STR     HEADER:BIASSEC
                CELL.TRIMSEC    STR     HEADER:DATASEC
                CELL.XPARITY    S32     -1      # This cell is read out in the opposite direction
        END
END

# How to translate PS concepts into FITS headers
TRANSLATION     METADATA
        FPA.NAME        STR     EXPNUM
        FPA.AIRMASS     STR     AIRMASS
        FPA.FILTER      STR     FILTER
        FPA.POSANGLE    STR     ROTANGLE
```

```
        FPA.RA          STR     RA
        FPA.DEC         STR     DEC
        FPA.RADECSYS    STR     RADECSYS
        FPA.MJD         STR     MJD-OBS
        CELL.EXPOSURE   STR     EXPTIME
        CELL.DARKTIME   STR     DARKTIME
        CELL.XBIN       STR     CCDBIN1
        CELL.YBIN       STR     CCDBIN2
        CELL.GAIN       STR     GAIN
        CELL.READNOISE  STR     RDNOISE
        CELL.SATURATION STR     SATURATE
END

# Default PS concepts that may be specified by value
DEFAULTS        METADATA
        CELL.BAD                S32     0
        CELL.YPARITY_DEPEND     STR     CHIP.NAME
        CELL.YPARITY    METADATA
                ccd00   S32     -1
                ccd01   S32     -1
                ccd02   S32     -1
                ccd03   S32     -1
                ccd04   S32     -1
                ccd05   S32     -1
                ccd06   S32     -1
                ccd07   S32     -1
                ccd08   S32     -1
                ccd09   S32     -1
                ccd10   S32     -1
                ccd11   S32     -1
                ccd12   S32     -1
                ccd13   S32     -1
                ccd14   S32     -1
                ccd15   S32     -1
                ccd16   S32     -1
                ccd17   S32     -1
                ccd18   S32     1
                ccd19   S32     1
                ccd20   S32     1
                ccd21   S32     1
                ccd22   S32     1
                ccd23   S32     1
                ccd24   S32     1
                ccd25   S32     1
                ccd26   S32     1
                ccd27   S32     1
                ccd28   S32     1
                ccd29   S32     1
                ccd30   S32     1
                ccd31   S32     1
                ccd32   S32     1
                ccd33   S32     1
                ccd34   S32     1
                ccd35   S32     1
        END
END

# How to translate PS concepts into database lookups
DATABASE        METADATA
        TYPE            dbEntry         TABLE           COLUMN          GIVENDBCOL      GIVENPS
#       CELL.GAIN       dbEntry         Camera          gain            chipId,cellId   CHIP.NAME,CELL.NAME
#       CELL.READNOISE  dbEntry         Camera          readNoise       chipId,cellId   CHIP.NAME,CELL.NAME

# A database entry refers to a particular column (COLUMN) in a
# particular table (TABLE), given certain PS concepts (GIVENPS) that
# match certain database columns (GIVENDBCOL).

END
```

B.2 MegaCam Splice

```
# The spliced MecaCam data is stored in single extensions for each chip

# How to recognise this type
RULE    METADATA
        TELESCOP        STR     CFHT 3.6m
        DETECTOR        STR     MegaCam
        EXTEND          BOOL    T
        NEXTEND         S32     36
END

# How to read this data
PHU             STR     FPA     # The FITS file represents an entire FPA
EXTENSIONS      STR     CHIP    # The extensions represent chips

# What's in the FITS file?
CONTENTS        METADATA
        # Extension name, components
        ccd00           STR     left right
        ccd01           STR     left right
        ccd02           STR     left right
        ccd03           STR     left right
        ccd04           STR     left right
        ccd05           STR     left right
        ccd06           STR     left right
        ccd07           STR     left right
        ccd08           STR     left right
        ccd09           STR     left right
        ccd10           STR     left right
        ccd11           STR     left right
        ccd12           STR     left right
        ccd13           STR     left right
        ccd14           STR     left right
        ccd15           STR     left right
        ccd16           STR     left right
        ccd17           STR     left right
        ccd18           STR     left right
        ccd19           STR     left right
        ccd20           STR     left right
        ccd21           STR     left right
        ccd22           STR     left right
        ccd23           STR     left right
        ccd24           STR     left right
        ccd25           STR     left right
        ccd26           STR     left right
        ccd27           STR     left right
        ccd28           STR     left right
        ccd29           STR     left right
        ccd30           STR     left right
        ccd31           STR     left right
        ccd32           STR     left right
        ccd33           STR     left right
        ccd34           STR     left right
        ccd35           STR     left right
END

# Specify the cells
CELLS           METADATA
        left            METADATA
                CELL.BIASSEC    STR     HEADER:BSECA
                CELL.TRIMSEC    STR     HEADER:TSECA
        END

        right           METADATA
                CELL.BIASSEC    STR     HEADER:BSECB
                CELL.TRIMSEC    STR     HEADER:TSECB
```

```
        END
END

# How to translate PS concepts into FITS headers
TRANSLATION     METADATA
        FPA.NAME        STR     EXPNUM
        FPA.AIRMASS     STR     AIRMASS
        FPA.FILTER      STR     FILTER
        FPA.POSANGLE    STR     ROTANGLE
        FPA.RA          STR     RA
        FPA.DEC         STR     DEC
        FPA.RADECSYS    STR     RADECSYS
        FPA.MJD         STR     MJD-OBS
        CELL.EXPOSURE   STR     EXPTIME
        CELL.DARKTIME   STR     DARKTIME
        CELL.XBIN       STR     CCDBIN1
        CELL.YBIN       STR     CCDBIN2
        CELL.GAIN       STR     GAIN
        CELL.READNOISE  STR     RDNOISE
        CELL.SATURATION STR     SATURATE
END

# Default PS concepts that may be specified by value
DEFAULTS        METADATA
        CELL.BAD                S32     0
        CELL.XPARITY            S32     1
        CELL.YPARITY            S32     1
END


# How to translate PS concepts into database lookups
DATABASE        METADATA
        TYPE            dbEntry         TABLE           COLUMN          GIVENDBCOL      GIVENPS
#       CELL.GAIN       dbEntry         Camera          gain            chipId,cellId   CHIP.NAME,CELL.NAME
#       CELL.READNOISE  dbEntry         Camera          readNoise       chipId,cellId   CHIP.NAME,CELL.NAME

# A database entry refers to a particular column (COLUMN) in a
# particular table (TABLE), given certain PS concepts (GIVENPS) that
# match certain database columns (GIVENDBCOL).

END
```

B.3 LRIS Blue

```
# The Low Resolution Imager and Spectrograph (LRIS) blue side

# We have no choice but to hard-code the various regions, because Keck
# only stores them as:
# WINDOW  = '1,0,0,2048,4096'
# PREPIX  =                   51
# POSTPIX =                   80
# BINNING = '1,1     '
# AMPPSIZE= '[1:1024,1:4096]'

# I don't know how we would get the IPP to react to changes in the
# windowing on the fly --- we have no mechanism for setting the region
# sizes on the basis of the above keywords.  Therefore, we hard-code
# the regions and assert on our assumptions in the RULE.


# How to identify this type
RULE    METADATA
        TELESCOP        STR     Keck I
        INSTRUME        STR     LRISBLUE
        AMPLIST         STR     1,4,0,0
```

```
        WINDOW          STR     1,0,0,2048,4096
        PREPIX          S32     51
        POSTPIX         S32     80
        BINNING         STR     1,1
        AMPPSIZE        STR     [1:1024,1:4096]
        NAXIS1          S32     4620
        NAXIS2          S32     4096
END

# How to read this data
PHU             STR     FPA     # The FITS file represents an entire FPA
EXTENSIONS      STR     NONE    # There are no extensions

# What's in the FITS file?
CONTENTS        METADATA
        LeftChip        STR     amp1 amp2
        RightChip       STR     amp3 amp4
END

# Specify the cell data
CELLS   METADATA
        amp1            METADATA
                CELL.BIASSEC    STR     VALUE:[1:51,1:4096];[4301:4380,1:4096]
                CELL.TRIMSEC    STR     VALUE:[205:1228,1:4096]
                CELL.GAIN       STR     VALUE:1.2
                CELL.READNOISE  STR     VALUE:5.6
        END

        amp2    METADATA
                CELL.BIASSEC    STR     VALUE:[52:102,1:4096];[4381:4460,1:4096]
                CELL.TRIMSEC    STR     VALUE:[1229:2252,1:4096]
                CELL.GAIN       STR     VALUE:1.3
                CELL.READNOISE  STR     VALUE:6.7
        END

        amp3            METADATA
                CELL.BIASSEC    STR     VALUE:[103:153,1:4096];[4461:4540,1:4096]
                CELL.TRIMSEC    STR     VALUE:[2253:3276,1:4096]
                CELL.GAIN       STR     VALUE:1.4
                CELL.READNOISE  STR     VALUE:7.8
        END

        amp4    METADATA
                CELL.BIASSEC    STR     VALUE:[154:204,1:4096];[4541:4620,1:4096]
                CELL.TRIMSEC    STR     VALUE:[3277:4300,1:4096]
                CELL.GAIN       STR     VALUE:1.5
                CELL.READNOISE  STR     VALUE:8.9
        END
END

# How to translate PS concepts into FITS headers
TRANSLATION     METADATA
        FPA.AIRMASS     STR     AIRMASS
        FPA.FILTER      STR     BLUFILT
        FPA.POSANGLE    STR     ROTPOSN
        FPA.RA          STR     RA
        FPA.DEC         STR     DEC
        CELL.EXPOSURE   STR     EXPOSURE
        CELL.DARKTIME   STR     EXPOSURE        // No special darktime header; use exposure time
        CELL.DATE       STR     DATE            // NOTE: There are TWO keywords called "DATE" (creation, exp)!
        CELL.TIME       STR     UT
END

# Default PS concepts that may be specified by value
DEFAULTS        METADATA
        FPA.RADECSYS    STR     ICRS
END
```

B.4 LRIS Red

```
# The Low Resolution Imager and Spectrograph (LRIS) red side

# We have no choice but to hard-code the various regions, because Keck
# only stores them as:
# WINDOW  = '0,0,0,2048,2048'
# PREPIX  =                   20
# POSTPIX =                   80
# BINNING = '1,1     '
# AMPPSIZE= '[1:1024,1:4096]'

# I don't know how we would get the IPP to react to changes in the
# windowing on the fly --- we have no mechanism for setting the region
# sizes on the basis of the above keywords.  Therefore, we hard-code
# the regions and assert on our assumptions in the RULE.


# How to identify this type
RULE    METADATA
        TELESCOP        STR     Keck I
        INSTRUME        STR     LRIS
        AMPLIST         STR     2,1,0,0
        WINDOW          STR     0,0,0,2048,2048
        PREPIX          S32     20
        POSTPIX         S32     80
        BINNING         STR     1, 1
        CCDPSIZE        STR     [1:2048,1:2048]
        NAXIS1          S32     2248
        NAXIS2          S32     2048
        IMTYPE          STR     TWOAMPTOP
END

# How to read this data
PHU             STR     CHIP    # The FITS file represents a single chip
EXTENSIONS      STR     NONE    # There are no extensions

# What's in the FITS file?
CONTENTS        STR     LeftSide RightSide

# Specify the cell data
CELLS   METADATA
        LeftSide        METADATA
                CELL.BIASSEC    STR     VALUE:[1:20,1:2048];[2089:2168,1:2048]
                CELL.TRIMSEC    STR     VALUE:[41:1064,1:2048]
                CELL.GAIN       STR     VALUE:1.2
                CELL.READNOISE  STR     VALUE:5.6
        END

        RightSide       METADATA
                CELL.BIASSEC    STR     VALUE:[21:40,1:2048];[2169:2248,1:2048]
                CELL.TRIMSEC    STR     VALUE:[1065:2088,1:2048]
                CELL.GAIN       STR     VALUE:1.3
                CELL.READNOISE  STR     VALUE:6.5
        END
END

# How to translate PS concepts into FITS headers
TRANSLATION     METADATA
        FPA.AIRMASS     STR     AIRMASS
        FPA.FILTER      STR     FILTER
        FPA.POSANGLE    STR     POSANG
        FPA.RA          STR     OBJ-RA
        FPA.DEC         STR     OBJ-DEC
        CELL.EXPOSURE   STR     EXPTIME
        CELL.DARKTIME   STR     DARKTIME
        CELL.DATE       STR     DATE-OBS
```

```
        CELL.TIME       STR     TIME-OBS
END

# Default PS concepts that may be specified by value
DEFAULTS        METADATA
        FPA.RADECSYS    STR     ICRS
END
```

B.5 GPC OTA

```
# The raw GPC data comes off the telescope with each of the chips stored in separate files

# How to identify this type
RULE    METADATA
#       TELESCOP        STR     PS1
#       DETECTOR        STR     GPC1
        EXTEND          BOOL    T
        NEXTEND         S32     64
        NAMPS           S32     64
END

# How to read this data
PHU             STR     CHIP    # The FITS file represents a single chip
EXTENSIONS      STR     CELL    # The extensions represent cells

# What's in the FITS file?
CONTENTS        METADATA
        # Extension name, type
        xy00    STR     pitch10u
        xy01    STR     pitch10u
        xy02    STR     pitch10u
        xy03    STR     pitch10u
        xy04    STR     pitch10u
        xy05    STR     pitch10u
        xy06    STR     pitch10u
        xy07    STR     pitch10u
        xy10    STR     pitch10u
        xy11    STR     pitch10u
        xy12    STR     pitch10u
        xy13    STR     pitch10u
        xy14    STR     pitch10u
        xy15    STR     pitch10u
        xy16    STR     pitch10u
        xy17    STR     pitch10u
        xy20    STR     pitch10u
        xy21    STR     pitch10u
        xy22    STR     pitch10u
        xy23    STR     pitch10u
        xy24    STR     pitch10u
        xy25    STR     pitch10u
        xy26    STR     pitch10u
        xy27    STR     pitch10u
        xy30    STR     pitch10u
        xy31    STR     pitch10u
        xy32    STR     pitch10u
        xy33    STR     pitch10u
        xy34    STR     pitch10u
        xy35    STR     pitch10u
        xy36    STR     pitch10u
        xy37    STR     pitch10u
        xy40    STR     pitch10u
        xy41    STR     pitch10u
        xy42    STR     pitch10u
        xy43    STR     pitch10u
        xy44    STR     pitch10u
```

```
        xy45    STR     pitch10u
        xy46    STR     pitch10u
        xy47    STR     pitch10u
        xy50    STR     pitch10u
        xy51    STR     pitch10u
        xy52    STR     pitch10u
        xy53    STR     pitch10u
        xy54    STR     pitch10u
        xy55    STR     pitch10u
        xy56    STR     pitch10u
        xy57    STR     pitch10u
        xy60    STR     pitch10u
        xy61    STR     pitch10u
        xy62    STR     pitch10u
        xy63    STR     pitch10u
        xy64    STR     pitch10u
        xy65    STR     pitch10u
        xy66    STR     pitch10u
        xy67    STR     pitch10u
        xy70    STR     pitch10u
        xy71    STR     pitch10u
        xy72    STR     pitch10u
        xy73    STR     pitch10u
        xy74    STR     pitch10u
        xy75    STR     pitch10u
        xy76    STR     pitch10u
        xy77    STR     pitch10u
END

# Specify the cell data
CELLS   METADATA
        pitch10u        METADATA
                CELL.BIASSEC    STR     VALUE:[575:606,1:594]
                CELL.TRIMSEC    STR     VALUE:[1:574,1:594]
        #       CELL.BIASSEC    STR     HEADER:BIASSEC
        #       CELL.TRIMSEC    STR     HEADER:DATASEC
        END

        # This is just in here for fun
        pitch12u        METADATA
                CELL.BIASSEC    STR     VALUE:[1:10,1:512];[523:574,1:512]
                CELL.TRIMSEC    STR     VALUE:[11:522,1:512]
        #       CELL.BIASSEC    STR     HEADER:BIASSEC
        #       CELL.TRIMSEC    STR     HEADER:TRIMSEC
        END
END


# How to translate PS concepts into FITS headers
TRANSLATION     METADATA
        CELL.BIN        STR     CCDSUM
        CELL.SATURATION STR     SATURATE
END

# Default PS concepts that may be specified by value
DEFAULTS        METADATA
        FPA.AIRMASS     F32     0.0
        FPA.FILTER      STR     NONE
        FPA.POSANGLE    F32     0.0
        FPA.RA          STR     0:0:0
        FPA.DEC         STR     0:0:0
        FPA.RADECSYS    STR     ICRS
        FPA.NAME        S32     0
        FPA.MJD         F32     12345.6789
        CELL.EXPOSURE   F32     0.0
        CELL.DARKTIME   F32     0.0
        CELL.GAIN       F32     1.0
        CELL.READNOISE  F32     0.0
```

```
        CELL.BAD        S32     0
        CELL.BIN        S32     1
        CELL.XPARITY    S32     1
        CELL.YPARITY    S32     1
END

# How to translate PS concepts into database lookups
DATABASE        METADATA
        TYPE            dbEntry         TABLE           COLUMN          GIVENDBCOL      GIVENPS
        CELL.GAIN       dbEntry         Camera          gain            chipId,cellId   CHIP,CELL
        CELL.READNOISE  dbEntry         Camera          readNoise       chipId,cellId   CHIP,CELL

# A database entry refers to a particular column (COLUMN) in a
# particular table (TABLE), given certain PS concepts (GIVENPS) that
# match certain database columns (GIVENDBCOL).

END
```

C Revision Change Log

C.1 Changes from version 00 (16 August 2004) to version 01 (12 October 2004)

- clarified the image offsets for pmFlatField ()
- changed return value to bool.
- added `pmCameraFromHeader`
- added `pmCameraValidateHeaders`
- added `pmFPAfromHeader`
- Added `pmReadoutCombine`

C.2 Changes from version 01 (12 October 2004) to version 02 (30 November 2004)

- `nBin` in `pmSubtractBias` is also interpreted as the number of spline pieces if spline fitting is specified.
- Refined `pmReadoutCombine` specification in response to bug 227.
- added details to the functions `pmCameraFromHeader`, `pmCameraValidateHeaders`, and `pmFPAfromHeader`.
- reorganiztion: placed configuration section up front, camera layout next, etc
- added details about configuration system
- added utility modules `pmConfigLoadSite`, `pmConfigLoadCamera`, `pmConfigLoadRecipe`
- added utility modules `pmConfigLookupSTR`, `pmConfigLookupS32`, `pmConfigLookupF64`, `pmConfigLookupRegion`,
- added discussion about Coordinate transforms
- added discussion about `pmReadoutLoad`
- added module `pmSubtractSky`

C.3 Changes from version 02 (30 November 2004) to version 03 (21 January 2005)

- Fixed up specification of `fitSpec` for `pmSubtractSky`.
- Added a `mask` to `pmSubtractSky`, and specified that binned pixels which are clipped may be interpolated over, or simply ignored.
- Added further explanation for `pmReadoutCombine`.
- Added Object Detection section
- Added PSPhot pseudo-C example

C.4 Changes from version 03 (21 January 2005) to version 04 (14 February 2005)

- changed entries of the form psXXX to pmXXX (object section)
- added enum psSourceType
- Specified appropriate image types for the phase 2 modules (bug 258).
- clarified pmSourceRoughClass
- clarified pmSourceAddModel, pmSourceSubModel

C.5 Changes from version 04 (14 February 2005) to version 05 (21 March 2005)

- Added section on image combination
- Added section on image subtraction

C.6 Changes from version 05 (21 March 2005) to version 06 (27 April 2005)

- changed `pmFindImagePeaks` to return an array, not a list
- replaced `pmCullPeaks` with `pmPeaksSubset` which returns a new array
- changed `pmModel` to use vectors for params and dparms.
- added nDOF and nIter to pmModel
- changed models to return psF64, not psF32, to match psMinimizeLMChi2Func

C.7 Changes from version 06 (27 April 2005) to version 07 (15 July 2005)

- Changed `psRegion *region` to `psRegion region` in prototypes (passed by value instead of by reference).
- `pmSourceMoments` does not require image parameter.
- Added `masks` to `pmRejectPixels`.

- Added `size` and `spatialOrder` to `pmSubtractionKernels`.
- added Image Hierarchy section from psLibSDRS
- added photometry section from psLibSDRS
- added object function abstractions to Objects
- modified pmSource to include modelPSF and modelFLT
- Major changes to configuration. Modifyied `pmConfig` functions, and `pmCameraFromHeader`, and added various other functions.
- Modifying `psReadout, psCell, psChip, psFPA` structures.
- Removing `psObservatory, psExposure, psGrommit` which were centered on slalib.
- Added `pmFPAConstruct, pmFPARead, pmFPAMorph, pmFPAWrite`.

C.8 Changes from version 07 (15 July 2005) to version 08 (13 Sept 2005)

- Added `bias` sections to `pmReadout`.
- Added `CELL.READDIR` to specify read direction; `pmCellGetReaddir`.
- `pmFPAMorph` specified, ready for coding.
- `pmConfigRead` shall call `psTimeInitialize`, `psLogSetLevel`, `psLogSetFormat`, `psTraceSetLevel`.
- Added `pmConfigDB`.
- Changed `pmNonLinearityLookup` to read a `psLookupTable`.
- Changed input types for `pmMaskBadPixels`.
- Added `pmFPAWriteMask`.
- `pmMaskBadPixels` shall grow the mask for saturated by 1, in addition to the explicit grow.
- Added section on “Paper Trail” to Phase 2 functions.
- Adding log destination to `pmConfigRead`.
- Changing details of focal plane hierarchy.
- Concepts are evaluated at ingest by `pmFPARead`.
- Modified `pmSubtractBias`.
- Remove mask from `pmFlatField` (mask is contained in the readout).
- cleaned up / extended discussion of FITS and the FPA hierarchy.
- added `CONCEPT.DEFAULT` to `DEFAULTS` table.

- added `psArgumentVerbosity` requirement to `pmConfigRead`.
- substantial changes to the Objects section:
  - added `psphotMaskValues`?
  - added SN to `pmMoments`
  - added `pmPSFClump`
  - modified `pmSourceType` enum list
  - added radius to `pmModel`
  - added `pmModelGroup`
  - added several Model Group functions
  - added `pmPSF` and related functions
  - added `pmPSFtry` and related functions
  - added `pmSourceDefinePixels`
  - modified `pmSourceLocalSky` API
  - modified `pmSourceMoments` API
  - added `pmSourcePSFClump`
  - modified `pmSourceRoughClass` API
  - dropped `pmSourceSetPixelsCircle` (replaced with `pmSourceDefinePixels` and the image mask functions.
  - added `pmModelFitStatus`
  - added `pmSourcePhotometry`
  - added `pmSourceDophotType`
  - added `pmSourceSextractType`
  - moved discussion of the object models to an appendix

C.9 Changes from version 08 (13 Sept 2005) to version 09 (18 Oct 2005)

- fix enum syntax
- added Astrometry Fitting Support (matching / fitting routines)
- added `pmAstromRadiusMatch`
- added `pmAstromGridMatch`
- added `pmAstromApplyGridMatch`
- added `pmAstromMeasureGradients`
- added `pmAstromFitFPA`
- added `pmAstromFitChip`
- added `pmAstromFitDistortion`

- renamed section 7 to Detrend Creation, moved `pmReadoutCombine` to a subsection
- added `pmFringeStats` (Fringe Amplitude subsection)
- added `pmFlatNormalization` (Flat-field renorm section)
- added `pmDetrendLookup`

C.10 Changes from version 09 (18 Oct 2005) to version 10 (06 Dec 2005)

- `pmSubtractBias`: The overscans are to be derived by the function using `CELL.BIASSEC` (not from the `bias` parameter passed to the function, which is supposed to hold the full-frame bias image).
- `pmSubtractBias`: The full-frame (bias and dark) subtractions should only be performed on the region of the image specified by `CELL.TRIMSEC`.
- `pmNonLinearityPolynomial`, `pmNonLinearityLookup`, `pmFlatField`, `pmMaskBadPixels`, `pmSubtractSky` are to act only on the region of the readout image specified by `CELL.TRIMSEC`
- Added `p_pmHDU`.
- Changed `pmFPA`, `pmChip`, `pmCell->private` to `hdu`.
- changed `pmAstrometryReadWCS` to `pmAstromReadWCS`
- changed `pmAstrometryWriteWCS` to `pmAstromWriteWCS`
- changed `pmAstromGridMatch` to output `pmAstromStats`
- moved various `pmAstromGridMatch` output values from metadata to the output `pmAstromStats`
- changed `pmAstromGridMatchAngle` to `pmAsromGridAngle`
- changed `pmAstromGridAngle` to output `pmAstromStats`
- changed `pmAstromRotateObj` to accept center as `psPlane`

C.11 Changes from version 10 (06 Dec 2005) to version 11 (22 Jan 2006)

- modification of bias subtraction API
- updates to object/psphot APIs