Picamera raw analysis

Release 0.1.0

Nov 22, 2021

Contents:

This module contains utilities for manipulating raw images from the Raspberry Pi Camera Module, version 2.

If you use it with images taken with other cameras, including version 1, it will most likely fail.

You can use this module on the command line to extract raw data from a JPEG file:

```
python -m picam_raw_analysis.extract_raw_image image.jpg
```

There is a script provided that will convert a white image into a lens shading table, in YAML format. To run this, use:

```
python -m picam_raw_analysis.lst_from_raw_white_image path/to/white/image.jpg --
↪output lens_shading.yaml
```

There is also a script to completely compensate an image, using the raw data in that image together with red, green, blue, and white calibration images. This can be run with:

```
python -m picam_raw_analysis.unmix_image path/to/calibration/folder image.jpg
```

Most of the functionality lives in submodules, but `load_raw_image` and `extract_file` are available at the top level as well as in the `extract_raw_image` submodule.

Finally, this module does not depend on `picamera` and should run (in Python 3) on any platform. To achieve this, `picamera_array` has been copied into this module, and `mo_stub` provides a dummy MMAL import. I'm not the author of `picamera.array` and it is copied here (in modified form) under the GPL.

This module has been written with the intention of it working in Python 2 or 3, but it has mostly been tested in Python 3.

CHAPTER 1

picam_raw_analysis package

1.1 Module contents

This module contains utilities for manipulating raw images from the Raspberry Pi Camera Module, version 2.

If you use it with images taken with other cameras, including version 1, it will most likely fail.

You can use this module on the command line to extract raw data from a JPEG file:

```
python -m picam_raw_analysis.extract_raw_image image.jpg
```

There is a script provided that will convert a white image into a lens shading table, in YAML format. To run this, use:

```
python -m picam_raw_analysis.lst_from_raw_white_image path/to/white/image.jpg --
↪output lens_shading.yaml
```

There is also a script to completely compensate an image, using the raw data in that image together with red, green, blue, and white calibration images. This can be run with:

```
python -m picam_raw_analysis.unmix_image path/to/calibration/folder image.jpg
```

Most of the functionality lives in submodules, but `load_raw_image` and `extract_file` are available at the top level as well as in the `extract_raw_image` submodule.

Finally, this module does not depend on `picamera` and should run (in Python 3) on any platform. To achieve this, `picamera_array` has been copied into this module, and `mo_stub` provides a dummy MMAL import. I'm not the author of `picamera.array` and it is copied here (in modified form) under the GPL.

This module has been written with the intention of it working in Python 2 or 3, but it has mostly been tested in Python 3.

(c) Richard Bowman 2019, released under GNU GPL v3

1.2 Submodules

1.3 picam_raw_analysis.extract_raw_image module

This module handles the extraction of raw data from JPEG + RAW files saved by the Raspberry Pi camera module, v2. It has not been tested with v1, and most likely will fail; there is no logic included for determining the sensor version from the EXIF metadata.

It can be run from the command line:

```
python -m picam_raw_analysis.extract_raw_image image.jpg
```

Multiple filenames may be specified, and the output images will be saved with the same filenames plus plus a suffix for each output type. All three output types will be generated for each input file:

`_raw16.tif`: 16-bit TIFF image with the full dynamic range of the 10-bit raw image. Each pixel will have a value ranging from 0 to 1024.

`_raw8.tif`: 8-bit TIFF image containing the top 8 bits of the 10-bit raw image.

`_exif.txt`: Extracted metadata from the JPEG file, in plain text format.

All of these functions are also accessible through the member functions of the module.

Copyright 2019 Richard Bowman, released under GNU GPL v3

class `picam_raw_analysis.extract_raw_image.`**`DummyCam`**

Bases: `object`

`resolution = (3280, 2464)`

`revision = 'IMX219'`

`sensor_mode = 0`

`picam_raw_analysis.extract_raw_image.`**`extract_file`**(*filename*, *extract_exif=True*, *extract_png=True*)

Extract metadata and raw image from a file, saving it as a text file and 8 and 16-bit TIFF images.

`picam_raw_analysis.extract_raw_image.`**`extract_file_cli`**()

Extract the raw data from a file using argparse for command line arguments

`picam_raw_analysis.extract_raw_image.`**`load_raw_image`**(*filename*, *ArrayType=<class 'picam_raw_analysis.picamera_array.PiSharpBayerArray'>*, *open_jpeg=False*)

Load the raw image data (and optionally the processed image data and EXIF metadata) from a file

1.4 picam_raw_analysis.lst_from_raw_white_image module

Generate a lens shading table based on previously-measured images and test it on the camera.

This script uses a raw white image to correct the lens shading table of a Raspberry Pi camera module. The revised lens shading table is uploaded to the camera, and images are acquired with it to test it's working properly.

It can be run from the command line:

```
python -m picam_raw_analysis.lst_from_raw_white_image path/to/white/image.jpg --
↪output lens_shading.yaml
```

Use the `--help` flag to obtain a usage message.

Copyright 2019 Richard Bowman, released under GNU GPL v3

`picam_raw_analysis.lst_from_raw_white_image.`**`channels_from_bayer_array`**(*bayer_array*)
Given the 'array' from a PiBayerArray, return the 4 channels.

`picam_raw_analysis.lst_from_raw_white_image.`**`lst_from_channels`**(*channels*)
Given the 4 Bayer colour channels from a white image, generate a LST.

1.5 picam_raw_analysis.unmix_image module

This script will correct for vignetting and saturation loss at the edges of a picamera v2 image.

It uses unmixing matrices generated by `picam_raw_analysis.unmixing_matrix`.

It can be run from the command line:

```
python -m picam_raw_analysis.unmix_image path/to/calibration/folder image.jpg
```

Use the `--help` flag to obtain a usage message.

Copyright Richard Bowman 2019, released under GNU GPL v3 or later

`picam_raw_analysis.unmix_image.`**`correct_image`**(*image*, *unmixing_matrix=None*, *norm_to_white=None*)
Process an image to remove vignetting and saturation loss.

The image should be an NxMx3 numpy array.

The unmixing matrix should be an NxMx3x3 array.

The normalisation image should be NxMx3

`picam_raw_analysis.unmix_image.`**`main`**()
Process images from the command line

`picam_raw_analysis.unmix_image.`**`upsample_1d`**(*arr*, *axis=0*, *zoom=16*)
Upsample one dimension of an array

`picam_raw_analysis.unmix_image.`**`upsample_xy`**(*arr*, *zoom=16*)
Upscale the X and Y dimensions of an image/spatially varying matrix

1.6 picam_raw_analysis.unmixing_matrix module

This module takes in red, green, blue, white images, and calculates the colour crosstalk that's happening. We then invert this matrix to undo the mixing.

(c) Richard Bowman 2019, released under GNU GPL v3 or later

`picam_raw_analysis.unmixing_matrix.`**`add_unmixing_args`**(*parser*)
Add the arguments for colour unmixing to an argparse.ArgumentParser

`picam_raw_analysis.unmixing_matrix.`**`bin`**(*image*, *b=2*)
Bin bxb squares of an image together

`picam_raw_analysis.unmixing_matrix.`**`calculate_calibration`**(*args*)
Based on the command-line args supplied, calculate unmixing and vignetting corrections

`picam_raw_analysis.unmixing_matrix.`**`central_colour`**(*image*) `picam_raw_analysis.unmixing_matrix.`**`colour_unmix_image`**(*image*, *calibration*, ***kwargs*)

Take a test image, and a set of W/R/G/B calibration images, and unmix the test image.

Arguments: image: NxNx3 image as a numpy.ndarray calibration: a dictionary with (at least) R, G, B, and W images keyword arguments are passed to colour_unmixing_matrices

Returns: an NxMx3 ndarray containing the unmixed image

`picam_raw_analysis.unmixing_matrix.`**`colour_unmixing_matrices`**(*cal*, *colour_target='rgb'*, *smoothing=None*)

Return a matrix that turns the camera's recorded colour back into "perfect" colour

cal should be a calibration run (dictionary) with, as a minimum, W, R, G, and B images.

calibration [dict] a dictionary with (at least) R, G, B, and W images

colour_target: string "rgb" (default) or "centre". "rgb" will unmix to fully saturated colours, while "centre" will unmix so the edges of the image match the centre of the image.

smoothing: None or float (default) for no smoothing, or a number (in pixels) to apply a Gaussian blur to the compensation matrices.

returns: an NxMx3x3 unmixing matrix

`picam_raw_analysis.unmixing_matrix.`**`crosstalk_matrices`**(*run*)

Construct a 4d array of colour crosstalk information.

This function returns a 3x3 matrix at each pixel of the calibration images. Inverting this matrix

`picam_raw_analysis.unmixing_matrix.`**`load_raw_image_and_bin`**(*filename*)

Load an image from the raw data in a jpeg file, and return a binned version.

`picam_raw_analysis.unmixing_matrix.`**`load_run`**(*folder*, *illuminations*)

Load the R,G,B,W calibration images

`picam_raw_analysis.unmixing_matrix.`**`main`**()

Construct a colour unmixing matrix and save it to a YAML file

1.7 picam_raw_analysis.dump_exif module

`picam_raw_analysis.dump_exif.`**`exif_data_as_string`**(*image*)

Extract the EXIF data from a PIL image object, and format as a string.

`picam_raw_analysis.dump_exif.`**`formatted_exif_data`**(*image*)

Retrieve an image's EXIF data and return as a dictionary with string keys

`picam_raw_analysis.dump_exif.`**`kv_to_string`**(*k*, *v*, *format=''*)

Consistently output a key-value pair as text

`picam_raw_analysis.dump_exif.`**`parse_maker_note`**(*maker_note*)

Split the "maker note" EXIF field from a Raspberry Pi camera image into useful parameters

`picam_raw_analysis.dump_exif.`**`print_kv`**(*k*, *v*, *format=''*)

Consistently print a key-value pair

1.8 picam_raw_analysis.picamera_array module

The picamera_array module is lifted more or less directly from the *picamera* module. It has been modified to remove dependencies on the rest of *picamera*, and has been extended to include some additional array types, notably *picam_raw_analysis.picamera_array.PiSharpBayerArray* (which implements minimally smoothed debayering).

class `picam_raw_analysis.picamera_array.`**`BroadcomRawHeader`**

Bases: `_ctypes.Structure`

`bayer_format`
Structure/Union member

`bayer_order`
Structure/Union member

`dummy`
Structure/Union member

`format`
Structure/Union member

`height`
Structure/Union member

`name`
Structure/Union member

`padding_down`
Structure/Union member

`padding_right`
Structure/Union member

`transform`
Structure/Union member

`width`
Structure/Union member

class `picam_raw_analysis.picamera_array.`**`PiAnalysisOutput`**(*camera*, *size=None*)

Bases: `io.IOBase`

Base class for analysis outputs.

This class extends `io.IOBase` with a stub *`analyze()`* method which will be called for each frame output. In this base implementation the method simply raises `NotImplementedError`.

`analyse`(*array*)
Deprecated alias of *`analyze()`*.

`analyze`(*array*)
Stub method for users to override.

`writable`()
Return whether object was opened for writing.

If False, write() will raise OSError.

`write`(*b*)

class `picam_raw_analysis.picamera_array.`**`PiArrayOutput`**(*camera*, *size=None*)

Bases: `_io.BytesIO`

Base class for capture arrays.

This class extends `io.BytesIO` with a *numpy* array which is intended to be filled when `flush()` is called (i.e. at the end of capture).

array
After `flush()` is called, this attribute contains the frame's data as a multi-dimensional *numpy* array. This is typically organized with the dimensions `(rows, columns, plane)`. Hence, an RGB image with dimensions *x* and *y* would produce an array with shape `(y, x, 3)`.

close()
Disable all I/O operations.

truncate(*size=None*)
Resize the stream to the given size in bytes (or the current position if size is not specified). This resizing can extend or reduce the current file size. The new file size is returned.

In prior versions of picamera, truncation also changed the position of the stream (because prior versions of these stream classes were non-seekable). This functionality is now deprecated; scripts should use `seek()` and *`truncate()`* as one would with regular `BytesIO` instances.

class picam_raw_analysis.picamera_array.**PiBayerArray**(*camera*, *output_dims=3*)
Bases: *`picam_raw_analysis.picamera_array.PiArrayOutput`*

Produces a 3-dimensional RGB array from raw Bayer data.

This custom output class is intended to be used with the `capture()` method, with the *bayer* parameter set to `True`, to include raw Bayer data in the JPEG output. The class strips out the raw data, and constructs a numpy array from it. The resulting data is accessed via the *`array`* attribute:

```
import picamera
import picamera.array

with picamera.PiCamera() as camera:
    with picamera.array.PiBayerArray(camera) as output:
        camera.capture(output, 'jpeg', bayer=True)
        print(output.array.shape)
```

The *output_dims* parameter specifies whether the resulting array is three-dimensional (the default, or when *output_dims* is 3), or two-dimensional (when *output_dims* is 2). The three-dimensional data is already separated into the three color planes, whilst the two-dimensional variant is not (in which case you need to know the Bayer ordering to accurately deal with the results).

Note: Bayer data is *usually* full resolution, so the resulting array usually has the shape (1944, 2592, 3) with the V1 module, or (2464, 3280, 3) with the V2 module (if two-dimensional output is requested the 3-layered color dimension is omitted). If the camera's `sensor_mode` has been forced to something other than 0, then the output will be the native size for the requested sensor mode.

This also implies that the optional *size* parameter (for specifying a resizer resolution) is not available with this array class.

As the sensor records 10-bit values, the array uses the unsigned 16-bit integer data type.

By default, de-mosaicing is **not** performed; if the resulting array is viewed it will therefore appear dark and too green (due to the green bias in the Bayer pattern). A trivial weighted-average demosaicing algorithm is provided in the *`demosaic()`* method:

```
import picamera
import picamera.array

```

(continues on next page)

(continued from previous page)

```
with picamera.PiCamera() as camera:
    with picamera.array.PiBayerArray(camera) as output:
        camera.capture(output, 'jpeg', bayer=True)
        print(output.demosaic().shape)
```

Viewing the result of the de-mosaiced data will look more normal but still considerably worse quality than the regular camera output (as none of the other usual post-processing steps like auto-exposure, white-balance, vignette compensation, and smoothing have been performed).

Changed in version 1.13: This class now supports the V2 module properly, and handles flipped images, and forced sensor modes correctly.

BAYER_OFFSETS = {0: ((0, 0), (1, 0), (0, 1), (1, 1)), 1: ((1, 0), (0, 0), (1, 1), (0

data_to_array(*data*)
Convert the cropped, reshaped array of 8 bit numbers into a sensible array

demosaic()
Perform a rudimentary de-mosaic of `self.array`, returning the result as a new array. The result of the demosaic is *always* three dimensional, with the last dimension being the color planes (see *output_dims* parameter on the constructor).

flush()
Does nothing.

output_dims

class picam_raw_analysis.picamera_array.**PiFastBayerArray**(*camera*, *output_dims=3*)
Bases: *picam_raw_analysis.picamera_array.PiBayerArray*

data_to_array(*data*)
Convert the cropped, reshaped array of 8 bit numbers into a sensible array

demosaic(*shift=0*)
Convert the raw Bayer data into a half-resolution RGB array.

This uses a really blunt demosaicing algorithm: group pixels in squares, and then use the red, blue, and two green pixels from each square to calculate an RGB value. This is calculated as three unsigned 8-bit integers.

As the sensor is 10 bit but output is 8-bit, we provide the `shift` parameter. Setting this to 2 will return the lower 8 bits, while setting it to 0 (the default) will return the upper 8 bits. In the future, there may be an option to work in 16-bit integers and return all of them (though that would be slower). Currently, if `shift` is nonzero and some pixels have higher values than will fit in the 8-bit output, overflow will occur and those pixels may no longer be bright - so use the `shift` argument with caution.

NB that the highest useful `shift` value is 3; while the sensor is only 10-bit, there are two green pixels on the sensor for each output pixel. Thus, we gain an extra bit of precision from averaging, allowing us to effectively produce an 11-bit image.

class picam_raw_analysis.picamera_array.**PiMotionAnalysis**(*camera*, *size=None*)
Bases: *picam_raw_analysis.picamera_array.PiAnalysisOutput*

Provides a basis for real-time motion analysis classes.

This custom output class is intended to be used with the *motion_output* parameter of the `start_recording()` method. While recording is in progress, the write method converts incoming motion data into numpy arrays and calls the stub *analyze()* method with the resulting array (which deliberately raises `NotImplementedError` in this class).

Note: If your overridden `analyze()` method runs slower than the required framerate (e.g. 33.333ms when framerate is 30fps) then the camera's effective framerate will be reduced. Furthermore, this doesn't take into account the overhead of picamera itself so in practice your method needs to be a bit faster still.

The array passed to `analyze()` is organized as (rows, columns) where `rows` and `columns` are the number of rows and columns of macro-blocks (16x16 pixel blocks) in the original frames. There is always one extra column of macro-blocks present in motion vector data.

The data-type of the array is an (x, y, sad) structure where `x` and `y` are signed 1-byte values, and `sad` is an unsigned 2-byte value representing the sum of absolute differences of the block.

An example of a crude motion detector is given below:

```
import numpy as np
import picamera
import picamera.array

class DetectMotion(picamera.array.PiMotionAnalysis):
    def analyze(self, a):
        a = np.sqrt(
            np.square(a['x'].astype(np.float)) +
            np.square(a['y'].astype(np.float))
            ).clip(0, 255).astype(np.uint8)
        # If there're more than 10 vectors with a magnitude greater
        # than 60, then say we've detected motion
        if (a > 60).sum() > 10:
            print('Motion detected!')

with picamera.PiCamera() as camera:
    with DetectMotion(camera) as output:
        camera.resolution = (640, 480)
        camera.start_recording(
              '/dev/null', format='h264', motion_output=output)
        camera.wait_recording(30)
        camera.stop_recording()
```

You can use the optional *size* parameter to specify the output resolution of the GPU resizer, if you are using the *resize* parameter of `start_recording()`.

write(*b*)

class picam_raw_analysis.picamera_array.**PiMotionArray**(*camera*, *size=None*)
Bases: `picam_raw_analysis.picamera_array.PiArrayOutput`

Produces a 3-dimensional array of motion vectors from the H.264 encoder.

This custom output class is intended to be used with the *motion_output* parameter of the `start_recording()` method. Once recording has finished, the class generates a 3-dimensional numpy array organized as (frames, rows, columns) where `rows` and `columns` are the number of rows and columns of macro-blocks (16x16 pixel blocks) in the original frames. There is always one extra column of macro-blocks present in motion vector data.

The data-type of the `array` is an (x, y, sad) structure where `x` and `y` are signed 1-byte values, and `sad` is an unsigned 2-byte value representing the sum of absolute differences of the block. For example:

```
import picamera
import picamera.array
```

(continues on next page)

(continued from previous page)

```
with picamera.PiCamera() as camera:
    with picamera.array.PiMotionArray(camera) as output:
        camera.resolution = (640, 480)
        camera.start_recording(
              '/dev/null', format='h264', motion_output=output)
        camera.wait_recording(30)
        camera.stop_recording()
        print('Captured %d frames' % output.array.shape[0])
        print('Frames are %dx%d blocks big' % (
            output.array.shape[2], output.array.shape[1]))
```

If you are using the GPU resizer with your recording, use the optional *size* parameter to specify the resizer's output resolution when constructing the array:

```
import picamera
import picamera.array

with picamera.PiCamera() as camera:
    camera.resolution = (640, 480)
    with picamera.array.PiMotionArray(camera, size=(320, 240)) as output:
        camera.start_recording(
            '/dev/null', format='h264', motion_output=output,
            resize=(320, 240))
        camera.wait_recording(30)
        camera.stop_recording()
        print('Captured %d frames' % output.array.shape[0])
        print('Frames are %dx%d blocks big' % (
            output.array.shape[2], output.array.shape[1]))
```

Note: This class is not suitable for real-time analysis of motion vector data. See the `PiMotionAnalysis` class instead.

`flush``()`
Does nothing.

class `picam_raw_analysis.picamera_array.`**`PiRGBAnalysis`**`(`*camera*, *size=None*`)`
Bases: `picam_raw_analysis.picamera_array.PiAnalysisOutput`

Provides a basis for per-frame RGB analysis classes.

This custom output class is intended to be used with the `start_recording()` method when it is called with *format* set to `'rgb'` or `'bgr'`. While recording is in progress, the `write()` method converts incoming frame data into a numpy array and calls the stub `analyze()` method with the resulting array (this deliberately raises `NotImplementedError` in this class; you must override it in your descendent class).

Note: If your overridden `analyze()` method runs slower than the required framerate (e.g. 33.333ms when framerate is 30fps) then the camera's effective framerate will be reduced. Furthermore, this doesn't take into account the overhead of picamera itself so in practice your method needs to be a bit faster still.

The array passed to `analyze()` is organized as (rows, columns, channel) where the channels 0, 1, and 2 are R, G, and B respectively (or B, G, R if *format* is `'bgr'`).

`write``(`*b*`)`

class picam_raw_analysis.picamera_array.**PiRGBArray**(*camera*, *size=None*)

Bases: *picam_raw_analysis.picamera_array.PiArrayOutput*

Produces a 3-dimensional RGB array from an RGB capture.

This custom output class can be used to easily obtain a 3-dimensional numpy array, organized (rows, columns, colors), from an unencoded RGB capture. The array is accessed via the *array* attribute. For example:

```
import picamera
import picamera.array

with picamera.PiCamera() as camera:
    with picamera.array.PiRGBArray(camera) as output:
        camera.capture(output, 'rgb')
        print('Captured %dx%d image' % (
                output.array.shape[1], output.array.shape[0]))
```

You can re-use the output to produce multiple arrays by emptying it with `truncate(0)` between captures:

```
import picamera
import picamera.array

with picamera.PiCamera() as camera:
    with picamera.array.PiRGBArray(camera) as output:
        camera.resolution = (1280, 720)
        camera.capture(output, 'rgb')
        print('Captured %dx%d image' % (
                output.array.shape[1], output.array.shape[0]))
        output.truncate(0)
        camera.resolution = (640, 480)
        camera.capture(output, 'rgb')
        print('Captured %dx%d image' % (
                output.array.shape[1], output.array.shape[0]))
```

If you are using the GPU resizer when capturing (with the *resize* parameter of the various `capture()` methods), specify the resized resolution as the optional *size* parameter when constructing the array output:

```
import picamera
import picamera.array

with picamera.PiCamera() as camera:
    camera.resolution = (1280, 720)
    with picamera.array.PiRGBArray(camera, size=(640, 360)) as output:
        camera.capture(output, 'rgb', resize=(640, 360))
        print('Captured %dx%d image' % (
                output.array.shape[1], output.array.shape[0]))
```

flush()

Does nothing.

class picam_raw_analysis.picamera_array.**PiSharpBayerArray**(*camera*, *output_dims=3*)

Bases: *picam_raw_analysis.picamera_array.PiBayerArray*

A PiBayerArray, demosaiced so as to preserve sharpness a bit more (esp. for green)

demosaic()

Perform a rudimentary de-mosaic of `self.array`, returning the result as a new array. The result of the demosaic is *always* three dimensional, with the last dimension being the color planes (see *output_dims* parameter on the constructor).

class picam_raw_analysis.picamera_array.**PiYUVAnalysis**(*camera*, *size=None*)

Bases: `picam_raw_analysis.picamera_array.PiAnalysisOutput`

Provides a basis for per-frame YUV analysis classes.

This custom output class is intended to be used with the `start_recording()` method when it is called with *format* set to `'yuv'`. While recording is in progress, the `write()` method converts incoming frame data into a numpy array and calls the stub `analyze()` method with the resulting array (this deliberately raises `NotImplementedError` in this class; you must override it in your descendent class).

Note: If your overridden `analyze()` method runs slower than the required framerate (e.g. 33.333ms when framerate is 30fps) then the camera's effective framerate will be reduced. Furthermore, this doesn't take into account the overhead of picamera itself so in practice your method needs to be a bit faster still.

The array passed to `analyze()` is organized as (rows, columns, channel) where the channel 0 is Y (luminance), while 1 and 2 are U and V (chrominance) respectively. The chrominance values normally have quarter resolution of the luminance values but this class makes all channels equal resolution for ease of use.

write(*b*)

class picam_raw_analysis.picamera_array.**PiYUVArray**(*camera*, *size=None*)

Bases: `picam_raw_analysis.picamera_array.PiArrayOutput`

Produces 3-dimensional YUV & RGB arrays from a YUV capture.

This custom output class can be used to easily obtain a 3-dimensional numpy array, organized (rows, columns, channel), from an unencoded YUV capture. The array is accessed via the `array` attribute. For example:

```
import picamera
import picamera.array

with picamera.PiCamera() as camera:
    with picamera.array.PiYUVArray(camera) as output:
        camera.capture(output, 'yuv')
        print('Captured %dx%d image' % (
                output.array.shape[1], output.array.shape[0]))
```

The `rgb_array` attribute can be queried for the equivalent RGB array (conversion is performed using the ITU-R BT.601 matrix):

```
import picamera
import picamera.array

with picamera.PiCamera() as camera:
    with picamera.array.PiYUVArray(camera) as output:
        camera.resolution = (1280, 720)
        camera.capture(output, 'yuv')
        print(output.array.shape)
        print(output.rgb_array.shape)
```

If you are using the GPU resizer when capturing (with the *resize* parameter of the various `capture()` methods), specify the resized resolution as the optional *size* parameter when constructing the array output:

```
import picamera
import picamera.array

with picamera.PiCamera() as camera:
    camera.resolution = (1280, 720)
```

(continues on next page)

(continued from previous page)

```
    with picamera.array.PiYUVArray(camera, size=(640, 360)) as output:
        camera.capture(output, 'yuv', resize=(640, 360))
        print('Captured %dx%d image' % (
                output.array.shape[1], output.array.shape[0]))
```

flush()
Does nothing.

rgb_array

picam_raw_analysis.picamera_array.**bytes_to_rgb**(*data*, *resolution*)
Converts a bytes objects containing RGB/BGR data to a *numpy* array.

picam_raw_analysis.picamera_array.**bytes_to_yuv**(*data*, *resolution*)
Converts a bytes object containing YUV data to a *numpy* array.

picam_raw_analysis.picamera_array.**raw_resolution**(*resolution*, *splitter=False*)
Round a (width, height) tuple up to the nearest multiple of 32 horizontally and 16 vertically (as this is what the Pi's camera module does for unencoded output).

1.9 picam_raw_analysis.mo_stub module

This module exists only for convenience, to allow the dummy picamera_array module to load with a minimum of modified code.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index