# Libnetpbm Image Processing Manual

**Libnetpbm Image Processing Manual**

Updated: December 2003

Table Of Contents <?makeman .SH NAME ?> <?makeman libnetpbm_image   overview of netpbm image-processing functions ?> <?makeman .SH DESCRIPTION ?>

This reference manual covers functions in the **libnetpbm** library for processing images, using the Netpbm image formats and the **libnetpbm** in-memory image formats.

For historical reasons as well as to avoid clutter, it does not cover the largely obsolete PBM, PGM, PPM, and PNM classes of **libnetpbm** functions. For those, see PBM Function Manual, PGM Function Manual, PPM Function Manual, and PNM Function Manual. Note that you do not need those functions to process PBM, PGM, PPM, and PNM images. The functions in this manual are sufficient for that.

The PPM Drawing function are covered separately in PPM Drawing Function Manual.

For introductory and general information using **libnetpbm**, see Libnetpbm User's Guide.

**libnetpbm** also contains functions that are not specifically oriented toward processing image data. Read about those in the Libnetpbm Utility Manual.

To use these services, #include **pam.h**.

## Types

Here are some important types that you use with **libnetpbm**:

sampleA sample of a Netpbm image. See the format specifications -- as an example, the red intensity of a particular pixel of a PPM image is a sample. This is an integer type.

tuple A tuple from a PAM image or the PAM equivalent of a PNM image. See the PAM format specification -- as an example, a pixel of a PPM image would be a tuple. A tuple is an array of samples.

samplenSame as **sample**, except in normalized form. This is a floating point type with a value in the range 0..1. 0 corresponds to a PAM/PNM sample value of 0. 1 corresponds to a PAM/PNM sample value equal to the image's maxval.

tuplenThe same as **tuple**, except composed of normalized samples (**samplen**) intead of regular samples (**sample**).

The main argument to most of the PAM functions is the address of a **pam** structure, which is defined as follows:

**struct pam {**
**int** *size*
**int** *len*
**FILE ****file*
**int** *format*
**int** *plainformat*
**int** *height*
**int** *width*
**int** *depth*
**sample** *maxval*
**int** *bytes_per_sample*

**char** *tuple_type***[256]; }**

See The Libnetbm User's Guide for information on the **pam** structure.

### Macros

**PNM_MAXMAXVAL** is the maximum maxval that Netpbm images could historically have: 255. Many programs aren't capable of handling Netpbm images with a maxval larger than this. It's named this way for backward compatibility -- it had this name back when it was the maximum maxval.

**PNM_OVERALLMAXVAL** is the maximum maxval that Netpbm images can have today (65535).

**PBM_FORMAT**, **RPBM_FORMAT**, **PGM_FORMAT**, **RPGM_FORMAT**, **PPM_FORMAT**, **RPPM_FORMAT**, and **PAM_FORMAT** are the format codes of the various Netpbm formats. **RPBM_FORMAT** is the raw PBM format and **PBM_FORMAT** is the plain PBM format, and so on. See the *format* member of the **pam** structure.

**PAM_FORMAT_TYPE(***format***)** gives the type of a format, given the format code. The types of formats are PBM, PGM, PPM, and PAM and macros for the type codes are, respectively, PBM_TYPE, PGM_TYPE, PPM_TYPE, and PAM_TYPE. Note that there are more format codes then there are format types because there are different format codes for the plain and raw subformats of each format.

### Functions

These interfaces are declared in **pam.h**.

### Memory Management

### Synopsis

**tuple ** pnm_allocpamarray( struct pam ****pamP***);**

**tuple * pnm_allocpamrow( struct pam ****pamP***);**

**void pnm_freepamarray( tuple *****tuplearray***, struct pam ****pamP***);**

**void pnm_freepamrow( tuple ****tuplerow***);**

**tuple * allocpamtuple( struct pam ****pamP***);**

**void pnm_freepamtuple( tuple** *tuple* **);**

**tuplen * pnm_allocpamrown( struct pam ****pamP***);**

**void pnm_freepamrown( tuple ****tuplenrow***);**

### Description

**pnm_allocpamarray()** allocates space for an array of tuples. **pnm_freepamarray()** frees an array space allocated by **pnm_allocpamarray()** or **pnm_readpam()**.

**pnm_allocpamrow()** allocates space for a row of a PAM image, in basic form. **pnm_freepamrow()** frees it.

**pnm_allocpamrown()** is the same as **pnm_allocpamrow()** except that it allocates space for a PAM row in the normalized form. **pnm_freepamrown()** is similarly like **pnm_freepamrow**.

### Reading Netpbm Files

### Synopsis

**void pnm_readpaminit( FILE ****file***, struct pam ****pamP***, int** *size***);**

**void pnm_readpamrow( struct pam ****pamP***, tuple ****tuplerow***);**

**tuple ** pnm_readpam( FILE ****file***, struct pam ****pamP***,**
**int** *size***);**

**void pnm_readpamrown( struct pam ***_pamP_**, tuplen ***_tuplenrow_**);**

**Description**

**pnm_readpaminit()** reads the header of a Netpbm image.

See above for a general description of the _pamP_ argument.

**pnm_readpaminit()** returns the information from the header in the ***_pamP_ structure. It does not require any members of ***_pamP_ through **tuple_type** to be set at invocation, and sets all of those members. It expects all members after **tuple_type** to be meaningful.

_size_ is the size of the ***_pamP_ structure as understood by the program processing the image. **pnm_readpaminit()** does not attempt to use or set any members of the structure beyond that. The point of this argument is that the definition of the structure may change over time, with additional fields being added to the end. This argument allows **pnm_readpaminit** to distinguish between a new program that wants to exploit the additional features and an old program that cannot (or a new program that just doesn't want to deal with the added complexity). At a minimum, this size must contain the members up through **tuple_type**. You should use the **PAM_STRUCT_SIZE** macro to compute this argument. E.g. **PAM_STRUCT_SIZE(tuple_type)**.

The function expects to find the image file positioned to the start of the header and leaves it positioned to the start of the raster.

**pnm_readpamrow()** reads a row of the raster from a Netpbm image file. It expects all of the members of the ***pamP** structure to be set upon invocation and does not modify any of them. It expects to find the file positioned to the start of the row in question in the raster and leaves it positioned just after it. It returns the row as the array of tuples _tuplerow_, which must already have its column pointers set up so that it forms a C 2-dimensional array. The leftmost tuple is Element 0 of this array.

**pnm_readpam()** reads an entire image from a PAM or PNM image file and allocates the space in which to return the raster. It expects to find the file positioned to the first byte of the image and leaves it positioned just after the image.

The function does not require ***_pamP_ to have any of its members set and sets them all. _size_ is the storage size in bytes of the ***_pamP_ structure, normally **sizeof(struct pam)**.

The return value is a newly allocated array of the rows of the image, with the top row being Element 0 of the array. Each row is represented as **pnm_readpamrow()** would return.

The return value is also effectively a 3-dimensional C array of samples, with the dimensions corresponding to the height, width, and depth of the image, in that order.

**pnm_readpam()** combines the functions of **pnm_allocpamarray()**, **pnm_readpaminit()**, and iterations of **pnm_readpamrow()**. It may require more dynamic storage than you can afford.

**pnm_readpamrown()** is like **pnm_readpamrow()** except that it returns the row contents in normalized form (composed of normalized tuples (**tuplen**) instead of basic form (**tuple**).

**pnm_readpaminit()** and **pnm_readpam** abort the program with a message to Standard Error if the PAM or PNM image header is not syntactically valid, including if it contains a number too large to be processed using the system's normal data structures (to wit, a number that won't fit in a C 'int').

**Writing Netpbm Files**

**Synopsis**

**void pnm_writepaminit( struct pam ***_pamP_**);**

**void pnm_writepamrow( struct pam ***_pamP_**, const tuple ***_tuplerow_**);**

**void pnm_writepam( struct pam ***_pamP_**, const tuple * const ***_tuplearray_**);**

**void pnm_writepamrown( struct pam ***_pamP_**, const tuplen ***_tuplerown_**);**

**void pnm_formatpamrow( struct pam ***_pamP_**, const tuple ***_tuplerow_ **unsigned char * const** _outbuf_**,**

**unsigned int * const** *rowSizeP* **);**

### Description

**pnm_writepaminit()** writes the header of a PAM or PNM image and computes some of the fields of the pam structure.

See above for a description of the *pamP* argument.

The following members of the ***pamP** structure must be set upon invocation to tell the function how and what to write. **size**, **len**, **file**, **format**, **height**, **width**, **depth**, **maxval**, **tuple_type**.

**pnm_writepaminit()** sets the **plainformat** and **bytes_per_sample** members based on the information supplied.

**pnm_writepamrow()** writes a row of the raster into a PAM or PNM image file. It expects to find the file positioned where the row should start and leaves it positioned just after the row. The function requires all the elements of ***pamP** to be set upon invocation and doesn't modify them.

*tuplerow* is an array of tuples representing the row. The leftmost tuple is Element 0 of this array.

**pnm_writepam()** writes an entire PAM or PNM image to a PAM or PNM image file. It expects to find the file positioned to where the image should start and leaves it positioned just after the image.

The following members of the ***pamP** structure must be set upon invocation to tell the function how and what to write: **size**, **len**, **file**, **format**, **height**, **width**, **depth**, **maxval**, **tuple_type**.

**pnm_writepam()** sets the **plainformat** and **bytes_per_sample** members based on the information supplied.

*tuplearray* is an array of rows such that you would pass to **pnm_writepamrow()**, with the top row being Element 0 of the array.

**pnm_writepam()** combines the functions of **pnm_writepaminit()**, and iterations of **pnm_writepamrow()**. Its raster input may be more storage than you can afford.

**pnm_writepamrown()** is like **pnm_writepamrow()** except that it takes the row contents in normalized form (composed of normalized tuples (**tuplen**) instead of basic form (**tuple**).

**pnm_formatpamrow()** is like **pnm_writepamrow()**, except that instead of writing a row to a file, it places the same bytes that would go in the file in a buffer you supply. There isn't an equivalent function to construct an image header; i.e. there is no analog to **pnm_writepaminit()**. But the header format, particularly for PAM, is so simple that you can easily build it yourself with standard C library string functions.

**pnm_formatpamrow()** was new in Netpbm 10.25 (October 2004).

## Transforming Pixels

### Synopsis

**void pnm_YCbCrtuple(**
**tuple***tuple*, **double ****YP*, **double ****CrP*, **double ****CbP*);

**void pnm_YCbCr_to_rgbtuple(**
**const struct pam * const** *pamP*,
**tuple const** *tuple*,
**double const** *Y*, **double const** *Cb*, **double const** *Cr*,
**int * const** *overflowP* **);**

**extern double pnm_lumin_factor[3];**

**void pnm_normalizetuple(**
**struct pam * const** *pamP*, **tuple const** *tuple*, **tuplen const** *tuplen*);

**void pnm_unnormalizetuple(**
**struct pam * const** *pamP*, **tuplen const** *tuplen*, **tuple const** *tuple*);

**void pnm_normalizeRow(**

**struct pam *** **const** *pamP*, **const tuple *** **const** *tuplerow*, **pnm_transformMap * const** *transform*, **tuplen *** **const** *tuplenrow*);

**void pnm_unnormalizeRow(**
**struct pam *** **const** *pamP*, **const tuplen *** **const** *tuplenrow*, **pnm_transformMap * const** *transform*, **tuple *** **const** *tuplerow*);

**void pnm_gammarown(**
**struct pam * const** *pamP*, **tuplen *** **const** *row* );

**void pnm_ungammarown(**
**struct pam * const** *pamP*, **tuplen *** **const** *row* );

**void pnm_applyopacityrown(**
**struct pam * const** *pamP*, **tuplen *** **const** *tuplenrow* );

**void pnm_unapplyopacityrown(**
**struct pam * const** *pamP*, **tuplen *** **const** *tuplenrow* );

**pnm_transformMap * pnm_creategammatransform(**
**const struct pam * const** *pamP* );

**void pnm_freegammatransform(**
**const pnm_transformMap * const** *transform*, **const struct pam *** **const** *pamP* );

**pnm_transformMap * pnm_createungammatransform(**
**const struct pam * const** *pamP* );

**void pnm_freeungammatransform(**
**const pnm_transformMap * const** *transform*, **const struct pam *** **const** *pamP* );

### Description

**pnm_YCbCrtuple()** returns the Y/Cb/Cr luminance/chrominance representation of the color represented by the input tuple, assuming that the tuple is an RGB color representation (which is the case if it was read from a PPM image). The output components are based on the same scale (maxval) as the input tuple, but are floating point nonetheless to avoid losing information due to rounding. Divide them by the maxval to get normalized [0..1] values.

**pnm_YCbCr_to_rgbtuple()** does the reverse. *pamP* indicates the maxval for the returned *tuple*, and the *Y*, *Cb*, and *Cr* arguments are of the same scale.

It is possible for *Y*, *Cb*, and *Cr* to describe a color that cannot be represented in RGB form. In that case, **pnm_YCbCr_to_rgbtuple()** chooses a color as close as possible (by clipping each component to 0 and the maxval) and sets *overflowP true. It otherwise sets *overflowP false. **pnm_lumin_factor[]** is the factors (weights) one uses to compute the intensity of a color (according to some standard -- I don't know which). pnm_lumin_factor[0] is for the red component, [1] is for the green, and [2] is for the blue. They add up to 1.

**pnm_gammarown()** and **pnm_ungammarown()** apply and unapply gamma correction to a row of an image using the same transformation as **pm_gamma()** and **pm_ungamma()**. Note that these operate on a row of normalized tuples (**tuplen**, not **tuple**).

**pnm_applyopacity()** reduces the intensity of samples in accordance with the opacity plane of an image. The opacity plane, if it exists, tells how much of the light from that pixel should show when the image is composed with another image. You use **pnm_applyopacity()** in preparation for doing such a composition. For example, if the opacity plane says that the top half of the image is 50% opaque and the bottom half 100% opaque, **pnm_applyopacity()** will reduce the intensity of each sample of each tuple (pixel) in the upper half of the image by 50%, and leave the rest alone.

If the image does not have an opacity plane (i.e. its tuple type is not one that **libnetpbm** recognizes as having an opacity plane), **pnm_applyopacity()** does nothing (which is the same as assuming opacity 100%). The tuple types that **libnetpbm** recognizes as having opacity are **RGB_ALPHA** and **GRAYSCALE_ALPHA**.

**pnm_unapplyopacity()** does the reverse. It assumes the intensities are already reduced according to the opacity plane, and raises back to normal.

**pnm_applyopacity()** works on (takes as input and produces as output) normalized, intensity-proportional tuples. That means you will typically read the row from the image file with **pnm_readpamrown()** and then gamma-correct it with **pnm_ungammarown()**, and then do **pnm_applyopacity()**. You then manipulate the row further (perhaps add it with other rows you've processed similarly), then do **pnm_unapplyopacity()**, then **pnm_gammarown()**, then **pnm_writegammarown()**.

**pnm_normalizeTuple()** and **pnm_unnormalizeTuple()** convert between a **tuple** data type and a **tuplen** data type. The former represents a sample value using the same unsigned integer that is in the PAM image, while the latter represents a sample value as a number scaled by the maxval to the range 0..1. I.e. **pnm_normalizeTuple()** divides every sample value by the maxval and **pnm_unnormalizeTuple()** multiples every sample by the maxval.

**pnm_normalizeRow()** and **pnm_unnormalizeRow()** do the same thing on an entire tuple row, but also have an extra feature: You can specify a transform function to be applied in addition. Typically, this is a gamma transform function. You can of course more easily apply your transform function separately from normalizing, but doing it all at once is usually way faster. Why? Because you can use a lookup table that is indexed by an integer on one side and produces a floating point number on the other. To do it separately, you'd either have to do floating point arithmetic on the normalized value or do the transform on the integer values and lose a lot of precision.

If you don't have any transformation to apply, just specify **NULL** for the *transform* argument and the function will just normalize (i.e. divide or multiply by the maxval).

Here's an example of doing a transformation. The example composes two images together, something that has to be done with intensity-linear sample values.

```
pnm_transformMap * const transform1 = pnm_createungammatransform(&inpam1);
pnm_transformMap * const transform2 = pnm_createungammatransform(&inpam2);
pnm_transformMap * const transformOut = pnm_creategammatransform(&outpam);
pnm_readpamrow(&inpam1, inrow1);
pnm_readpamrow(&inpam2, inrow2);
pnm_normalizeRow(&inpam1, inrow1, transform1, normInrow1);
pnm_normalizeRow(&inpam2, inrow2, transform2, normInrow2);
for (col = 0; col < outpam.width; ++col)
   normOutrow[col] = (normInrow1[col] + normInrow2[col])/2;
pnm_unnormalizeRow(&outpam, normOutrow, transformOut, outrow);
pnm_writepamrow(&outpam, outrow);
```

To specify a transform, you must create a special **pnm_transformMap** object and pass it as the *transform* argument. Typically, your transform is a gamma transformation because you want to work in intensity-proportional sample values and the PAM image format uses gamma-adjusted ones. In that case, just use **pnm_creategammtransform()** and **pnm_createungammatransform()** to create this object and don't worry about what's inside it.

**pnm_creategammatransform()** and **pnm_createungammatransform()** create objects that you use with **pnm_normalizeRow()** and **pnm_unnormalizeRow()** as described above. The created object describes a transform that applies or reverses the ITU-R Recommendation BT.709 gamma adjustment that is used in PAM visual images and normalizes or unnormalizes the sample values. **pnm_freegammatransform()** and **pnm_freeungammatransform()** destroy the objects.

**Miscellaneous**

**Synopsis**

**void pnm_checkpam( struct pam ****pamP***, const enum pm_check_type** *check_type***, enum pm_check_code ****retvalP***);**

**void pnm_nextimage( FILE ****file***, int * const** *eofP***);**

**Description**

**pnm_checkpam()** checks for the common file integrity error where the file is the wrong size to contain the raster, according to the information in the header.

**pnm_nextimage**()positions a Netpbm image input file to the next image in it (so that a subsequent **pnm_readpaminit**() reads its header).

---

**Table Of Contents**


Types

Macros

Functions

Memory Management

Reading Netpbm Files

Writing Netpbm Files

Transforming Pixels

Miscellaneous