Raspberry Pi Camera Algorithm and Tuning Guide

Raspberry Pi Trading Ltd.

Colophon

Compiled on 31/01/2021
Version 1.2(5b3eff95b9618ac0829cf76181539efbad07e9f2)

Revision History

Version	Date	Description
1.2	30-Jan-2021	Update formatting and style template to match internal guidelines.
1.1	27-Jan-2021	Update to CamHelper and various algorithm APIs. Also use of qcam replaced by libcamera-apps.
1.0	15-May-2020	Initial revision.

Contents

List of Figures

List of Tables

1 Introduction

The "Raspberry Pi Camera Algorithm and Tuning Guide" is intended for users of the Raspberry Pi computer with an image sensor (camera) connected through the Raspberry Pi's CSI (Camera Serial Interface) camera port, such as either of the official Raspberry Pi camera boards using the version 1 (Omnivision OV5647) or version 2 (Sony imx219) sensors, or the High Quality Camera (*aka.* the HQ Cam, based on the Sony imx477).

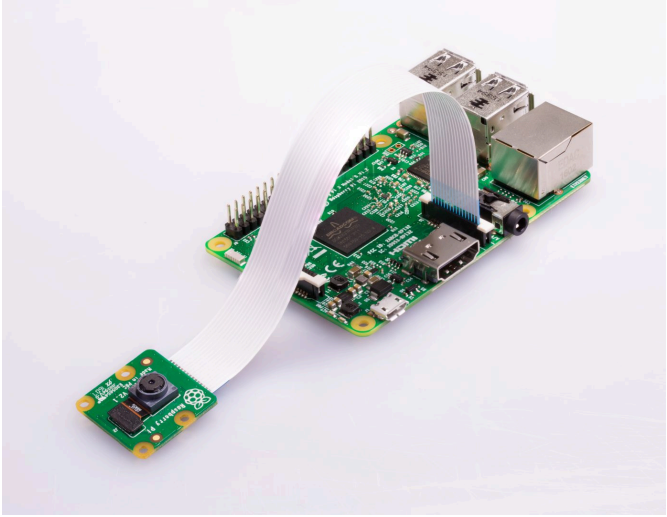

Figure 1: Connecting a camera board through the CSI port.

The software stack driving the camera system will be *libcamera*. Experience has shown that driving complex camera systems directly through kernel (typically, V4L2) drivers is very difficult, often leading to large amounts of undesirable and highly platform-specific application code. For this reason a much higher level userspace camera stack, *libcamera*, has emerged providing mechanisms to integrate 3rd party image sensors and Image Signal Processors (ISPs).

Here we describe just such an integration, showing how drivers can be written to make the components of the Raspberry Pi imaging system work with *libcamera*, concentrating in particular on the processes of calibrating and tuning the ISP to work well with different image sensors. Moreover, the *libcamera* integration avoids using any of the proprietary control algorithms supplied by the chip vendor (Broadcom). Rather, Raspberry Pi is providing its own control algorithms, running directly on the chip's ARM cores, as open source code which can be easily inspected and modified by users.

This document is not specifically a tutorial on or guide to *libcamera* itself, for which the reader is referred to http://libcamera.org for more information.

The remainder of this document is organised as follows.

- Chapter 2: Overview. This is a brief overview of the system as a whole, showing what is already provided, and what must be added in terms of image sensor drivers and ISP control algorithms to obtain a successful images.
- Chapter 3: Camera Drivers. As far as possible the camera, or image sensor, is driven by a standard type of kernel driver. However, there are one or two details involved in making drivers work optimally in the Raspberry Pi framework.
- Chapter 4: Control Algorithm Overview. Raspberry Pi provides a control algorithm framework which makes it very easy for applications to use Raspberry Pi's, or anyone else's, ISP control algorithms.
- Chapter 5: Raspberry Pi Control Algorithms. Here we supply a detailed description of Raspberry Pi's implementation of the control algorithms.
- Chapter 6: Camera Tuning Tool. Finally, there is also a Camera Tuning Tool which automates most of the process of tuning the ISP control algorithms for different image sensors.

For further help beyond the contents of this user guide, please visit the Raspberry Pi Camera Board forum at https://www.raspberrypi.org/forums.

2 Overview

We have already seen the camera board connected through the ribbon cable to the CSI port on the Raspberry Pi itself.

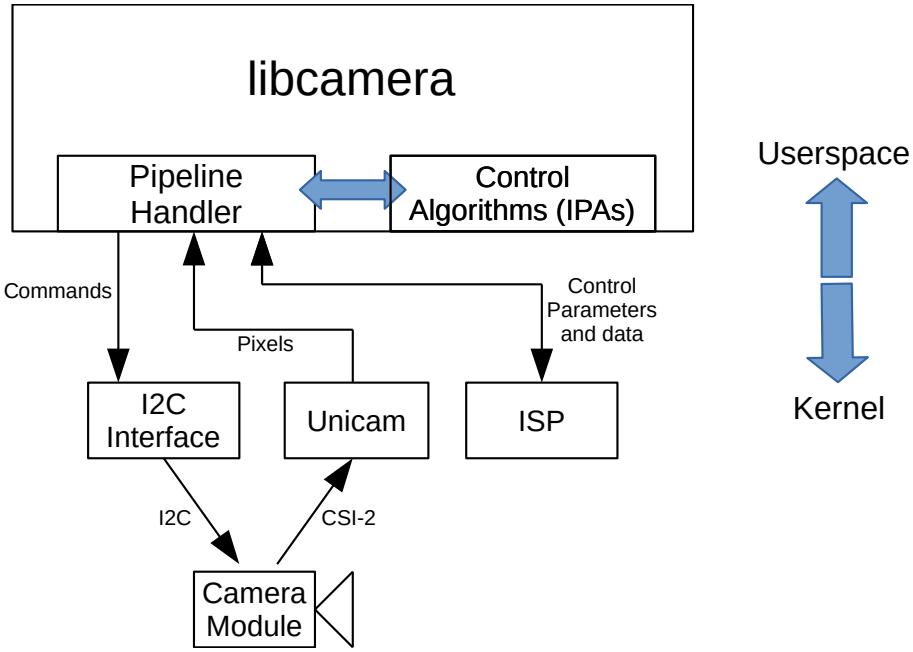

Figure 2: Overview of the libcamera system, explained below.

2.1 The Camera Module

Raspberry Pi already produces compatible camera boards based on the Sony imx219 (the version 2 camera), the Sony imx477 (the HQ Cam) and of course there is an older (version 1) camera board based on the Omnivision OV5647.

These are Bayer sensors, that is, they output so-called "raw" Bayer images which have not yet been processed into anything a user could recognise. These raw pixels are transmitted back to the system-on-chip (SoC) in discrete image *frames*, using the MIPI CSI-2 protocol. This is the 2nd version of the CSI (Camera Serial Interface) defined by the MIPI (originally "Mobile Industry Processor Interface") Alliance, an organisation dedicated to standardising interfaces between different components of mobile devices.

2.2 CSI Connector

Besides supplying power and clock signals, the ribbon cable performs two principal functions. It allows commands (typically register updates) to be sent to the image sensor using the I^2C interface. These values define the operating mode of the sensor, including the output image size and framerate, as well as commands to start and stop streaming images.

Secondly, the output images from the camera are transmitted through the cable along the CSI-2 interface back to the Broadcom SoC (system-on-chip) at the heart of the Pi.

A more detailed specification of the modules and connector, including a link to the schematics, can be found at https://www.raspberrypi.org/documentation/hardware/camera/README.md.

2.3 On-Chip Hardware

On the Broadcom SoC we have the following.

- An I^2C interface, for sending commands to the image sensor.
- A CSI-2 Receiver, for receiving image frames, in the form of pixel data, back from the sensor. The CSI-2 Receiver on the Pi's Broadcom SoC is also referred to by the name *Unicam*.
- And an ISP (Image Signal Processor) which converts the raw Bayer images from the sensor into something a user might recognise. The ISP also produces statistics from the pixel data, from which a number of software control algorithms will deduce appropriate parameters to be fed to the ISP in order to produce pleasing output images.

Interactions between these on-chip hardware devices is mediated though interrupts, responded to by the camera software stack.

2.4 Software and Control Algorithms

libcamera is responsible for handling and fulfilling application requests using Raspberry Pi's *pipeline handler*. In turn, the pipeline handler must ensure that the camera and ISP control algorithms are invoked at the correct moment, and must fetch up to date parameter values for each image frame.

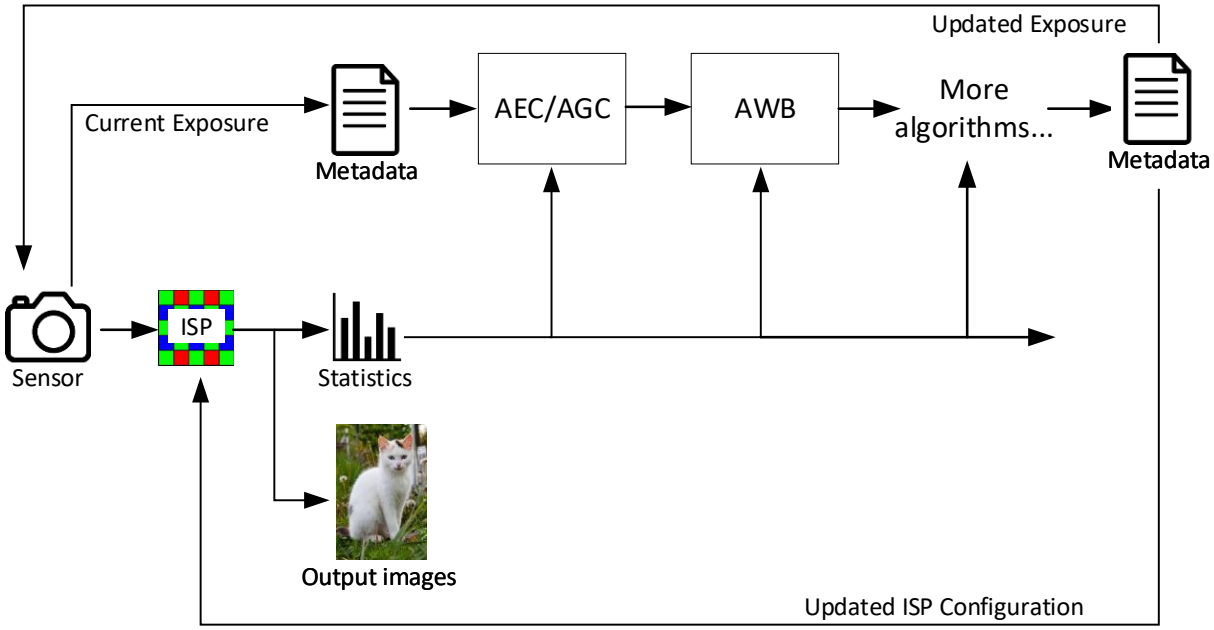

Figure 3: Control algorithms use image metadata and statistics to update their parameters.

The Raspberry Pi control algorithm framework associates a buffer of *metadata* with each image, containing exposure and other information about it. The algorithms each receive this metadata and statistics about the image which were calculated by the hardware ISP. They run in turn to compute new and updated values which are deposited back into the metadata.

Finally, the pipeline handler reads values out of the metadata and uses them to update the camera's exposure settings and the ISP parameters for the next frame. There will be more details on this in chapter 4 (Control Algorithm Overview).

3 Driver Framework

Our *Unicam* kernel driver follows the standard Video for Linux 2 (V4L2) API (see https://linuxtv.org/downloads/v4l-dvb-apis/uapi/v4l/v4l2.html). It is responsible for:

1. Interfacing with the V4L2 sensor subdevice driver to perform standard operations (start/stop streaming, set/get modes, etc.)
2. Providing two output nodes: Bayer image data on the `/dev/video0` node and sensor embedded data on the `/dev/video1` node.

After getting a bayer frame buffer from the *Unicam* driver, we pass it to our ISP driver. Again, this driver follows the standard V4L2 API. This driver is responsible for:

1. Exposing a set of V4L2 "controls" used to configure the pipeline blocks within the ISP hardware.
2. Taking the bayer input framebuffer, together with an appropriate set of output buffers and configuration, and generating an output in RGB or YUV format.
3. Generating some statistics used by our Control Algorithms (more on this later).
4. Providing one input node on the `/dev/video13` for the bayer input framebuffer.
5. Providing three output nodes: Two image outputs on the `/dev/video14` and `/dev/video15` nodes, and statistics output on the `/dev/video16` node.

3.1 Provisions for the Sensor Subdevice

Embedded Data

In most cases, sensors transmit embedded data together with each image frame. This embedded data tells us what settings (mainly gain and shutter speed) were used to expose the frame. This is crucially important for our AGC algorithm. In order for the Unicam driver to correctly capture this data into a buffer, the sensor subdevice must advertise that embedded buffer is available. To do this, the driver must register 2 pads with the Media Controller framework (see https://linuxtv.org/downloads/v4l-dvb-apis/kapi/mc-core.html):

```
  u16 num_pads = 2;
  struct media_entity entity;
  struct media_pad pads[2];
  media_entity_pads_init(&entity, num_pads, pads);
```

The driver must also advertise the format and size of the embedded buffer in a way similar to the following code snippet:

```
static int sensor_enum_mbus_code(struct v4l2_subdev *sd,
                                 struct v4l2_subdev_pad_config *cfg,
                                 struct v4l2_subdev_mbus_code_enum *code)
{
  if (code->pad == 0) {
    /* Pad 0 is the image data pad */
    code->code = MEDIA_BUS_FMT_SRGGB10_1X10
  } else if (code->pad == 1) {
    /* Pad 1 is the embedded data pad */
    code->code = MEDIA_BUS_FMT_CSI2_EMBEDDED_DATA;
  } else
    return -EINVAL;

  return 0;
}

static int sensor_get_pad_format(struct v4l2_subdev *sd,
                                 struct v4l2_subdev_pad_config *cfg,
                                 struct v4l2_subdev_format *fmt)
{
```

```
  if (fmt->pad == 0) {
    sensor_get_image_format(cfg, fmt);
  } else {
    /* Two lines of embedded data with a max line of 8192 pixels. */
    fmt->format.width = 8192;
    fmt->format.height = 2;
    fmt->format.code = MEDIA_BUS_FMT_CSI2_EMBEDDED_DATA;
  }

  return 0;
}
```

A full implementation can be found in the **imx219** kernel driver (drivers/media/i2c/imx219.c).

When there is no embedded data available (as is the case with the OV5647 sensor), the Raspberry Pi pipeline handler will see that no embedded data node is advertised, and instead will resort to counting frames. It will assume a fixed number of frames of delay between values being requested and the sensor applying them. Whilst theoretically not as reliable, in practice this strategy has been found to work adequately.

NOTE: There is no official support in V4L2 for sensor embedded data. Using a media controller pad to advertise embedded data as described above is yet to be approved by the framework maintainers and will likely be changing in the near future.

V4L2 Controls and IOCTLs

Sensor subdevices must expose the following V4L2 "controls" if they are to work correctly with the Raspberry Pi pipeline in *libcamera*:

- V4L2_CID_EXPOSURE for setting exposure time in lines.
- V4L2_CID_ANALOGUE_GAIN for setting analogue gain.
- V4L2_CID_VBLANK for setting vblanking (effectively frame length) in lines.
- V4L2_CID_HFLIP and V4L2_CID_VFLIP for setting image orientation.
- V4L2_CID_PIXEL_RATE for getting the pixel clock rate in the sensor. This is used to calculate exposure time and vblanking in lines.

For further details on these controls, see https://www.linuxtv.org/downloads/v4l-dvb-apis-new/userspace-api/v4l/ext-ctrls-image-source.html and https://www.linuxtv.org/downloads/v4l-dvb-apis-new/userspace-api/v4l/control.html.

Additionally, all sensor subdevices must support the VIDIOC_SUBDEV_G_SELECTION ioctl to return the sensor mode crop region. See https://www.linuxtv.org/downloads/v4l-dvb-apis-new/userspace-api/v4l/vidioc-subdev-g-selection.html for further details.

Gain and Exposure Staggered Writes

The sensor subdevice driver registers V4L2_CID_EXPOSURE and V4L2_CID_ANALOGUE_GAIN controls for userspace to use to update the sensor's exposure and analogue gain settings.

Because it often takes a different number of frames for these settings to take effect (depending on the camera), they must be written to the sensor in an appropriately staggered manner. This *staggering* of the values is taken care of in the Raspberry Pi pipeline handler. For example, it might typically take 2 frames for exposure changes to take effect, but only 1 frame for analogue gain changes. In this case, as soon as frame N has started, we will write the analogue gain for frame $N + 1$, along with the exposure value for frame $N + 2$.

Start of Frame Events

We have seen that the Raspberry Pi pipeline handler running in userspace is sensitive to being able to count frames reliably, and has to be able to update values, possibly in this staggered manner, before another frame starts. To make this work reliably, the Unicam driver signals `V4L2_EVENT_FRAME_SYNC` events up to userspace whenever a new frame starts arriving from the sensor. This gives userspace code the maximum possible time to send updated values for future frames.

3.2 The `CamHelper` Class

Unfortunately there is a certain amount of information that we need to know about cameras and camera modes but which the kernel driver framework (V4L2) does not supply. For this reason we provide the `CamHelper` class, along with derived class implementations for all of Raspberry Pi's supported sensors. These are all device-specific functions, but implemented in userspace rather than by extending the existing kernel framework. The principal functions of this class are:

1. To convert exposure times (in microseconds) to and from the device-specific representation (usually a number of lines of pixels) used by V4L2.
2. To convert analogue gain values to and from the device-specific "gain codes" that must be supplied to V4L2.
3. To allow parsing of embedded data buffers returned by a sensor.
4. To indicate how many frames of delay there are when updating the sensor's exposure time or analogue gain values.
5. To indicate how many frames are invalid and may need to be dropped on startup, or after changing the sensor mode and re-starting streaming.
6. To indicate the physical orientation of the sensor, that is to say, whether it is mounted upside-down on the board or not.

Generally speaking it should be sufficient to copy one of our existing implementations and work from there. In particular note that

- Sensors that do not supply embedded data buffers should copy the OV5647 implementation (`cam_helper_ov5647.cpp`). In these circumstances we are passed dummy metadata buffers which need to be parsed by the `MdParserRPi` class.
- Sensors that do supply embedded data buffers should copy the imx477 version (`cam_helper_imx477.cpp`). If the sensor in question follows the general SMIA pattern then this implementation, and the metadata parser, should in fact be quite close to what is required.

The `CamHelper` class methods are well documented in `cam_helper.hpp`, however, we do further draw the reader's attention to the `CameraMode` class, which contains the important parameters related to each different camera mode. The fields of the `CameraMode` are populated by the IPA, so they should automatically contain the correct values once the kernel driver is functioning properly. These fields documented in `controller/camera_mode.h`, but we elaborate on them here.

Field	Description
`name`	A human readable name for the camera mode.
`bitdepth`	The number of bits delivered by the sensor for each pixel. For most sensors this might be 8, 10 or 12.
`width`	The width of the frame, in pixels, being delivered by the sensor.
`height`	The height of the frame, in pixels, being delivered by the sensor.
`sensor_width`	This is the width of the highest resolution frame that could be delivered by the sensor.
`sensor_height`	The height of the highest resolution frame that could be delivered by the sensor.

bin_x	The amount of pixel binning in the horizontal direction, that is, the number of pixels that are averaged together to form one output pixel. When there is no binning this has the value 1; for 2x2 binning, this has the value 2.
bin_y	As bin_x, but in the vertical direction.
crop_x	The horizontal offset, measured in the *full resolution image*, of the top left pixel being read out. This is therefore in the same units as sensor_width.
crop_y	The vertical offset, measured in the *full resolution image*, of the top left pixel being read out. This is therefore in the same units as sensor_height.
scale_x	The relative scaling of each output pixel to each full resolution pixel. So when there is no cropping at all, we should have that $width * scale\_x = sensor\_width$. Consequently, a 2x2 binning mode should have the value 2 here. Any right-hand crop would be given by $sensor\_width - crop\_x - scale\_x * width$.
scale_y	The relative scaling of each output row to each full resolution pixel row. So when there is no cropping at all, we should have that $height * scale\_y = sensor\_height$. Consequently, a 2x2 binning mode should have the value 2 here. Any bottom edge crop would be given by $sensor\_height - crop\_y - scale\_y * height$.
noise_factor	The ratio of the standard deviation of the pixel noise compared to the full resolution mode. So for example, in a 2x2 binning mode, 4 pixels are averaged for each output pixel. Therefore the value here should be $\sqrt{4} = 2$.
line_length	The time to deliver a single row of pixels, in nanoseconds.
min_frame_length	The minimum allowable frame length for this mode.
max_frame_length	The maximum allowable frame length for this mode.

Table 2: Fields of the CameraMode class.

3.3 Device Tree Considerations

The camera driver needs to be listed as part of the Linux device tree so that it is correctly loaded. The details are largely beyond the scope of this document, however, we draw the reader's attention to the following resources.

1. In the raspberrypi/linux Github repository (https://github.com/raspberrypi/linux), please consult the Documentation/devicetree/bindings/media/i2c directory for example bindings, such as imx219.txt or ov5647.txt.
2. The existing overlays in the linux source tree at arch/arm/boot/dts/overlays. For example, look for imx219-overlay.dts or ov5647-overlay.dts.
3. There is existing documentation at the following link https://www.raspberrypi.org/documentation/configuration/device-tree.md.

4 Control Algorithm Overview

4.1 Framework

Whilst Raspberry Pi provides a complete implementation of all the necessary ISP control algorithms, these actually form part of a larger framework. The principal components of this framework are described as follows.

- The `Controller` class is a relatively thin layer which manages all the control algorithms running under its *aegis*. It is responsible for loading the control algorithms at startup and invoking their methods when instructed to through *libcamera*.
- The `Algorithm` class is specialised to implement each individual control algorithm. The `Controller` invokes the methods of each algorithm as instructed.

The principal methods of the `Algorithm` class are

1. The `Prepare` method which is invoked just before the ISP is started. It is given information about the captured image (including shutter speed and amount of analogue gain applied by the sensor) and must decide the actual values to be programmed into the ISP.
2. The `Process` method which is invoked just after the ISP has run. This method is given a buffer containing the various statistics the ISP has just collected, and is expected to initiate new computations which will determine the values written by the `Prepare` method when it runs again.

Other methods include the `Read` method, which loads the algorithm and may also initialise default values, and the `SwitchMode` method, which is invoked whenever a new `CameraMode` is selected for use.

In the following discussion, pathnames of specific files are given relative to the location of the Raspberry Pi specific code in the *libcamera* source tree, namely `libcamera/src/ipa/raspberrypi`, unless we explicitly state otherwise.

4.2 Defining and Loading Algorithms

At startup, the `Controller` loads and initialises those algorithms that are listed in a JSON file. The idea is that a separate JSON file (the *camera tuning*) is loaded for each different kind of camera. Every algorithm is registered with the system using a static `RegisterAlgorithm` initialiser, making it easy to add more algorithms to the system.

The JSON file lists camera algorithms by name, conventionally given in a *<vendor>.<moniker>* format. The *moniker* can be thought of as the *type* of the algorithm, such as AWB (Automatic White Balance) or AGC/AEC (Automatic Gain/Exposure Control). For instance, Raspberry Pi's implementation of AWB is named `rpi.awb`. The implementation of each algorithm defines what parameters it has and in its `Read` method how it parses them from the file or sets default values.

If, for example, the mythical *Foo Corporation* were to produce their own implementation of AWB, it might be named `foo.awb`, and it could be listed directly in the JSON camera tuning file. Any algorithms no longer wanted could be deleted from the JSON file, or commented out by renaming them. For instance, changing `rpi.awb` to `x.rpi.awb` will cause the Raspberry Pi version to be silently ignored. We encourage developers to use a separate folder for any algorithms they implement, so while Raspberry Pi's algorithms all live in `controller/rpi`, Foo Corporation's algorithms might live in `controller/foo`.

Finally, we note that the order in which algorithms are listed in the JSON file is the order in which the controller will load and initialise them, and is also the order in which the `Prepare` and `Process` methods of the algorithms will run.

4.3 Standard Algorithms

The framework defines a number of abstract classes derived from `Algorithm` that provide standard interfaces to particular kinds of algorithms. For example, the `class AwbAlgorithm` (`controller/awb_algorithm.hpp`) requires the implementation of a `SetMode` method which allows applications to set the AWB mode to, for example, `"tungsten"`, `"daylight"`, and so forth. Developers are encouraged to use these standard algorithm types when this is possible.

Furthermore, implementations of these standard algorithms are expected, by convention, to have a name which ends with the appropriate moniker for that type of algorithm. So an AWB algorithm would have a name always ending in `awb`, and AGC/AEC (Automatic Gain/Exposure Control) would always end in `agc`, facilitating the type of code shown below.

```
Algorithm *algorithm = controller->GetAlgorithm("awb");
AwbAgorithm *awb_algorithm = dynamic_cast<AwbAlgorithm *>(algorithm);
if (awb_algorithm) {
        awb_algorithm->SetMode("sunny");
}
```

The full set of standard monikers for our different types of algorithm is set out in the table below, though there is no reason in principle why developers should not invent new ones if they are appropriate for their own use cases. The algorithms themselves are explained in detail in the following chapter.

Moniker	Description
`alsc`	Automatic Lens Shading Correction algorithm.
`agc`	Automatic Gain/Exposure Control algorithm (AGC/AEC).
`awb`	Automatic White Balance algorithm (AWB).
`black_level`	Algorithm to set the correct black level for the sensor.
`ccm`	Algorithm to calculate the correct Colour Correction Matrix (CCM).
`contrast`	Adaptive global contrast and gamma control algorithm.
`dpc`	Algorithm to set the appropriate level of Defective Pixel Correction.
`geq`	Algorithm to set an appropriate level of Green Equalisation.
`lux`	Algorithm that estimates an approximate lux level for the scene.
`noise`	Algorithm to calculate the noise profile of the image given the current conditions.
`sdn`	Algorithm to set an appropriate strength for the Spatial Denoise block.
`sharpen`	Algorithm to set appropriate sharpness parameters.

Table 3: Algorithm type monikers.

4.4 Algorithm Communication

Algorithms communicate with the outside world, and with each other, through *image metadata*. Image metadata is an instance of the `Metadata` class (`controller/metadata.hpp`) which is created for each new image from the camera, and travels alongside the image through the entire processing pipeline; it is available to both the `Prepare` and `Process` methods. Code external to the algorithms themselves will look in the image metadata to find updated values to program into the ISP, and some algorithms may also look in the metadata to find the results calculated by other algorithms that they wish to use.

This information written into the image metadata is normally referred to as the algorithm's *status*, and is implemented by a class whose name ends in `Status`, normally found in the files `controller/*_status.h`. For example, the CCM algorithm (which calculates the colour correction matrix for the ISP) relies on the estimated colour temperature which can be found in the `AwbStatus` object (`controller/awb_status.h`) in the image

metadata. In particular, items are stored in the image metadata indexed by a string key, and by convention this key is made up of two parts joined by a period (.), the first being the algorithm moniker (awb) and the second the word status.

Furthermore, we normally arrange that the status metadata contains any user-programmable settings that affect the result of the algorithm. This is so that an application can request that a setting be changed, and then sit back and monitor the image metadata to discover when it has actually happened. For example, we have seen the SetMode method of the AWB algorithm. When the AwbStatus metadata shows the new mode name in its mode field, then we know the change has taken effect.

When replacing existing algorithms, care must be taken still to generate the correct status information in the image's metadata, as other (un-replaced) algorithms may still expect it. So any replacement AWB algorithm would still have to write out an AwbStatus object for the CCM algorithm (under the key awb.status), otherwise the CCM algorithm would need amending or replacing also. Custom algorithms are free to communicate using the image metadata too, only in this case the expectation is that any metadata keys would begin with the organisation name. To wit, our friends at Foo Corporation might file their secret AWB parameters under foo.awb.secret_parameters.

Recall that the order in which Prepare and Process methods are run matches that of the JSON file, meaning that when one algorithm has a dependency on the status information of another then that determines their relative ordering within the file. So in the case of our example, we will have to list rpi.ccm **after** rpi.awb.

4.5 Performance Considerations

When writing new control algorithms, care must be taken not to delay the operation of the imaging pipeline any longer than necessary. This means that both the Prepare and Process methods for any algorithm must take considerably less than a millisecond, and that any significant amount of computation must be backgrounded using an asynchronous thread. Amongst the Raspberry Pi algorithms, examples of this can be found in the AWB (controller/rpi/awb.cpp) and ALSC (Automatic Lens Shading Correction - controller/rpi/alsc.cpp) algorithms.

Although the details are of course up to developers, the general pattern is for the Process method to re-start the computation (in the asynchronous thread) if the previous computation has finished. Often it makes sense to wait a few frames before re-starting it as the results are not usually so critical that we need them immediately, and we can thereby save on CPU load too.

The Prepare method normally has a notion of *target values*, and it filters the values that it outputs slowly towards these target values with every frame (for example, using an IIR filter). It watches for any in-progress asynchronous computation to finish though it does not block waiting for it. If unfinished, it will simply leave the target values unchanged and try again on the next frame; if finished, this merely causes an update to its target values, which it will immediately start filtering towards.

4.6 Example Camera Tuning File

```
{
    "rpi.black_level":
    {
        "black_level": 4096
    },
    "rpi.awb":
    {
        "bayes": 0
    },
    "rpi.agc":
    {
        "metering_modes":
        {
            "centre-weighted": {
                "weights": [4, 4, 4, 2, 2, 2, 2, 1, 1, 1, 1, 0, 0, 0, 0]
            }
        },
        "exposure_modes":
```

```
        {
            "normal":
            {
                "shutter": [ 100, 15000, 30000, 60000, 120000 ],
                "gain":    [ 1.0, 2.0,   3.0,   4.0,   6.0    ]
            }
        },
        "constraint_modes":
        {
            "normal":
            [
            ]
        },
        "y_target": [ 0, 0.16, 1000, 0.165, 10000, 0.17 ]
    },
    "rpi.ccm":
    {
        "ccms":
        [
            { "ct": 4000, "ccm": [ 2.0, -1.0, 0.0, -0.5, 2.0, -0.5, 0, -1.0,
    2.0 ] }
        ]
    },
    "rpi.contrast":
    {
        "ce_enable": 0,
        "gamma_curve": [
            0,     0,
            1024,  5040,
            2048,  9338,
            3072,  12356,
            4096,  15312,
            5120,  18051,
            6144,  20790,
            7168,  23193,
            8192,  25744,
            10240, 30035,
            11264, 32005,
            12288, 33975,
            13312, 35815,
            14336, 37600,
            15360, 39168,
            16384, 40642,
            18432, 43379,
            20480, 45749,
            22528, 47753,
            24576, 49621,
            26624, 51253,
            28672, 52698,
            30720, 53796,
            32768, 54876,
            36864, 57012,
            40960, 58656,
            45056, 59954,
            49152, 61183,
            53248, 62355,
            57344, 63419,
            61440, 64476,
            65535, 65535
        ]
    }
}
```

The contents of the above JSON file are not particularly important for the moment, however we note the use of named Raspberry Pi (`rpi`) algorithms. This example file is deliberately very simple, but contains enough information to give quite recognisable images from almost any sensor.

4.7 Directory Structure

A summary of the folders and files provided as part of the Raspberry Pi framework is given below.

Path	Description
`cam_helper*`	Implementation of the `CamHelper` class for different sensors.
`md_parser*`	Functionality pertaining to the parsing of camera embedded data (see chapter 3).
`controller/*_status.h`	Definitions of the status objects placed into the image metadata by control algorithms.
`controller/*_algorithm.hpp`	Interfaces for standard algorithms, such as AWB, AGC/AEC etc.
`controller/metadata.hpp`	Class representing image metadata.
`controller/histogram.cpp`	A simple histogram class that may be useful to control algorithm implementations.
`controller/pwl.cpp`	A simple implementation of piecewise linear functions that may be useful to control algorithms. Many of Raspberry Pi's own control algorithms depend on these.
`controller/rpi`	Folder containing all of Raspberry Pi's algorithm implementations.
`data`	Folder containing existing camera tuning JSON files.
`utils/raspberrypi/ctt`	The Camera Tuning Tool (see chapter 6). Note that this path lies under the top-level *libcamera* directory.

Table 4: Raspberry Pi controller folder structure.

5 Raspberry Pi Control Algorithms

This section documents Raspberry Pi's implementation of the ISP and camera control algorithms. For each algorithm we give an overview and if necessary explain any concepts underlying the algorithm. We list all of the tuning parameters for each algorithm including what they mean and their default values, and explicitly describe any metadata dependencies. Note further that when any of these algorithms runs, the image metadata already contains the `DeviceStatus` metadata (`controller/device_status.h`), giving the shutter speed and analaogue gain of the current image.

The algorithms are listed in the order in which they would normally appear in the JSON file as this is probably the most natural order in which to understand them. Unlike some other camera systems there is no AF (autofocus) algorithm as the Raspberry Pi does not at this time support any camera modules that require it.

Many of the parameters listed here can be left at default values, however, others may appear distinctly esoteric. Users are reminded that a Camera Tuning Tool (CTT) is provided (the subject of chapter 6) which facilitates the automatic generation of JSON tuning files with all these difficult values correctly calibrated.

5.1 Black Level

5.1.1 Name

`rpi.black_level`

5.1.2 Overview

The pixels output by the camera normally include a "pedestal" value or *black level*. Pixels at or below this level should be considered black. For a camera outputting 10-bit pixel values (in the range 0 to 1023) a typical black level might be 64, but the correct value would be found in the sensor manaufacturer's datasheet. The number entered into the tuning file here is agnostic of the sensor's native bit depth and always expects a value from a 16-bit dynamic range. So in this case, we would supply the value 4096 ($= 64/1024 \times 65536$).

Name	Default	Description
`black_level`	4096	Specifies a default black level value to use for all three RGB channels, unless over-ridden by subsequent entries.
`black_level_r`	*Optional*	If specified, over-rides the default `black_level` for the red channel.
`black_level_g`	*Optional*	If specified, over-rides the default `black_level` for the green channel.
`black_level_b`	*Optional*	If specified, over-rides the default `black_level` for the blue channel.

Table 5: Black level algorithm parameters.

5.1.3 Metadata Dependencies

The Black Level control algorithm has no dependency on any other image metadata.

The Black Level control algorithm writes a `BlackLevelStatus` object (`controller/black_level_status.h`) into the image metadata. This gives a (16-bit) black level value for each of the three RGB channels.

5.2 Defective Pixel Correction (DPC)

5.2.1 Name

`rpi.dpc`

5.2.2 Overview

Camera images may contain a number of "stuck" or sometimes "weak" pixels. Pixels in the former category are stuck at a particular value, sometimes completely black or sometimes completely white, whereas the latter kind (weak pixels) vary atypically according to the incoming light and can be brighter or darker than their neighbours. In both cases this can lead to undesirable image artifacts.

Figure 4: Magnified image showing a bright defective pixel - its effect spreads slightly because of the filtering operations performed using it.

The ISP hardware has the ability to filter some of these out adpatively, without knowing in advance where they are. We expose a simple control for this, namely *no correction*, *normal correction* or *strong correction*.

5.2.3 Parameters

Name	Default	Description
`strength`	1	Strength of the correction. 0 = no correction, 1 = "normal" correction and 2 = "strong" correction.

Table 6: Defective pixel correction algorithm parameters.

5.2.4 Metadata Dependencies

The DPC control algorithm has no dependency on any other image metadata.

The DPC control algorithm writes a `DpcStatus` object (`controller/dpc_status.h`) into the image metadata.

5.3 Lux

5.3.1 Name

`rpi.lux`

5.3.2 Overview

The Lux algorithm does not actually generate any parameters that are subsequently programmed into the ISP. Instead, it merely provides a single location where an estimate of the lux level (amount of light) of the current image can be generated. It writes this value into the image metadata where downstream control algorithms can find it and use it.

It is set up with the exposure/gain parameters and known lux level of a reference image, which must be calibrated during the camera tuning process (most straightfowardly with the CTT). It generates its lux estimate for the current image by a simple ratio calculation comparing current exposure/gain and statistics values to the ones recorded for the reference image.

5.3.3 Parameters

Name	Default	Description
`reference_shutter_speed`	*Required*	Shutter speed (exposure time) of the reference image, in microseconds, e.g. `33000`.
`reference_gain`	*Required*	Analogue gain of the reference image, e.g. `1.5`.
`reference_Y`	*Required*	Measured Y (luminance) value of the reference image. This should normally be obtained through the CTT.
`reference_lux`	*Required*	Estimated lux level of the reference image scene. Normally supplied by the CTT.

Table 7: Lux algorithm parameters.

5.3.4 Metadata Dependencies

The Lux control algorithm requires the presence of the `DeviceStatus` metadata (which should automatically always be available).

The Lux control algorithm writes a `LuxStatus` object (`controller/lux_status.h`) into the image metadata.

5.4 Noise

5.4.1 Name

`rpi.noise`

5.4.2 Overview

The Noise algorithm also does not calculate any new parameters for the ISP, it merely calculates the *noise profile* for the current image. It does this by being calibrated with the noise profile of a reference image and then, for each image passing through the pipeline, adjusts the noise profile according to the analogue gain of the current image and also information about the *camera mode*. The camera mode is significant because most cameras are able to average pixels together, outputting a less noisy image than the native full resolution image. These values would normally be calculated by the CTT (Camera Tuning Tool).

5.4.3 Parameters

Name	Default	Description
`reference_constant`	*Required*	Constant offset of the noise profile. In practice, it can be taken always to have the value zero.
`reference_slope`	*Required*	Slope of the noise profile, plotted against the square root of the pixel value. Supplied by the CTT, a typical value might be somewhere around `3.0`.

Table 8: Noise algorithm parameters.

5.4.4 Metadata Dependencies

The Noise control algorithm requires the presence of the `DeviceStatus` metadata (which should automatically always be available).

The Noise control algorithm writes a `NoiseStatus` object (`controller/noise_status.h`) into the image metadata.

5.5 GEQ (Green Equalisation)

5.5.1 Name

`rpi.geq`

5.5.2 Overview

Standard Bayer cameras produce raw images with two slightly different kinds of green pixels, those on the red rows, and those on the blue rows. Under some circumstances these can respond slightly differently even when exposed to a solid and completely uniform colour. Downstream processing (especially *demosiacking*) can have a tendency to pick up and accentuate these differences, giving rise to what are frequently described as *maze artifacts*.

Green Equalisation is the process of correcting green imbalance, shown in the image above. Most simply, it is applying a low pass filter to the green pixels, but it needs to be given some expectation of the size of the imbalance so as not to blur the image more than is necessary. The Green Equalisation parameters would normally be provided by the CTT.

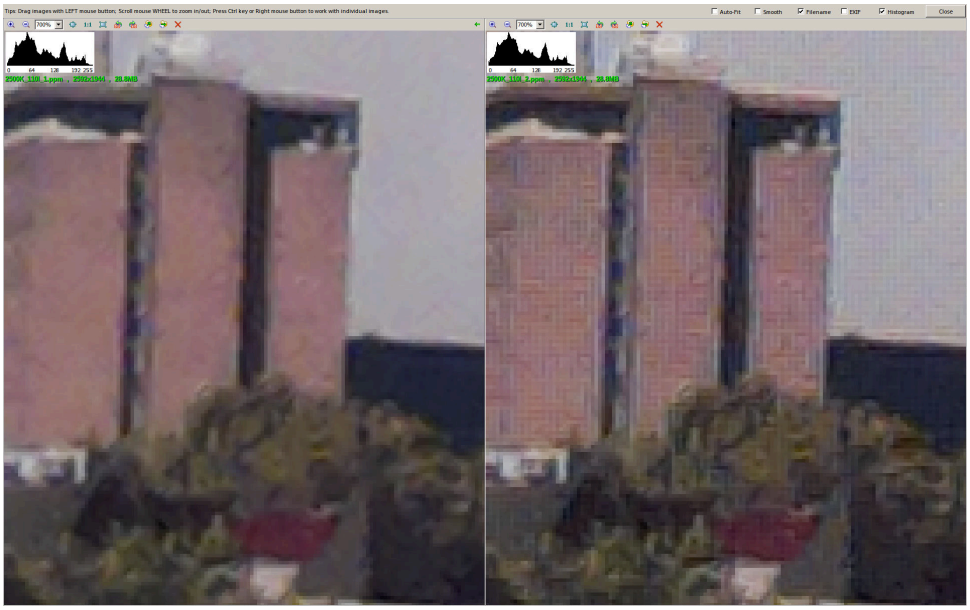

Figure 5: Highly magnified image showing maze artifacts (right), and correction (left).

5.5.3 Parameters

Name	Default	Description
`offset`	0	Constant offset of the green imbalance level.
`slope`	0.0	Slope of the green imbalance level (compared to pixel value). Must be < 1.0.
`strength`	*Optional*	If specified, a piecewise linear function that applies a gain to the green imbalance level based on estimated lux level. This allows even stronger green imbalance correction in very low light situations.

Table 9: Green equalisation algorithm parameters.

5.5.4 Metadata Dependencies

The Green Equalisation algorithm requires the presence of the `DeviceStatus` metadata (which should automatically always be available). It also requires the presence of `LuxStatus` metadata from the Lux control algorithm.

The Green Equalisation algorithm writes a `GeqStatus` object (`controller/geq_status.h`) into the image metadata.

5.6 SDN (Spatial Denoise)

5.6.1 Name

`rpi.sdn`

5.6.2 Overview

The SDN (Spatial Denoise) control algorithm generates parameters for controlling the ISP's spatial denoise function. It derives a new noise profile from the one calculated (and placed in the image metadata) by the Noise control algorithm, and this is ultimately passed to the ISP. It also determines the "strength" of the denoise function, meaning how much of the final pixel is taken from the denoised pixel, the remainder being made up by the un-denoised pixel. This turns out to be helpful because denoising an image completely can leave it looking a little artificial or "plasticky"; leaving some noise in the final image normally proves to be aesthetically beneficial.

The SDN control algorithm parameters can normally just be left at their default values.

Figure 6: Same image as figure 5, highly magnified, showing "noise speckles".

Colour channel denoising is another operation controlled by the SDN algorithm. Colour denoising can operate in two modes - a high quality mode, or a fast mode. The former should be used for still image captures where frame time is not critical, while the latter should be used for viewfinder and video recoding where processing time is more important to avoid dropping frames. No colour denoising is enabled by default.

5.6.3 External API

The `Sdn` class is derived from the `DenoiseAlgorithm` class. As such it defines the following publicly accessible methods.

DenoiseAlgorithm class API
`SetMode(DenoiseMode mode)` Sets the operating mode for SDN. `DenoiseMode` may be one of the following enum values {`Off`, `ColourOff`, `ColourFast`, `ColourHighQuality`}.

Table 10: Denoise algorithm public API.

An example use of the DenoiseAlgorithm method:

```
Algorithm *algorithm = controller->GetAlgorithm("sdn");
DenoiseAgorithm *deniose_algorithm = dynamic_cast<AgcAlgorithm *>(algorithm);
if (denoise_algorithm) {
    denoise_algorithm->SetMode(DenoiseMode::ColourHighQuality)
}
```

5.6.4 Parameters

Name	Default	Description
deviation	3.2	Scaling applied to the Noise control algorithm's noise profile. Increasing this means wider pixel differences will be denoised (producing a blurrier image) and reducing it results in only smaller pixel differences being denoised. Making this too small will produce an image with little "noise speckles" in it.
strength	0.75	Proportion of the final output pixel made up by the denoised pixel value, the remainder (0.25 in the default case) coming from the original (noisy) pixel.

Table 11: Spatial denoise algorithm parameters.

5.6.5 Metadata Dependencies

The SDN control algorithm requires the presence of the NoiseStatus metadata from the Noise control algorithm.

The SDN control algorithm writes a DenoiseStatus object (controller/denoise_status.h) into the image metadata.

5.7 AWB (Automatic White Balance)

5.7.1 Name

rpi.awb

5.7.2 Overview

Readers are no doubt aware how, when you look at a particular (non-changing) object, its colour always appears the same, irrespective of whether you look at it outside in broad daylight or indoors under some kind of artificial light. If it looks *orange* outdoors, then it looks *orange* indoors too. Yet if you measured the colour directly using a spectrometer, you'd discover that *outdoors* the object is much *bluer* and *indoors* it is much *redder*.

The Human Visual System has adapted amazingly over time to hide these differences from us - a phenomenon known as *Colour Constancy*. Unfortunately no such magic occurs when viewing a digital image so we have to make this correction ourselves. This is the job of the AWB (Automatic White Balance) Algorithm. We describe the algorithm below, though readers should note that the settings and calibrations should normally be produced by the Camera Tuning Tool.

Raspberry Pi takes a simple Bayesian approach to the problem. We need to define a *feasible* set of illuminants and then, for any given image, choose that illuminant from the set which gives us the *most likely* resulting colours. The mathematically inclined will recognise this as a *maximum likelihood estimation* problem. Finally we apply the correction to the image according to the selected illuminant.

The Feasible Set

The illuminants that we encounter in the real world have historically tended to lie on the so-called *Planckian Locus*, as the light is produced by some kind of heat-radiating body (be it the sun, or a tungsten filament). In recent years, with the advent of fluorescent, LED and other light sources, this is less true, however, with some slight loosening of our constraints it can still be considered an adequate approximation.

The feasible set is constructed by measuring the response of the sensor to a known grey target. In practice this means capturing an image of something like a Macbeth Chart (with objectively grey squares) under each of our possible illuminants, and measuring the average RGB value of the grey squares. For each RGB triple we calculate normalised colour values $r = R/G$ and $b = B/G$, and plot these on a graph called the *Colour Temperature Curve* (or *CT Curve*) for the camera. Each point on the curve can be associated with the colour temperature of the illuminant that provided it.

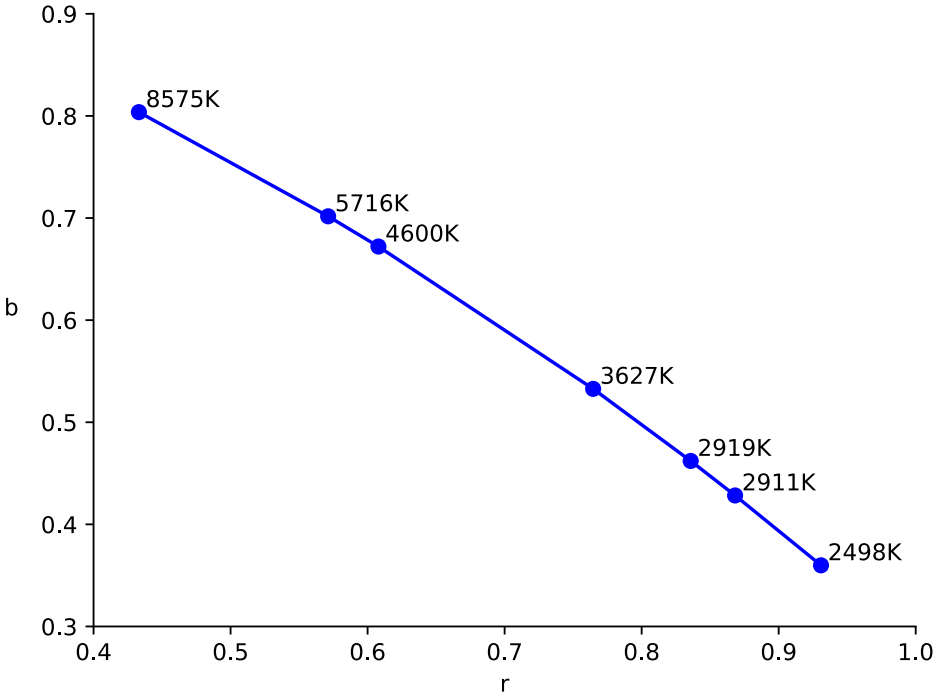

Figure 7: CT Curve showing colour temperature associated with each point.

Note how, in this example, colour temperatures get closer at the left hand (daylight) end of the curve, and spread out at the right hand (indoor) end. In particular, the points for colour temperatures 2911K and 2919K are quite distinct even though the colour temperatures are virtually identical. This is most likely because they were produced by different kinds of illuminant - perhaps one was a fluorescent tube and the other an incandescent bulb.

Finally, whilst the (r, b) coordinate gives the response of the camera to light of that colour temperature, the red and blue gains that need to be applied to the image in order to white balance it correctly (that is, make actual grey have an equal response on all RGB channels) are

$$gain_r = \frac{1}{r}$$

and

$$gain_b = \frac{1}{b}$$

Bayes Theorem

In our case we are wishing to find the illuminant I with the highest probability given the observed pixel data D, written as $P(I \mid D)$. According to Bayes' Theorem we have

$$P(I \mid D) = \frac{P(D \mid I) \cdot P(I)}{P(D)}$$

where

$P(I)$ is the prior probability of the illuminant I,
$P(D)$ is the prior probability of the observed pixel data,
$P(D \mid I)$ is the posterior probability of the observed pixel data D given the illuminant I, and finally
$P(I \mid D)$ is the posterior probability of the illuminant I given the observed pixel data D.

The Priors

Firstly, in our case we take $P(D)$ as constant, therefore the problem reduces to finding

$$\arg\max_{I} P(D \mid I) \cdot P(I)$$

It is up to us how we wish to define $P(I)$, and indeed this is one of the principal means of tuning the AWB algorithm. Typically, we might choose larger (more likely) values for daytime illuminants when the ambient conditions (as measured by the Lux control algorithm) suggest the scene is very bright. Accordingly, the indoor illuminants would be dispreferred under these conditions. In medium to low light situations we would most likely reverse these biases, favouring the indoor illuminants instead.

Calculating the Posteriors

$P(D \mid I)$ is not difficult to calculate providing we have some kind of probabilistic model of the colours we expect to see in an output image. In our case we regard certain colours as more likely than others. In particular the most likely colour is generally grey or near-grey (the correspondance with simple "Grey World" algorithms should be clear), but we could consider others too - blue sky, green grass and skin tones might be the most likely candidates. (Note, however, that Raspberry Pi's implementation only looks for near-grey colours at this time.) The procedure is then to take the raw camera image, apply the red and blue gains for a particular illuminant, and then assess how plausible the result looks.

We illustrate this below with an indoor image taken under incandascent lighting; note that the raw camera image (as is often the case) looks rather green.

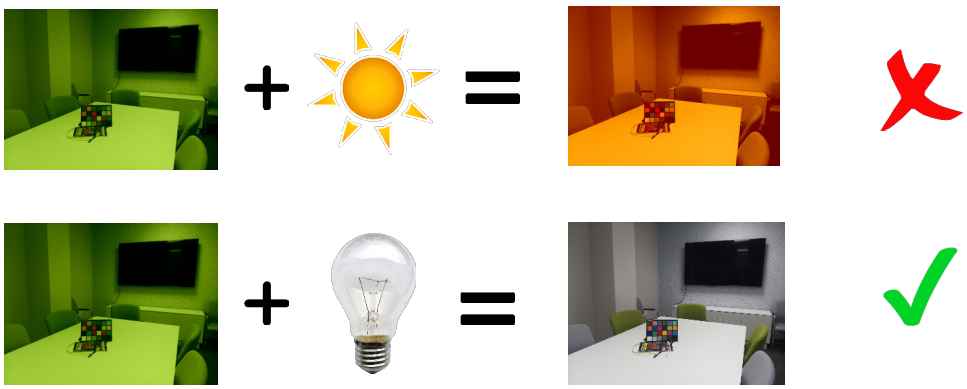

Figure 8: AWB example - the incandascent illuminant gives a more plausible result than the daylight one.

We also need to specify how we calculate the probability of any pixel's colour. Given the pixel's RGB colour values we calculate, as before, $r = R/G$ and $b = B/G$. For a given illuminant I (and hence known $gain_r$ and $gain_b$ values), we define colour difference statistics

$$\Delta r = gain_r \cdot r - 1$$

$$\Delta b = gain_b \cdot b - 1$$

Observe how, for the illuminant which turns the pixel grey we have that $\Delta r = \Delta b = 0$.

Now we assume that a 2-dimensional Gaussian distribution gives a reasonable approximation to the PDF (probability density function) of our variables Δr and Δb, meaning that for a single pixel we have

$$P(D \mid I) \propto e^{-(\frac{\Delta r^2}{\sigma_r^2} + \frac{\Delta b^2}{\sigma_b^2})}$$

where σ_r and σ_b are the standard deviations of the distribution, and we use $\propto$ in place of equality because we have elided the (unimportant) fixed constant (that makes the PDF integrate to unity).

Now, we do not actually work directly on pixels as this would be far too slow. Instead the image is divided into $16 \times 12 = 192$ regions for which the hardware ISP calculates R, G and B averages; in many ways it can be thought of as if we are operating on an image only 16×12 pixels in size. So extending our probability calculation to the entire image, and assuming that all the regions are independent (an assumption we shall return to later), we obtain the following expression for the posterior probability of the illuminant:

$$P(I \mid D) \propto P(I) \prod_{regions} e^{-(\frac{\Delta r^2}{\sigma_r^2} + \frac{\Delta b^2}{\sigma_b^2})}$$

Log Likelihood

At this point it should be clear that working in terms of *log likelihood* is going to be more convenient. The large product turns simply into a sum, and as we are defining $P(I)$ ourselves anyway we may as well just define its log likelihood instead. We use L in place of P to denote log likelihoods.

Further, we take $\sigma_r = \sigma_b$ as it actually makes little difference to the end results, and then can drop this term entirely as we can scale our $L(I)$ function (the prior log likelihood of the illuminant) to compensate. Finally we are left with finding

$$\arg\max_I L(I) - \sum_{regions} (\Delta r^2 + \Delta b^2)$$

5.7.3 Implementation

Some notes on the Raspberry Pi implementation of the algorithm described above. It can be found in `controller/rpi/awb.cpp`.

1. The prior for the illuminant depends only on estimated lux level. Normally this involves interpolating between the two priors either side of the current lux estimate.
2. The CT Curve definition in the tuning file makes 2 piecewise linear functions, giving us r and b in terms of the colour temperature T. Thus $r = r(T)$ and $b = b(T)$.
3. The algorithm starts with a coarse search taking relatively large steps down the CT curve.
4. The "white point" the algorithm is looking for, in $(\Delta r, \Delta b)$ space is normally given by $(0, 0)$. However the configuration parameters (below) do allow this to be moved in case, for example, a slightly warmer look is desired.
5. Subsequently there is a fine search around the area found. The fine search is allowed to wander transversely off the CT Curve (within limits) to better accommodate lamps that do not lie perfectly on the curve (often including fluorescent lights and so forth).
6. As mentioned previously, the algorithm only expects neutral colours at this time.

7. The squared terms in the log likelihood expression get clamped; this reflects the fact that there genuinely are non-neutral colours in the world, and once something looks non-neutral it is counterproductive to penalise it further just because it appears *very* non-neutral.
8. For when no camera calibration has yet been done, our implementation includes a very plain Grey World algorithm that gives adequate results in many circumstances. When the Grey World algorithm is used, the estimated colour temperature produced by the AWB algorithm is not meaningful.

5.7.4 External API

The `Awb` class is derived from `AwbAlgorithm`. As such it defines the following publicly accessible methods.

AwbAlgorithm class API
`GetConvergenceFrames()` This returns the number of frames the AWB algorithm recommends be dropped, while it converges, when the camera and the AWB algorithm are started from scratch.
`SetMode(std::string const &mode_name)` This method takes the name of the desired AWB mode (a string such as `"tungsten"` or `"daylight"`). The effect is to limit the range of the coarse search to the range listed alongside the mode (which must be preset) in the JSON file.
`SetManualGains(double manual_r, double manual_b)` Switches to using the provided manual red and blue gains. Set both values to zero to switch back to auto mode.

Table 12: AWB algorithm public API.

An example use of AWB methods:

```
Algorithm *algorithm = controller->GetAlgorithm("awb");
AwbAgorithm *awb_algorithm = dynamic_cast<AwbAlgorithm *>(algorithm);
if (awb_algorithm) {
    awb_algorithm->SetMode("sunny");
}
```

5.7.5 Parameters

Name	Default	Description
`bayes`	1	Whether to use the Bayesian algorithm, or the simplified Grey World algorithm. Note that if the CT Curve is not defined the implementation will default back to Grey World in any case.
`frame_period`	10	Only run the AWB calculation every time this many frames have elapsed.
`speed`	0.05	IIR filter speed determining how quickly the image adapts towards the most recent results of the AWB calculation. A speed of zero effectively disables AWB, and a value of 1 causes the AWB calculation results to be followed immediately with no damping.
`startup_frames`	10	At startup, run the AWB algorithm as often as possible (ignoring `frame_period`) for this many frames. During these frames the `speed` parameter is also treated as having the value 1. This speeds up initial convergence when a camera application is opened.
`convergence_frames`	3	When queried by the *GetConvergenceFrames* method, the algorithm will return this value as the number of frames to drop.

ct_curve	*Optional*	Defines the functions $r = r(T)$ and $b = b(T)$. The numbers in this list always appear in threes. The first value of the triple is the colour temperature (T), the second is the r value and the third is the b value. The triples should be listed in increasing order of colour temperature.
priors	*Optional*	A list of illuminant priors for different lux levels. For each lux level the prior is a list of numbers appearing in pairs, where the first is the colour temperature and the second is the value of the log likeihood. If omitted, the priors are simply assumed to be identically zero everywhere.
modes	*Required*	A list of AWB modes giving a name (a character string) and the "lo" and "hi" range in terms of colour temperature on the CT curve that must be searched.
min_pixels	16	Minimum number of pixels in a statistics region for that region to be used in the AWB calculations.
min_G	32	Minimum average green value in a statistics region (out of a 16-bit dynamic range) for the region to be used in the AWB calculations.
min_regions	10	Minimum number of usable statistics regions for the AWB algorithm to run (otherwise the target red and blue gains will be left unchanged).
delta_limit	0.15	Limit to the value of the squared colour error term in the log likeihood expression; stops very non-neutral colours being excessively penalised.
coarse_step	0.2	Governs the size of the steps taken in the coarse search phase of the algorithm.
transverse_pos	0.01	How far off the CT curve the fine search is allowed to go, in the direction of less green illuminants.
transverse_neg	0.01	How far off the CT curve the fine search is allowed to go, in the direction of more green illuminants.
sensitivity_r	1.0	Calibration of the red sensitivity of the sensor being used compared to the one that was used for AWB tuning. This value cannot be set by the tuning process and would need some form of measurement for every sensor individually. Normally it lies close enough to 1.0 to make little difference.
sensitivity_b	1.0	Calibration of the blue sensitivity of the sensor being used compared to the one that was used for AWB tuning. This value cannot be set by the tuning process and would need some form of measurement for every sensor individually. Normally it lies close enough to 1.0 to make little difference.
whitepoint_r	0.0	The whitepoint for the Δr coordinate.
whitepoint_b	0.0	The whitepoint for the Δb coordinate.

Table 13: AWB algorithm parameters.

5.7.6 Metadata Dependencies

The AWB control algorithm requires the presence of the LuxStatus metadata from the Lux control algorithm.

The AWB control algorithm writes an AwbStatus object (controller/awb_status.h) into the image metadata.

5.7.7 Extensions

There are many relatively minor improvements and tweaks we could make to the algorithm described here. However, we have predominantly chosen to keep the approach simple and avoided anything that would complicate either the algorithm itself or the tuning process it would require, even at the cost of some modestly improved performance. We discuss just a couple of such ideas here.

Uniform Colour

One possible weakness of the algorithm is that it could be influenced excessively by large objects of a uniform but non-grey colour. In Bayesian terms, if you have already seen something bright red in an image, then it's more likely for neighbouring parts of the image to have the same colour as it might just be the same object. This obviously strikes at that assumption of independence that we made eariler, and such ideas are not difficult to incorporate. When we calculate the sum of squares term in the log likelihood expression for a particular patch in the image (*viz.* $\Delta r^2 + \Delta b^2$) we could also calculate a similar term measuring the distance from (some of) the neighbours' colours, and then simply pick the smallest.

Of course, experimentation would be required with a reasonable corpus of test images to determine the precise details, not least the relative strength of this effect overall and how it might vary.

Non-grey Colours

Another fairly clear improvement might be to "look for" colours other than grey in an image. For instance, it might prove beneficial to look for things like blue sky, green grass and even skin tones. At one level this is not difficult either. We compute the error term as before, but also compute the distance from these other measured colours, and again take the smallest. However, this does end up complicating the algorithm and can make the tuning process more difficult. In particular:

- These additional target colours would need to be measured, probably in the colourspace of the camera, and this causes additional calibration/tuning problems. Or you could try to measure them in sRGB which might save remeasuring them all the time - though then the AWB algorithm will implicitly include a colourspace conversion, which is all starting to get a bit circular.
- These extra colours are not likely to be present in all situations. For example blue sky is unlikely to feature in low light images, and matching it erroneously could throw the whole algorithm off. So there would have to be additional *prior knowledge* for each target colour expressing its effect in the known conditions.
- Finally, we have ended up with quite a lot of extra effects, parameters and new kinds of *priors* to calibrate. Determining appropriate values is non-trivial, and depending on the details, may have to be repeated for every new camera module.

5.8 AGC/AEC (Automatic Gain Control / Automatic Exposure Control)

5.8.1 Name

`rpi.agc`

5.8.2 Overview

The AGC/AEC algorithm controls the shutter time of the image sensor, the analogue gain applied by the image sensor and the digital gain applied by the ISP. Normally we refer simply to the "AGC algorithm" but it encompasses all these aspects. The metering process uses both a matrix of rectangular regions spread across the image and constraints derived from an intensity histogram of pixels in the image, as explained in the following sections.

Region-based Metering

The image is divided up into regions as shown in figure 9.

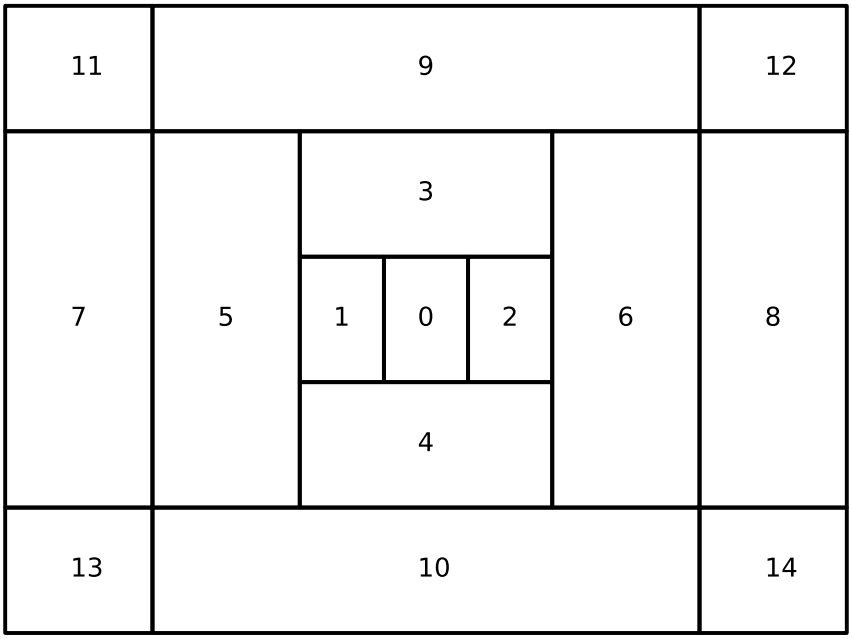

Figure 9: AGC metering regions.

An average luminance value for the image is calculated according to

$$Y = \frac{\sum_{i \in \text{regions}} w_i Y_i}{\sum_{i \in \text{regions}} w_i}$$

for weights w_i (defined in the JSON tuning file) and average region luminance values Y_i (as numbered in figure 9). This value is driven towards a target value `y_target`, also given in the tuning file. The weights can be adjusted to create various forms of average, centre-weighted or spot metering.

Histogram Constraints

Beyond controlling the weighted luminance value, the algorithm also uses a histogram, collected by the ISP on an earlier frame, to impose various other constraints on the final target exposure value.

We assume basic familiarity the the idea of an intensity histogram for an image. We define $F(p)$ to be the cumulative frequency of the histogram, and further normalise it so that the input value p is 0 at the bottom of the histogram and 1 at the very top. We also normalise the result so that $F(p) = 0$ indicates there are no pixels below p and $F(p) = 1$ means than all pixels lie below p. Further, we adopt the convention that pixels are spread evenly throughout the "bin" in which they lie, so that $F(p)$ is a continuous (in fact, monotonically increasing) function.

Observe that we always have $F(0) = 0$ and $F(1) = 1$.

Next we recap the notion of a *quantile*, which gives us the point $p = Q(q)$ in the range such that a proportion q of the pixels lie below p. Some quantiles are familiar to us, for example $Q(0.25)$, $Q(0.5)$ and $Q(0.75)$ are known as, respectively, the lower quartile, the median and the upper quartile of a distribution. We note further that F and Q are in some sense inverses, as $Q(F(p)) = p$, though this breaks down when F has flat regions (that is, there are no pixels in a part of the range) and Q becomes non-single-valued.

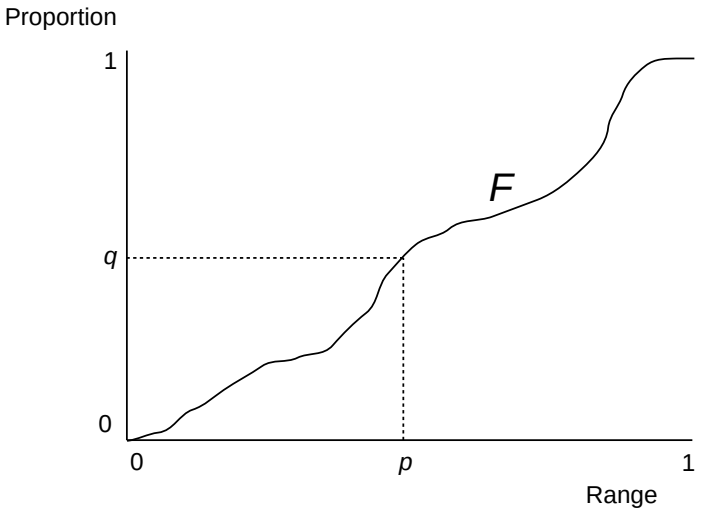

Figure 10: Cumulative frequency: a proportion q of the pixels lie below the value p

Q being ill-defined explains why quantiles are not ideal for metering as they are. Imagine, for instance, a picture containing only black and white pixels (an image of a newspaper page, perhaps). The median pixel will be either white or black, and will flick suddenly between them when only a single extra pixel pushes the number of white pixels either above or below the 50% mark. Any metering scheme based on this will be subject to occasional violent oscillations. Instead the Raspberry Pi algorithm uses the concept of the *inter-quantile mean* instead. Here we define $I(q_{lo}, q_{hi})$ to be the mean of all the pixels between the q_{lo} and q_{hi} quantiles. Observe how the possibly ill-defined nature of these locations in the pixel range become irrelevant as there are then no pixels to average there. We refer to $q_{hi} - q_{lo}$ as the *width* of the inter-quantile mean. We note that

- A narrow width gives precise control of a particular place in the histogram.
- A large width gives a stable response in the location of the inter-quantile mean.
- In practice we of course have to find some kind of happy medium.

Finally, our histogram constraints comprise therefore an inter-quantile mean (specifying values for q_{lo} and q_{hi}), a target Y value for it, and an indication of whether this is to be a *lower* or an *upper* bound for our final target exposure value. We illustrate this with a pair of examples.

1. $I(0.98, 1) = 0.5$, lower bound. Here we are saying that the top 2% of the histogram must lie at or above 0.5 in the pixel range (metering all happens before gamma, so this is a moderately bright value). Or in short, we are requiring "some of the pixels to be reasonably bright". This is a good strategy for snowy or beach scenes, also for images of documents. It actually raises the exposure of the whole scene; without it, the entire image would look dull grey which is undesirable in such circumstances. On the other hand when, as is usually the case, there is an abundance of bright and dark pixels, this constraint has no effect, so it is often reasonable to apply this constraint all the time.
2. $I(0.98, 1) = 0.8$, upper bound. This requires the top 2% of pixels to lie at or below 0.8 in the pixel range (actually a very bright value post-gamma). Effectively this lowers the exposure in order to stop pixels saturating, and is effective in metering for highlights or perhaps as part of an HDR metering strategy.

A number of these constraints can be grouped together as a named *constraint mode*, in which they are applied in sequence one after another. Applications are able to choose which of the available constaint modes in the JSON file they wish to use.

Exposure Modes

The region-based averages and histogram constraints determine the total exposure of the final scene, however they do not determine how to divide this up between the shutter time and the analogue gain. This is the job of the *exposure mode*. Each named exposure mode (in the JSON file) consists of a list of shutter times and analogue gains. First the desired analogue gain is set to 1 and the shutter time is allowed to ramp to the first value in the list. Thereafter, the analogue gain is allowed to ramp to the first value in its list. The procedure then simply repeats with the second value in each of the lists.

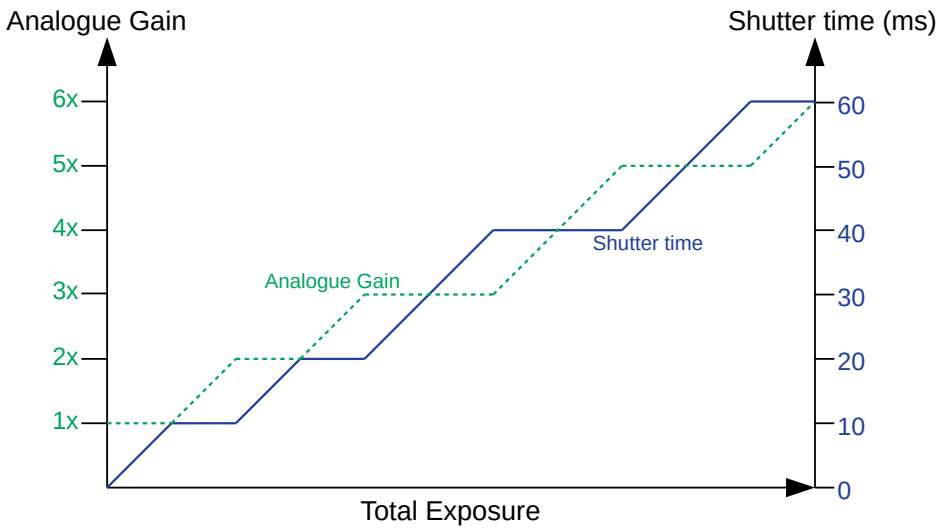

Figure 11: An example exposure profile - analogue gain and shutter time increase one after the other.

The maximum possible total exposure is determined by the final values in these two lists. Note how

- The maximum shutter time can be limited to avoid dropping the frame rate (assuming the camera driver supports this behaviour).
- If the camera driver does not support an analogue gain as high as listed in the exposure modes, the AGC algorithm will simply make up the difference with digital gain.
- For any total exposure value (less than the maximum), there is only a single pair of analogue gain and shutter time values that achieves it.

5.8.3 Implementation

The implementation of the AGC algorithm can be found in `controller/rpi/agc.cpp`. We note the following.

- Y targets are generally defined by piecewise linear functions, making it possible to vary the Y target with estimated lux level.
- Mostly the algorithm can be regarded as simply adjusting the total exposure of the images, filtering the requested values over time to prevent sudden changes. When it needs to reduce the exposure substantially, it may elect to cut the camera exposure more rapidly, whilst (temporarily) hiding any sudden changes from observers using increased digital gain. (Please refer to the source code for more details.)
- The algorithm is able to avoid exposure times that will result in flicker under artificial lighting. It would have to be given the period (in microseconds) of the lighting cycle.
- The implementation allows for the application to fix the shutter time and/or the analogue gain (for example, for fixed ISO exposures).
- The Raspberry Pi AGC algorithm does not currently handle flashes as we have no camera modules or boards that incorporate them.
- The Raspberry Pi AGC algorithm does not currently handle variable apertures as we have no camera modules that feature them.

- Our `Histogram` class, which provides histogram-related methods, is implemented in `controller/histogram.cpp`, though it does not in fact normalise the input and output ranges of the histogram as we did in the theoretical discussion above.

5.8.4 External API

The `Agc` class is derived from the `AgcAlgorithm` class. As such it defines the following publicly accessible methods.

AgcAlgorithm class API
`GetConvergenceFrames()` This returns the number of frames the AEC/AGC algorithm recommends be dropped, while the AGC converges, when the camera and the AGC algorithm are started from scratch.
`SetEv(double ev)` Sets the exposure compensation according to a linear scale. So for an extra stop, use `ev` = 2. For an extra half stop, use `ev` = 1.414.
`SetFlickerPeriod(double flicker_period)` Set the lighting cycle period in microseconds. Set to zero to disable flicker avoidance.
`SetFixedShutter(double fixed_shutter)` Sets a fixed shutter time, meaning only analogue (and digital) gain can be used to affect exposure. Set to 0 to cancel fixed shutter operation.
`SetMaxShutter(double max_shutter)` Sets maximum allowable shutter speed to use. This is typically based on the application requested framerate. Set to 0 to disable any limitation on the maximum shutter speed.
`SetFixedAnalogueGain(double fixed_analogue_gain)` Sets a fixed analogue gain value, meaning only shutter time can be varied to affect exposure. Set to 0 to cancel fixed analogue gain operation.
`SetMeteringMode(std::string const &metering_mode_name)` Choose which metering mode to use from the JSON file.
`SetExposureMode(std::string const &exposure_mode_name)` Choose which exposure mode to use from the JSON file.
`SetConstraintMode(std::string const &contraint_mode_name)` Choose which (histogram) constraint mode to use from the JSON file.

Table 14: AGC algorithm public API.

An example use of AGC methods:

```
Algorithm *algorithm = controller->GetAlgorithm("agc");
AgcAgorithm *agc_algorithm = dynamic_cast<AgcAlgorithm *>(algorithm);
if (agc_algorithm) {
    agc_algorithm->SetEv(1.414);
    agc_algorithm->SetExposureMode("sport");
}
```

5.8.5 Parameters

Name	Default	Description
startup_frames	10	For this many frames the AGC algortihm runs with maximum speed (1.0) so as to adapt quickly to the initial conditions.
convergence_frames	6	When queried by the *GetConvergenceFrames* method, the algorithm will return this value as the number of frames to drop, unless both shutter and gain values have been set (when there is no convergence required).
metering_modes	*Required*	A list of metering modes. Each mode specifies a name and 15 numbers, which are the weights for the region-based AGC calculation.
exposure_modes	*Required*	A list of exposure modes. Each mode specifies a name and a list of of *shutter* and *gain* values which define how the total exposure time is to be broken up between shutter time and analogue gain.
constraint_modes	*Required*	A list of constraint modes. Each mode specifies a name and a list of histogram constraints, involving the bound type (UPPER or LOWER), q values for an inter-quantile mean, and a Y target value for it.
y_target	*Required*	Pieceswise linear function defining a Y target value (out of a maximum value of 1.0) for the region-based metering.
base_ev	1.0	Optional parameter that linearly scales the final target exposure value of the algorithm. Useful for tuning if ever it is desired for the AGC to be globally a bit brighter (> 1.0) or darker(< 1.0) purely for reasons of taste.
speed	0.2	Speed of adaptation of the AGC algorithm. A larger value (with a limit of 1.0) makes the algorithm adjust the camera's exposure more quickly to the scene conditions.

Table 15: AGC algorithm parameters.

5.8.6 Metadata Dependencies

The AGC control algorithm requires the presence of

1. The DeviceStatus metadata (which should always be available from the camera).
2. The LuxStatus metadata from the Lux control algorithm.
3. The AwbStatus metadata from the AWB algorithm. Note that it is therefore best to list the AGC algorithm in the JSON file after the AWB algorithm (there is no reverse dependence of the AWB on the AGC algorithm).

The AWB control algorithm writes an AgcStatus object (controller/agc_status.h) into the image metadata.

5.9 ALSC (Automatic Lens Shading Correction)

5.9.1 Name

`rpi.alsc`

5.9.2 Overview

Lens shading, also known as *vignetting*, is the problem where light does not fall equally on all parts of the sensor, particularly near the edges. It causes obvious darkening but also discolouration to parts of the image, as the effect is not identical in all three colour channels.

Figure 12: Lens shading: no correction (left), luminance correction only (middle), full correction (right).

We see how the uncorrected image (left) shows obvious darkening towards the edges. The middle image corrects this luminance fall-off, but we still see the colours shifting towards the image edge. Whilst the very centre isn't too bad we clearly see progressively more of a cyan cast as we move away. The final image on the right shows how the image should look - there is no (or minimal) luminance fall-off, and the colour shift has been corrected.

Lens shading is corrected by creating a spatial 2-dimensional table of gain values that covers the image in question. The gains are located at regular horizontal and vertical intervals and pixels are corrected by interpolating a local gain value from the four nearest values on the grid. We create a separate table for each of red, green and blue so as to be able to correct the varying colours caused by the lens shading effect.

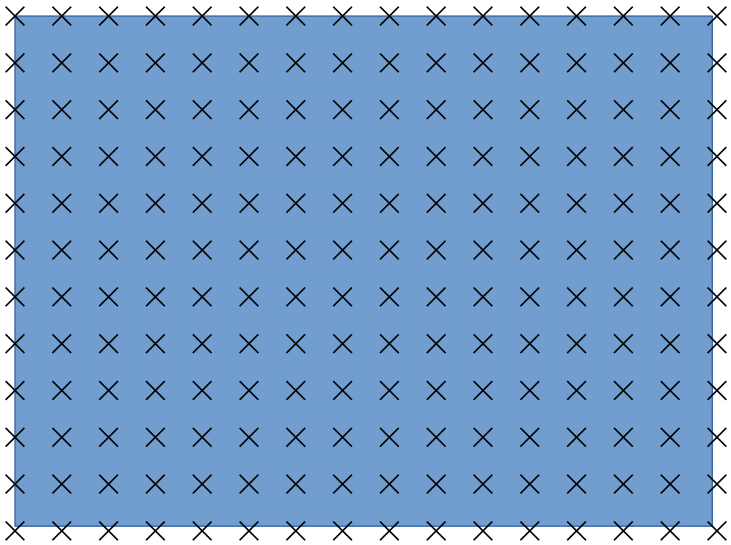

Figure 13: A grid of 16x12 lens shading gains. The grid may extend slightly beyond the image to ensure every pixel is covered.

It is generally found that while luminance correction is beneficial it is usually not necessary to correct it fully. Firstly we are quite accustomed to seeing images with at least a little darkening in the periphery and secondly, full correction can boost noise more than we might like. However, colour shading problems are generally viewed as more serious as they cause very obvious colour shifts and casts (especially when the background of an image is very uniform), and there is no real upside to not fixing them.

The Raspberry Pi ALSC algorithm separates the luminance and colour shading problems. We always correct any colour shading artifacts fully, and then a proportion of the luminance fall-off according to preference. We calculate separate tables for the red and blue channels to correct the colour shading (and set the green table to unity everywhere) and then post-multiply each of these by a single luminance correction table.

ALSC tables and parameters would normally be calculated automatically from a set of calibration images by the Camera Tuning Tool.

Luminance Shading

Luminance shading is measured straightforwardly from a set of calibration images. Whilst the algorithm calculates only a single average luminance correction table, there would be no difficulty in extending this to a more complex scheme should we wish.

Colour Shading

This is by far the more difficult part of the problem. We take a hybrid approach to the problem, calculating the red and blue colour correction tables partly by calibration and partly using an adaptive procedure.

Historically the preferred method for colour shading correction has been to measure separate correction tables for each of red and blue under different colour temperature illuminants. Then the correct tables are chosen (or interpolated) according to the current colour temperature being reported by the AWB algorithm. There are a couple of problems with this.

1. The nature of the colour shading actually varies with the whole spectrum of the illuminant, not just its apparent colour temperature. Using only the colour temperature based tables will invariably leave some illuminants with residual colour shading problems (this can be a particular problem when dealing with mixtures of incandescent, fluorescent or other types of light).
2. There are also variations between different camera modules. Whilst we may tune the colour shading correction tables with a particular reference (or "golden") sensor, other sensors in use out in the field may respond slightly differently, also causing residual colour shading problems (unless we go to great expense and calibrate every single module).

Our solution is to fix "as much as we can" using the colour temperature based calibration method, and then further to apply the adaptive procedure to "mop up" some of the residual problems. The adaptive procedure is calibrated so as not to be more aggressive than is necessary.

Colour Temperature based calibration

Here we simply list tables of G/R (or G/B for the blue tables) for our calibration images. We may list a separate table for each colour temperature that we have. As the algorithm runs it will interpolate as necessary so as to create an appropriate estimate of a table for the current illuminant. These tables are then applied to the *statistics* information from the ISP, so that the adaptive algorithm, which runs next, only makes the additional changes required on top of the calibration-based fixes.

5.9.3 Adaptive ALSC Algorithm

Without loss of generality we shall assume we are working with the red colour channel. Exactly the same process can be followed subsequently for blue (*mutatis mutandis*). We also denote each of the grid regions (or cells) with a number $i, 0 \leq i < N$, where N is the total number of grid cells ($16 \times 12 = 192$ in the case of the VC4 platform).

We start by defining a chrominance statistic for each grid cell so that $C_i = R_i/(G_i + k)$, using a small constant k to avoid numerical instabilities and potential division by zero, and also a neighbourhood $\mathcal{N}_i$ consisting of the north, south, east and west neighbours of grid cell i. Writing $\#\mathcal{N}_i$ to denote the number of neighbours of a grid cell we note that $\#\mathcal{N}_i = 4$ for all interior cells, 3 for edge cells and 2 for the four corner cells.

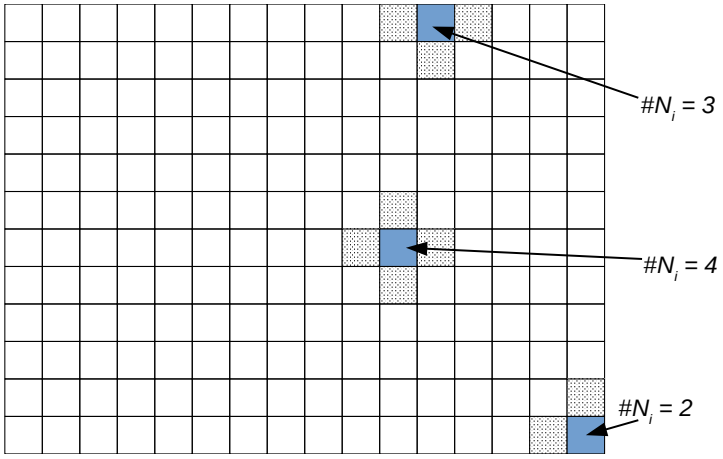

Figure 14: Cells and their neighbours in a 16x12 grid.

Now, the problem at hand is to find multipliers λ_i so that $\lambda_i C_i$ is the same "in places where it looks like it should be". To cope with this somewhat nebulous phrase we introduce a matrix of weights w_{ij} defined for $j \in \mathcal{N}_i$ and zero otherwise. Notice that w_{ij} forms a large (192 x 192 in this case) but sparse matrix with only $\#\mathcal{N}_i$ non-zero entries on each row. Where non-zero, w_{ij} will be calculated by comparing C_i and C_j. When they are very similar then w_{ij} takes a value near one; when they are very different, the value goes to zero. It could be implemented by a Gaussian-style (e^{-x^2}) function.

Now we can define a constraint saying that $\lambda_i C_i$ should be the same as its neighbours, taking account of the defined weights. Thus:

$$\lambda_i C_i = \frac{\sum_{j \in \mathcal{N}_i} w_{ij} \lambda_j C_j}{\sum_{j \in \mathcal{N}_i} w_{ij}}$$

There is obviously a danger here that all the w_{ij} can go to zero, so in practice we add another term weighted by the small value ε saying that "when the neighbours don't help, simply use the average of the neighbour's λ values". Hence:

$$\lambda_i C_i = \frac{\sum_{j \in \mathcal{N}_i} w_{ij} \lambda_j C_j + \frac{\varepsilon}{\#\mathcal{N}_i} (\sum_{j \in \mathcal{N}_i} \lambda_j) C_i}{\sum_{j \in \mathcal{N}_i} w_{ij} + \varepsilon}$$

Rearranging a little, we end up with a linear system of N equations:

$$(\varepsilon + \sum_{j \in \mathcal{N}_i} w_{ij}) C_i \lambda_i - \sum_{j \in \mathcal{N}_i} (w_{ij} C_j + \frac{\varepsilon}{\#\mathcal{N}_i} C_i) \lambda_j = 0$$

Rewriting these coefficients as a matrix M and the λ_i as a vector λ we finally have

$$M\lambda = 0$$

Solution

Astute readers will observe that, as there is no particular reason why the matrix M is singular, then there is only a single solution to this system, namely $\lambda = 0$. In practice, however, the equations lend themselves to the *Gauss-Seidel* method (https://en.wikipedia.org/wiki/Gauss%E2%80%93Seidel_method) wherein λ_i is updated from the i^{th} equation according to:

$$\lambda_i = \frac{\sum_{j \in \mathcal{N}_i} (w_{ij} C_j + \frac{\varepsilon}{\#\mathcal{N}_i} C_i) \lambda_j}{(\varepsilon + \sum_{j \in \mathcal{N}_i} w_{ij}) C_i}$$

Note how, informally, $\lambda_i C_i$ is replaced by a convex combination of its neighbours; therefore the iterations do not run off towards zero but remain bounded, iterating to a place of lower overall colour error.

5.9.4 Implementation

The ALSC algorithm is implemented in `controller/rpi/alsc.cpp`.

- The statistics are in fact obtained with the colour shading correction already applied. Therefore the algorithm has to start by dividing out its estimate of the previous correction before continuing (the function `copy_stats`).
- The calibrated tables are expected to be for the full field of view sensor frame. When operating in a cropped mode these tables need to be cropped and resampled (see `resample_cal_table`).
- The Gauss-Seidel method is applied with a small amount of over-relaxation so as to reach a point of low colour error more quickly (`gauss_seidel2_SOR`).
- We apply a normalisation to the λ values, such that $\min_i \lambda_i = 1$. This is because it is only their relative values that matter and we do not wish to add any unnecessary extra gain into the image (or indeed apply any gains less than unity).
- The main part of the algorithm - constructing the matrix M and the Gauss-Seidel iterations - runs asynchronously, and only every few frames. Other than that the implementation closely follows the method and nomenclature in this document.
- Lens shading algorithms do not have any additional public methods (at this time).

5.9.5 Parameters

Name	Default	Description
`frame_period`	12	The ALSC algorithm runs once per this many input frames. Note that new tables are generated on every frame, being filtered gradually towards the results of the most recent run of the ALSC computation.
`startup_frames`	10	For this many frames the ALSC algortihm runs as often as possible (ignoring `frame_period`) so as to adapt quickly to the initial conditions.
`speed`	0.05	Coefficient of the IIR filter that adapts the tables, on every frame, towards the latest ALSC result. A value of 1 would adopt the most recently calcualted results immediately.
`sigma`	0.1	Default value for the standard deviation of colour differences (R/G or B/G), used in calculating the weights w.
`sigma_Cr`	`sigma`	Standard deviation of red channel colour differences (R/G).
`sigma_Cb`	`sigma`	Standard deviation of blue channel colour differences (R/G).

min_count	10	Number if pixels in a statistics region for that region's data to be considered meaningful.
min_G	800	Average green value (out of a 16-bit dynamic range) in a statistics region for that region's data to be considered meaningful.
n_iter	28	Maximum number of Gauss-Seidel iterations for perform.
omega	1.3	Amount of over-relaxation to apply in Gauss-Seidel iterations. A value greater than 1 implies (increasing) over-relaxation; values less than 1 would imply under-relaxation.
calibrations_Cr	*Optional*	Tables, one for each colour temperature listed, giving the gains required to correct the red colour channel for that colour temperature in a 16x12 spatial grid. If omitted, the algorithm will behave as if these tables contained unity values everywhere.
calibrations_Cb	*Optional*	Tables, one for each colour temperature listed, giving the gains required to correct the blue colour channel for that colour temperature in a 16x12 spatial grid. If omitted, the algorithm will behave as if these tables contained unity values everywhere.
luminance_strength	1.0	Proportion of the full luminance correction to apply, so that zero means no luminance correction and 1 means 100% correction.
luminance_lut	*Optional*	16x12 spatial luminance correction table. This table gives gains that will be applied to all colour channels for full, or 100% luminance correction.
default_ct	4500	Colour temperature (in kelvin) to be assumed if no AWB metadata is found.
threshold	1e-3	Gauss-Seidel iterations stop once the step sizes in the solution are all less than this value.

Table 16: ALSC algorithm parameters.

5.9.6 Metadata Dependencies

The AGC control algorithm requires the presence of the AwbStatus metadata from the AWB algorithm. Note that it is therefore best to list the ALSC algorithm in the JSON file after the AWB algorithm (there is no reverse dependence of the AWB on the AGC algorithm).

The ALSC control algorithm writes an AlscStatus object (controller/alsc_status.h) into the image metadata.

5.10 Contrast

5.10.1 Name

`rpi.contrast`

5.10.2 Overview

The Contrast Control Algorithm is responsible for calculating a gamma curve with which to program the ISP hardware. In the first instance, the JSON tuning file may supply a gamma curve, and applications may allow users manually to change contrast and brightness (as is customary with monitors and TVs). Additionally, the algorithm has some ability to watch the image histogram being reported by the ISP and alter the gamma curve slightly so as to enhance the global contrast.

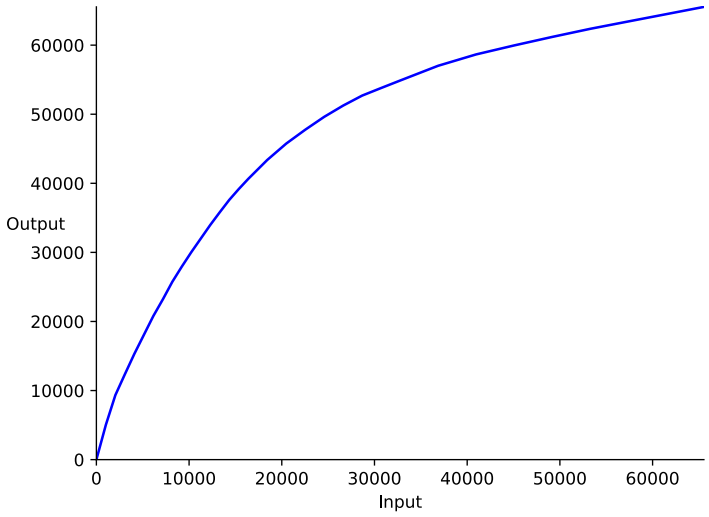

Figure 15: An example gamma curve with 16-bit inputs and outputs.

Note how the gamma curve is more properly an *inverse* gamma curve, looking more like $y = x^{1/2.2}$ than $y = x^{2.2}$. Furthermore, the "theoretical 2.2 gamma" is rarely used in practice as it gives rise to rather washed-out low contrast images.

Adpative Contrast Enhancement

Adaptive Contrast Enhancement uses the image histograms collected by the ISP. It works by selecting a particular *quantile* near the bottom of the histogram and moving it to a particular place in the output pixel range. A similar calculation is performed for a point near the top of the histogram. Normally we use this to ensure the resulting images have sufficiently strong contrast.

The algorithm also allows brightness and contrast to be adjusted globally. After applying the gamma curve to the pixel data, the following transformation is applied.

$$pixel_{out} = (pixel_{in} - 32768) * contrast + 32768 + brightness$$

(note: pixel values are always treated as being unsigned 16-bit numbers).

5.10.3 External API

The Contrast class is derived from ContrastAlgorithm. This means it implements the following additional public methods.

ContrastAlgorithm class API
SetBrightness(double brightness) Sets a fixed brightness offset (a pixel value out of a 16-bit range).
SetContrast(double contrast) Sets a fixed contrast value. A value less than one decreases the contrast, a value greater than one increases the contrast.

Table 17: Contrast algorithm public API.

```
    Algorithm *algorithm = controller->GetAlgorithm("contrast");
    ContrastAgorithm *contrast_algorithm = dynamic_cast<ContrastAlgorithm *>(
   algorithm);
    if (contrast_algorithm) {
        contrast_algorithm->SetBrightness(2000);
        contrast_algorithm->SetContrast(1.1);
    }
```

5.10.4 Parameters

Name	Default	Description
startup_frames	6	The first few frames emerging from the ISP can be rendered entirely as black, in case algorithms like AEC/AGC are still converging (and application code is not hiding them).
ce_enable	1	Whether to enable adaptive contrast enhancement. If set to zero, the gamma curve is used exactly as specified (subject to user brightness/-contrast settings).
lo_histogram	0.01	Quantile near the bottom of the histogram to be moved when ce_enable is set.
lo_level	0.015	Level (out of a maximum of 1.0) in the range where the current lo_-histogram quantile will be moved.
lo_max	500	Maximum amount (out of 16 bits) by which the low quantile may be moved.
hi_histogram	0.95	Quantile near the top of the histogram to be moved when ce_enable is set.
hi_level	0.95	Level (out of a maximum of 1.0) in the range where the current hi_-histogram quantile will be moved.
hi_max	2000	Maximum amount (out of 16 bits) by which the high quantile may be moved.
gamma_curve	*Required*	Gamma curve. A list of alternating input and output points, each given out of a 16-bit range. There must be an even number of points in this list.

Table 18: Contrast algorithm parameters.

5.10.5 Metadata Dependencies

The Contrast Control Algorithm does not require any other metadata.

The Contrast Control Algorithm writes a `ContrastStatus` object (`controller/contrast_status.h`) into the image metadata.

5.11 CCM (Colour Correction Matrices)

5.11.1 Name

`rpi.ccm`

5.11.2 Overview

The CCM (Colour Correction Matrix) Algorithm selects an appropiate CCM for the prevailing conditions to convert the camera's notion of RGB into our standard version of RGB according to

$$\begin{pmatrix} R_{standard} \\ G_{standard} \\ B_{standard} \end{pmatrix} = C \begin{pmatrix} R_{camera} \\ G_{camera} \\ B_{camera} \end{pmatrix}$$

where C is a 3x3 matrix. Note that the rows of C will normally sum to 1 (otherwise grey pixels will not be preserved).

Calibration is usually performed using the Camera Tuning Tool, and consists of a list of colour temperatures, each followed by a 3x3 matrix written out as 9 values in normal reading order. During operation, the algorithm will estimate a CCM - interpolating as necessary - using the current estimated colour temperature as reported by the AWB algorithm.

In the following example colour matrix

$$\begin{pmatrix} 1.80439 & -0.73699 & -0.06739 \\ -0.36073 & 1.83327 & -0.47255 \\ -0.08378 & -0.56403 & 1.64781 \end{pmatrix}$$

observe how we have large values on the diagonal and smaller but still significant negative values next to them - this is fairly typical.

The algorithm allows the colour saturation of the output image to be altered. It does this by transforming the colour matrix $C_{original}$ from above according to the following equation for the saturation parameter s.

$$C_{new} = A^{-1} \begin{pmatrix} 1 & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & s \end{pmatrix} A \, C_{original}$$

where the matrix A here converts from RGB to YCbCr, *viz.*

$$A = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{pmatrix}.$$

5.11.3 External API

The `Ccm` class is derived from `CcmAlgorithm` and therefore implements the additional public method `SetSaturation(double saturation)` which sets the desired level of colour saturation to the value given.

So for example, a value of zero will turn an image greyscale; values greater than one will produce over-saturated colours.

5.11.4 Parameters

Name	Default	Description
`ccms`	*Required*	List of colour temperatures followed by 3x3 colour matrix.
`saturation`	*Optional*	An optional piecewise linear function that allows colour saturation to vary with the estimated lux level of the image (for example, to reduce colour saturation in low light). It is specified by listing alternative lux and saturation values.

Table 19: CCM algorithm parameters.

5.11.5 Metadata Dependencies

The CCM Control Algorithm requires

1. The `AwbStatus` metadata from the AWB algorithm, and
2. the `LuxStatus` metadata from the Lux algorithm.

The CCM Control Algorithm writes a `CcmStatus` object (`controller/ccm_status.h`) into the image metadata.

5.12 Sharpening

5.12.1 Name

`rpi.sharpen`

5.12.2 Overview

The Sharpening Control Algorithm gives rudimentary control over the Sharpening hardware block in the ISP according to the following simple model.

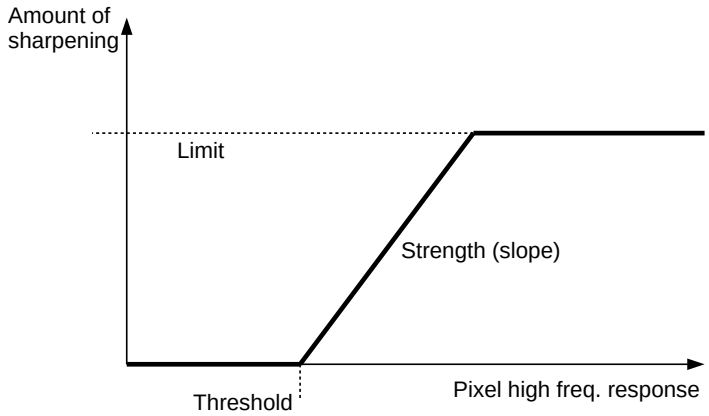

Figure 16: Sharpening Parameter model, where:

- the pixel high frequency response is measured, and nothing below *threshold* gets sharpened,
- the *strength* gives the rate (or slope) of how much sharpening increases with the pixel high frequency response, and
- *limit* is a maximum value to the amount of sharpening applied.

The parameters are normally varied automatically according to the mode currently being used by the camera. The default configuration and values should normally be sufficient for many purposes, although they may need adjustment by hand for some sensors.

5.12.3 Parameters

Name	Default	Description
`threshold`	1.0	Threshold below which pixels are not sharpened. Use values < 1 to sharpen more low contrast pixels; use values > 1 to sharpen fewer low contrast pixels.
`strength`	1.0	Slope of the rate at which the smount of sharpening applied increases. Use values < 1 to reduce the rate at which sharpening increases; use values > 1 to apply increasing amounts of sharpening more quickly.
`limit`	1.0	Limit to the total amount of sharpening applied. Use value < 1 for a lower maximum amount of sharpening; use values > 1 to allow larger amounts to sharpening to be applied.

Table 20: Sharpening algorithm parameters.

5.12.4 Metadata Dependencies

Sharpening has no dependencies on any other metadata.

The Sharpening Control Algorithm writes a `SharpenStatus` object (`controller/sharpen_status.h`) into the image metadata.

5.13 Metadata and Statistics Usage

We finish with a diagram (figure 17) showing how image metadata is both read and written to by the various stages of procesing.

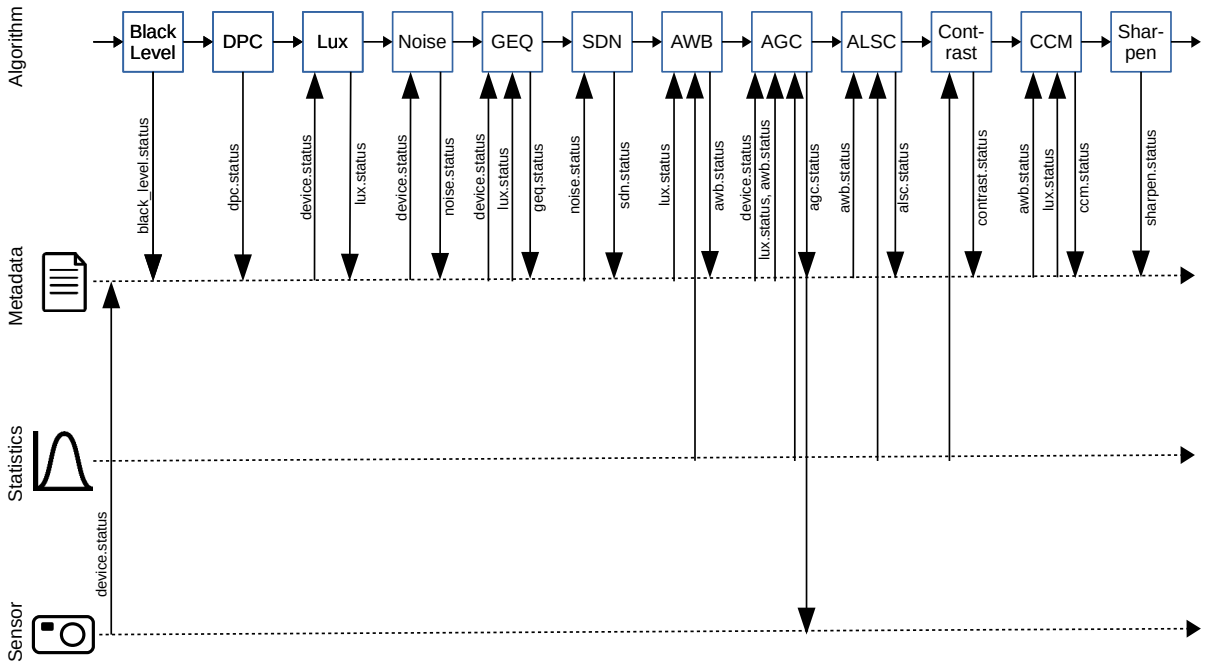

Figure 17: Usage of image metadata and other information by ISP control algorithms.

In figure 17, observe that:

- Device metadata (`device.status`) is always supplied from the camera information.
- After AGC/AEC has run, updated exposure and gain values are fed back to the camera (though this happens outside of the algorithm itself).
- Some algorithms use image statistics but many do not. Many use only the `device.status` from the image metadata.
- All algorithms write `status` information back to the image metadata.

6 Camera Tuning Tool

6.1 Overview

The Camera Tuning Tool (CTT) is a Python program designed to produce a fully working camera tuning JSON file from a relatively small set of calibration images. Once the tool has run, there should be either no or only minimal further tweaking to the JSON file required in order to obtain the desired image quality. The tuning algorithms are furthermore designed to work with a minimum amount of expensive or specialised equipment. The processes required in creating a finished camera tuning are as follows.

1. Firstly, a functional V4L2 camera driver must be written (see Chapter 3). For the purposes of writing the camera driver, an uncalibrated tuning file (`data/uncalibrated.json`) can be copied which should provide recognisable images. Care should be taken, however, that the black level listed in the file is adjusted to match the black level specified in the sensor data sheet (and scaled up to a 16-bit range).
2. The set of calibration images must be captured. Again, this should use the uncalibrated tuning file. There are two types of calibration images, those with a Macbeth chart, and a further set of completely uniform images for measuring lens shading.
3. With the calibration images all correctly named and stored in a folder, the CTT can be run. The CTT finds Macbeth charts in images automatically and uses them to measure noise profiles, green imbalance, white balance and colour matrices.
4. The output JSON file of the CTT can be used directly, possibly with minor further tweaking.

6.2 Raspberry Pi *libcamera-apps*

Image capture is most conveniently performed using Raspberry Pi's *libcamera-apps*. For readers who are familiar with the existing *raspistill* and *raspivid* applications, the *libcamera-apps*, including *libcamera-still* and *libcamera-vid* are *libcamera*-based replacements. Though not completely identical in functionality, they aim to be very similar.

Full instructions, including how to obtain and build the *libcamera-apps*, can be found here: https://github.com/raspberrypi/libcamera-apps.

6.3 Software Requirements

The CTT requires Python 3 to be installed, and uses the following additional modules.

```
matplotlib
scipy
numpy
cv2
imutils
sklearn
pyexiv2
rawpy
```

The following commands are sufficient to install all the required libraries and modules in an otherwise clean Ubuntu 18.04.4 LTS installation. This process should be similar in other distributions.

```
sudo apt install python3-pip libexiv2-dev libboost-python-dev
pip3 install opencv-python imutils matplotlib scikit-learn py3exiv2 rawpy
```

6.4 Equipment

The additional equipment required for camera tuning is described below.

6.4.1 X-rite (Macbeth) Colour Checker

These charts are very well known, and one is shown in figure 18. They normally cost less than £100.

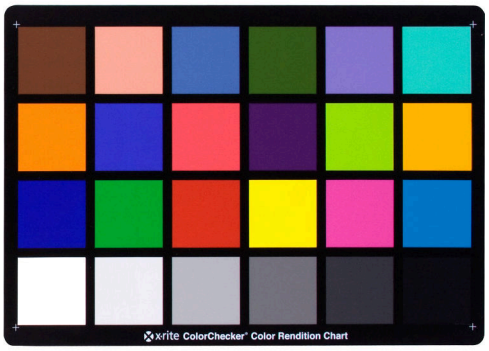

Figure 18: X-rite Macbeth Chart.

6.4.2 Colorimeter

Some kind of colorimeter is required in order to measure the colour temperature and light level of the calibration images. One such is shown in figure 19, costing normally a few hundred pounds.

Figure 19: A Colorimeter.

Depending on the application it may be possible to get by with significantly simpler instruments. For example, a mobile phone may well report colour temperature and light level, and while its readings may be quite approximate, it may suffice for certain use cases.

6.4.3 Integrating Spheres and Flat Field LEDs

There are some relatively sophisticated pieces of equipment - integrating spheres and flat field LED lamps - that can help calibrate lens shading, and these are likely to prove beneficial when they are available. However, the CTT is designed to produce reasonable results without them, so unless a particular measurable degree of accuracy is required in the correction, no further equipment should really prove necessary.

6.5 Capturing Calibration Images with *libcamera*

The tuning process requires raw images to be captured and fed into the tuning tool. We recommend Raspberry Pi's *libcamera-apps* for this purpose, as described earlier, however the *qcam* application supplied with *libcamera* itself is also suitable. Both applications will capture the required full resolution raw files (in DNG, or Adobe "digital negative" format).

In due course we expect the necessary executable files to be included as part of the standard Raspberry Pi OS distribution. For now, however, we refer users to the linked *libcamera-apps* instructions above. Anyone wanting to study and modify the code would need to follow these steps in any case.

6.5.1 Prepare your Pi

For an up-to-date guide on setting up the build system and runtime environment for *libcamera* on the Raspberry Pi, please visit https://github.com/raspberrypi/documentation/blob/master/linux/software/libcamera/README.md

6.5.2 Capturing a Raw Image

Camera tuning files are kept in the `data` folder under the Raspberry Pi IPA folder, and the control algorithms expect to find a JSON file there, named according to the sensor name exposed in the driver, for every camera that is used. When tuning a new camera, of course, we don't have one so we provide a tuning file for *uncalibrated* cameras which can be copied. For example, when tuning a new sensor named `xyz123` we simply copy `uncalibrated.json` to `xyz123.json`, all in the same folder.

The newly copied uncalibrated tuning will produce recognisable images from pretty much any sensor, but there is just one single parameter that *does* have to be configured before use. This is the *black level*, and it must be set to the correct value from the sensor datasheet scaled up to 16 bits. So if your sensor produces 10-bit samples, and the black level is 64, then the correct value in the new uncalibrated tuning file (`xyz123.json`) will be 4096.

To capture a DNG using *libcamera-still*, you should ensure its location is in your PATH variable, or specify the location precisely when starting it. Now enter (substitute an image name of your choice for `image.jpg`)

```
libcamera-still -r -o image.jpg
```

which will produce a full resolution jpeg file named `image.jpg` and a DNG file `image.dng` (the same name but with the extension `.dng`). This second file contains the raw data that produced the jpeg.

Note that the CTT may sometimes discard an image if it is too over-exposed to be useful, and it will report this in the console log. Mostly the tuning produced should still be reasonable, but you may wish to retake those particular images with a lower exposure, for example:

```
libcamera-still -r -o image.jpg --ev -1.0
```

This will reduce the exposure by one stop.

6.6 Image Capture Requirements

As we have already explained, there are two distinct types of calibration images. Because we are most likely to be using the uncalibrated tuning, it is possible that the colour balance may appear incorrect in some of the images. Rest assured that this makes no difference - it is only the raw camera data (which is unaffected) that is used by the tuning tool; the image displayed while capturing is purely for convenience and to aid framing each shot.

6.6.1 Macbeth Chart Images

The aim here is to capture images containing a Macbeth chart. The CTT locates the Macbeth charts automatically, therefore the chart should be

- Reasonably central,
- Not too small,
- Reasonably straight, and
- Unobscured, except perhaps a little bit at the edges (where maybe it is being held).

The images should be captured under a range of different colour temperates spanning at least the operating conditions the camera is expected to encounter. For a camera system to be used across a wide range of conditions, this might include lamps from about 2500K up to about 8000K. It is acceptable to capture images both in a camera lab that can generate the different illuminants, and/or out in real world operating conditions. In general, the closer the calibration pictures to the final operating environment, the better the results are likely to be. Below are a couple of acceptable calibration pictures.

Figure 20: Acceptable Macbeth Chart calibration images.

The left hand image is taken in a small purpose-built camera lab/studio, whist the second was merely taken out in an office lobby. We would probably not recommend going very much smaller than the chart in this left hand image. Observe further that the right hand image shows some barrel distortion - at the level shown the CTT is still able to work with the image.

Once captured, images should be named so as to contain the colour temperature (followed by 'k' or 'K') and the lux level (followed by 'l or 'L') in the filename. Valid filenames might include

- `imx219_2954k_1749l.dng` for an image captured in 1749 lux with a colour temperature of 2954K, or
- `1749L_2954K.dng` for the same image.

6.7 Lens Shading Images

As with the Macbeth calibration images, images should be captured in different colour temperatures depending on what illuminants are available (though not necessarily the same ones). Images that accurately reflect and cover the range of operating conditions are likely to be beneficial. Here, however, the image must be of a completely featureless, flat and uniform surface, such as a wall. Ideally the surface should probably be reasonably neutral in colour though this is not critical. Again, the filenames should contain the colour temperature (followed by 'k' or 'K'). The lux level does not need to be recorded on this occasion, though the filenames **must** include `alsc` so that these images can be distinguished from the Macbeth chart images as they are loaded.

Where specialised equipment is available a single lens shading image for each colour temperature should be sufficient. Otherwise images should be captured where the luninance variation across the scene is as low as practicably possible, and the CTT will expect a number of calibration images for each colour temperature, where each image should be taken with the camera rotated at a different angle. For example, for each colour temperature we might submit

- 2 images each, one where the camera is the usual way up and the other where the camera is held upside down, or
- 4 images each, where each image rotates the camera by a further 90 degrees.

The CTT averages the images for each colour temperature, thereby reducing the effect of the uneven illumination on the scene and making the intrinsic lens-produced colour and luminance shifts more discernible.

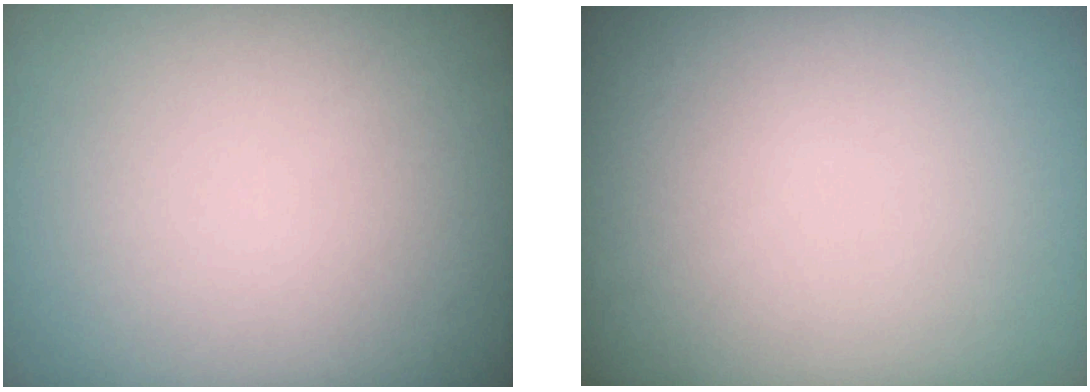

Figure 21: Lens shading images `alsc_3850k_1.dng` and `alsc_3850k_1u.dng`. Notice how the left image is slightly darker on the right whereas the right image is slightly darker on the left - because the camera had been turned by 180 degrees.

The single luminance correction table required by the ALSC algorithm is calculated by averaging together all the lens shading calibration images, for all colour temperatures.

6.8 Creating the Tuning

6.8.1 Collecting the Files

Once all of the calibration images have been captured, they should be placed together into a single folder with no other image files. The directory listing below shows an example of what such a calibration folder might contain.

```
2498K_1061.dng
2811K_4031.dng
2911K_12081.dng
2919K_6051.dng
3627K_12471.dng
4600K_9981.dng
5716K_10691.dng
8575K_1701.dng
alsc_3000K_1.dng
alsc_3000K_1u.dng
```

```
alsc_3850K_1.dng
alsc_3850K_1u.dng
alsc_6000K_1.dng
alsc_6000K_1u.dng
```

In this example we have eight Macbeth chart images covering eight different illuminants and three pairs of lens shading images covering only three colour temperatures, and where the second of the pair is rotated 180 degrees compared to the first. Please be aware that there should be only a single Macbeth chart image at each distinct colour temperature.

6.8.2 Running the Tool

The tuning tool can be found under the root *libcamera* directory in `utils/raspberrypi/ctt` and can be run by executing `ctt.py` found there. It takes the following arguments

Name	Default	Description
`-i`	*Required*	Path of the folder containing the calibration images.
`-o`	*Required*	Name of the output JSON file to contain the camera tuning.
`-c`	*Optional*	The CTT takes a JSON file containing its own configuration options (see below). If omitted, default parameters are used.
`-l`	`ctt_log.txt`	Name of a log file to which diagnostic output from the tool is written.

Table 21: Camera tuning tool command line options.

For example, one might, in the *libcamera* folder `utils/raspberrypi/ctt`, enter

```
./ctt.py -i ~/imx219_calibration_folder -o imx219.json
```

The tool takes quite a few seconds to run, reporting its progress during this time. The default configuration should normally be appropriate, but an example CTT configuration file is included below (also `ctt_config_example.json` in the same folder) for completeness.

```
{
  "disable": [],
  "plot": [],
  "alsc": {
    "do_alsc_colour": 1,
    "luminance_strength": 0.5
  },
  "awb": {
    "greyworld": 0
  },
  "blacklevel": -1,
  "macbeth": {
    "small": 0,
    "show": 0
}
```

If necessary the fields here can be modified, as follows.

Name	Description
`"disable":`	Do not tune the listed stages. Generally should be left as `[]`.
`"plot":`	Show plots during the tuning process.

`"do_alsc_colour":`	Whether to calibrate for colour shading correction. Would be omitted for a monochrome sensor.
`"luminance_strength":`	Luminance shading strength to specify in the final output file, between zero and one (inclusive).
`"greyworld":`	Whether to request Grey World AWB in the tuning file, or the (usual) Bayesian algorithm.
`"blacklevel":`	Black level to use for this sensor (out of 16 bits). The value -1 indicates to attempts to deduce it from the input file, if possible. Otherwise it is recommended to enter the correct black level obtained from the sensor data sheet.
`"small":`	Whether to search for extra small Macbeth charts in images (takes slightly longer).
`"show":`	Whether to show locations of Macbeth charts found in images.

Table 22: Camera tuning tool runtime config parameters.

6.9 Tweaking the Tuning produced by the Tool

6.9.1 Blocks not Tuned

There are a handful of blocks that the CTT does not actually tune but merely outputs default parameters. In these cases the default parameters should normally be sufficient (they are already "adaptive" to the image in some sense) and there would be no particular reason to change them other than as part of any more general aesthetic alterations to the tuning file. These blocks are:

1. Defective Pixels (`rpi.dpc`) - this will be left at "normal" correction strength which should usually be appropriate.
2. Spatial Denoise (`rpi.sdn`) - there is no particular tuning for this block, however, as it inherits and uses the image noise profile which *is* tuned, there should be no particular need to change anything here.
3. Contast (`rpi.contrast`) - the CTT outputs a gamma curve though in fact it is merely a fixed "sensible" gamma curve. No actual measurements are made in order to deduce it, though again, it should work well in nearly all circumstances.
4. Sharpening (`rpi.sharpen`) - the default parameters have been set to give reasonable results with our existing sensors. However, it is possible that other sensors may require some manual tuning. For example, for lower resolution sensors with larger pixels we might expect to have to reduce the amount of sharpening.

6.9.2 Guidance on how to Tweak the Tuning

The CTT makes every effort to produce a JSON tuning file that can be used directly and with minimal further effort. Of course there occasions when we may wish to tweak it, either for purely aesthetic reasons ("I want less sharpening", or "I want brighter colours") or because the camera is used in a slightly different situation than the one tuned for (maybe it is used under an illuminant that was not in the calibration set, meaning the colours could be slightly "off"). We include some general guidance for each block on how it might be tweaked.

Black Level (`rpi.black_level`)

There is nothing really to tweak here as the black level should match the one in camera data sheet (scaled up to 16 bits).

Defective Pixel Correction (`rpi.dpc`)

There is only a single `strength` parameter here, which defaults to 1 ("normal"). If defective pixels are not apparent it may be worth setting this down to zero as this would in theory allow very slightly more detail to be recovered, although the effect would be very marginal.

If defective pixels *are* a problem then the value should be increased to 2. Note that testing should always include low light images as "weak" pixels tend to come crawling out of the woodwork when the analogue gain is high.

Lux (`rpi.lux`)

There should not be anything to tweak here. The values are derived from one of the Macbeth Chart calibration images and unless these were categorised with an incorrect lux reading it is hard to envisage what one might need to change.

Noise (`rpi.noise`)

The noise profile itself - defined by the parameters here - should generally not be changed. If more or less spatial denoise is wanted then it would be better to alter the Spatial Denoise parameters (`rpi.sdn`).

Green Equalisation (`rpi.geq`)

Test images will have to be examined to determine if there is a green imbalance problem, as evidenced by the "maze" artifacts discussed previously. If the calibration images make up a wider set of conditions than those in which the camera will be used, then it is possible the numbers here could be reduced. However, if maze artifacts are visible then they will need to be increased. The values here (offset and slope) define a straight line - if maze artifacts are seen in the darker parts of an image then increasing the offset may be more effective; if they are seen in the bright parts of an image then increasing the slope may be recommended.

Maze artifacts may depend on both the light level and colour temperature of an illuminant and it is hard to predict exactly when they are likely to be worst. Therefore a reasonably broad test set would be advised to verify any changes.

Spatial Denoise (`rpi.sdn`)

The Spatial Denoise control algorithm takes the previously calculated Noise Profile and derives ISP parameters from it. It has two parameters - `deviation` which multiplies the standard deviation of the pixel noise, and `strength` which determines how much of the original (noisy) pixel to blend back in. We can provide the following rules of thumb.

1. If an image is showing speckles of noise (the "noise speckles" we saw earlier) then the `deviation` (default value 3.2) should be inceased. Once there are no noise speckles there is no benefit in increasing this further (you will just "eat" image detail).
2. It may sometimes be possible to reduce the deviation, though careful testing would be required to ascertain whether the level is still sufficient.
3. Once the deviation is set appropriately, the amount of denoise is controlled with the `strength` (where zero effectively disables denoise altogether and one denotes maximum denoise). Increasing this will use more of the denoised pixel, though there may come a point where the image starts to look "plasticky" or "like an oil painting". In this case the strength should be reduced, allowing a little of the original noise back.

Figure 22: Too little denoise (left), and on the right the leaves look even more plasticky than they really are!

AWB (`rpi.awb`)

Problems related to colours are very difficult to debug without capturing an image, in the problematic conditions, with a Macbeth Chart - so doing this should really be the first step. Once such an image is obtained it may simply be worth re-running the CTT with the new image.

Secondly, the AWB algorithm has some global "sensitivity" parameters. These are primarily designed for coping with module to module variation among the cameras, but can also be used to shift the AWB tuning slightly towards either purple (increase `sensitivity_r` and `sensitivity_b` by 5 or 10 percent) or towards green (reduce them by a similar proportion). This accounts quite conveniently for users who may prefer a very slightly greeny-yellow tone to their colours and those who do not.

The other global parameters that we may wish to adjust based on personal taste are `whitepoint_r` and `whitepoint_b`. For instance, people often quite like a slightly "warmer" look to their images, so this may sometimes be a worthwhile change to make globally. The values can be experimented with, but setting `whitepoint_r` to 0.05 and `whitepoint_b` to -0.05 should produce a slight but noticeable warming.

Beyond this we have to start delving more into the depths of the algorithm. Questions to ask are:

Figure 23: Effect of setting sensitivities to 0.8 (left), 1.0 (centre) and 1.2 (right).

- Does the illuminant of my problem picture actually lie on the CT curve? If not you might need to move at least part of it.
- Is it searching enough of the CT curve to find my illuminant?
- Is it that some of my illuminants are on the curve but others lie too far to the side? If both need to work you might need to increase the `transverse_pos` or `transverse_neg` values, which determine how far off the curve an illuminant may lie.

Sometimes it may simply be that the illuminant is in the feasible set but the image content is misleading the algorithm (it has a colour cast that can be interpreted as being caused by a different illuminant). In some cases the prior illuminant distributions can be tweaked to force the algorithm to a particular solution - for example, in bright conditions we bias the results quite strongly towards sunlight, whatever the image content. But bear in mind that AWB is fundamentally an under-constrained problem - you can never be totally sure that the white wall under a yellowish illuminant wasn't really a yellow wall under a white-ish illuminant.

AGC/AEC (`rpi.agc`)

The CTT just outputs a fixed set of AGC/AEC parameters, though in practice this set works pretty well with any camera.

If the tuning is found to be globally a bit too dark or a bit too bright, this can easily be adjusted using the `base_ev` parameter. Setting this to a value less than one makes everything a bit darker whilst leaving everything else untouched, and a value greater than one makes everything a bit brighter.

Figure 24: `base_ev` at the default value 1.0 on the left, and at 1.414 - about half a stop - on the right.

If more specific changes are required it is largely a matter of taste how to set:

- The weightings of the AGC regions (in the metering modes);
- The manner in which analogue gain and exposure are divided up in the exposure modes (possibly so as to limit the camera framerate);
- The various Y target functions (which allow the Y target to change with lux level); and
- The histogram constraints (though an understanding of the section on the AGC algorithm will be required).

Automatic Lens Shading Correction (`rpi.alsc`)

There are a number of large tables here generated by the CTT, and it's unlikely that one would ever want to make changes to these by hand. However, there are a number of parameters that one might reasonably want to fiddle with.

- `omega` controls the amount of over-relaxation in the Gauss-Seidel method. The default value is 1.3; any value close to 1 is likely to work fine.
- `luminance_strength` determines the amount of luminance correction (recall that colour shading correction is always applied fully). Full luminance correction would be applied by the value 1.0, though in practice small values in the 0.5 to 0.8 range will look better.
- `n_iter` gives the number of Gauss-Seidel iterations. Generally the iterations stop before this limit anyway, so changing it does not have a great effect. The value zero can be useful as a way of turning off the adaptive algorithm completely (only the pre-measured calibrated tables will be applied).
- Note also that the pre-measured calibrations can be disabled either by deleting them from the file, or renaming `calibrations_Cr` to something like `x.calibrations_Cr`, and similarly for `calibrations_Cb`.
- The sigma values can be increased to force the adaptive algorithm to "mop up" more residual colour shading. Once you get to values beyond 0.01 there is a risk that some genuine colour variation in the image might start to get slightly smeared; such large values are therefore not recommended.

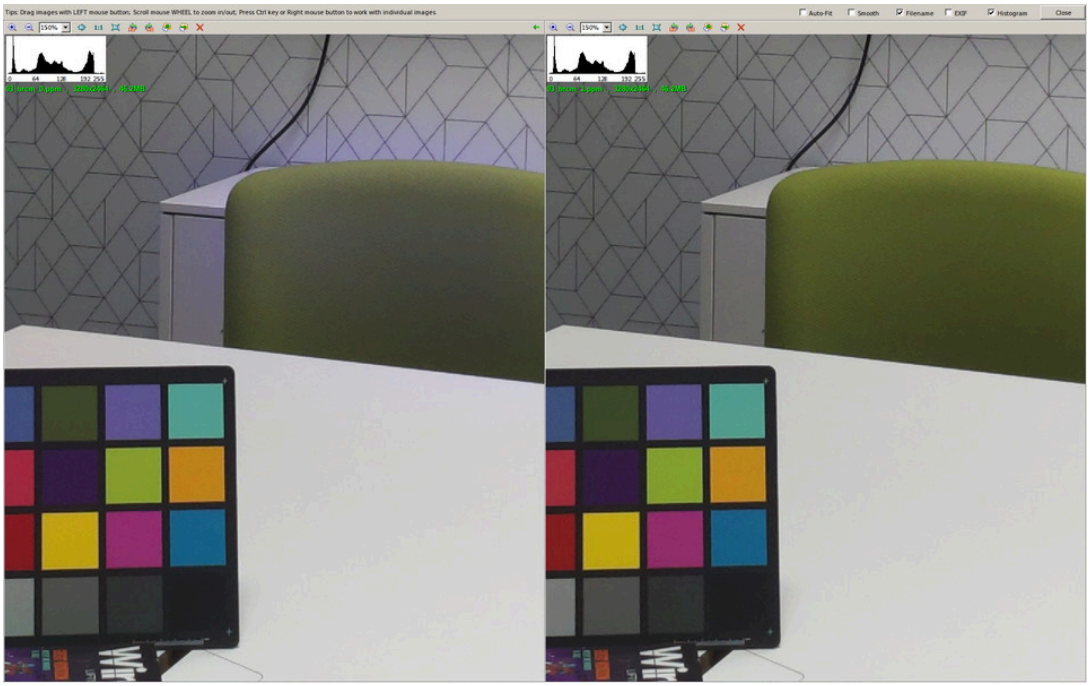

Figure 25: `sigma` has the extreme value 0.03 on the left - spot the purple halo round the green chair.

Contrast (`rpi.contrast`)

Firstly, the gamma curve can easily be changed or replaced by another curve, there is no constraint on how many points there must be or at what intervals they lie - any piecewise linear function will do. The values must of course occur in pairs (each pair of numbers is an (x,y) point), and obey the 16-bit dynamic range.

Setting `ce_enable` to zero completely disables any runtime adaptation of the gamma curve. When set to one the global contrast of an image can be increased by one or more of:

- Choosing a higher quantile value for `lo_histogram` (default 0.01).
- Choosing a lower level in the output range `lo_level` (default 0.015).
- `lo_max` (default 500) might need to be increased if the desired adjustment is larger than this value.

The high end of the dynamic range can be manipulated similarly using the "hi" (rather than "lo") versions of these parameters, however, the effect is not usually as significant.

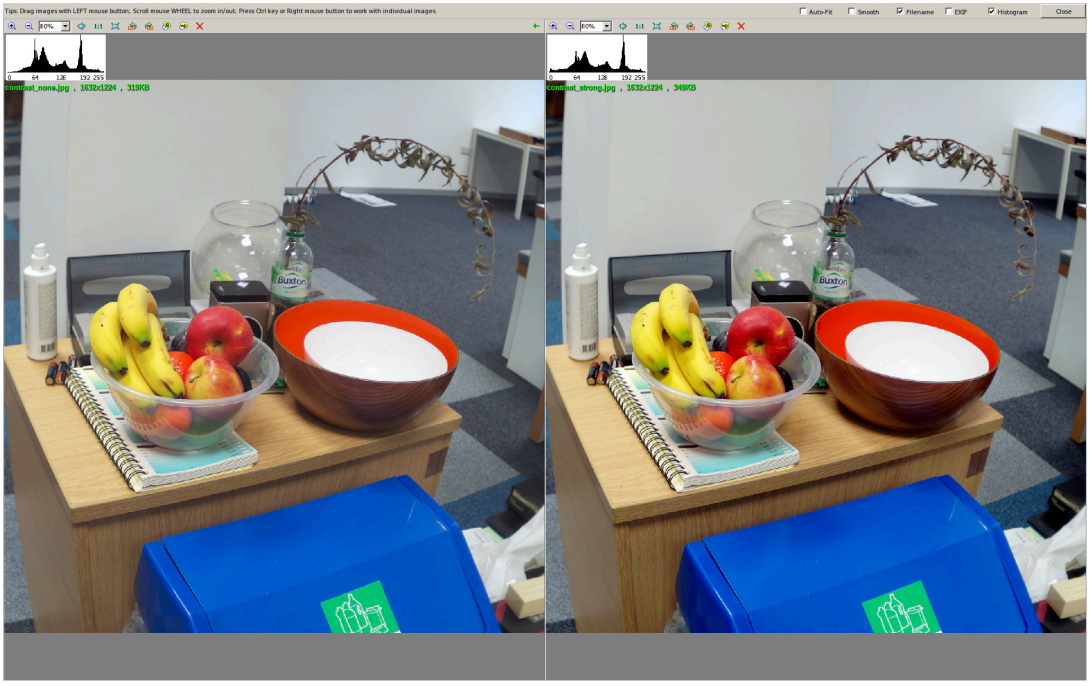

Figure 26: No contrast enhancement (left) and strong contrast enhancement (right) - note the change to the image histogram.

Colour Correction Matrices (`rpi.ccm`)

Matrices are calculated by the CTT from the Macbeth Chart images. It fits them in RGB space as this is most simple, however, there would be other arguably better ways to accomplish this if one wished - for example measuring the fit error in Lab* space.

Otherwise there are some easy manipulations enabled by the `saturation` piecewise linear function. For instance one could globally reduce or increase the colour saturation. Adding the following would increase colour saturation globally by 10%.

```
    "saturation": [0, 1.1, 10000, 1.1]
```

Sharpening (`rpi.sharpen`)

All the parameters (`threshold`, `strength` and `limit`) assume default values of 1.0. If it is clear that significantly less (or more) sharpening is required, for example if the sensor is exhibiting rather different behaviour from that which we have been used to, it may be worth applying a few rounds of:

- *halving* all the parameters (if there was far too much sharpening), or
- *doubling* all the parameters (when there was far too little sharpening)

until the degree of sharpening appears to be approximately of the right order of magnitude. Once this has been achieved the parameters can be fine tuned. If more sharpening is required:

1. Consider lowering the `threshold` as this will cause more low-contrast detail to be picked up and sharpened.
2. Consider increasing `limit` - this will cause the strongest most obvious edges to be sharpened more.
3. Consider increasing `strength` - this will cause all details to be sharpened more up to the given `limit`.

In order to reduce the amount of sharpening, the reverse steps should be taken. In particular, if it looks as though noise is being picked up and sharpened unwantedly, then the `threshold` value needs to be raised.

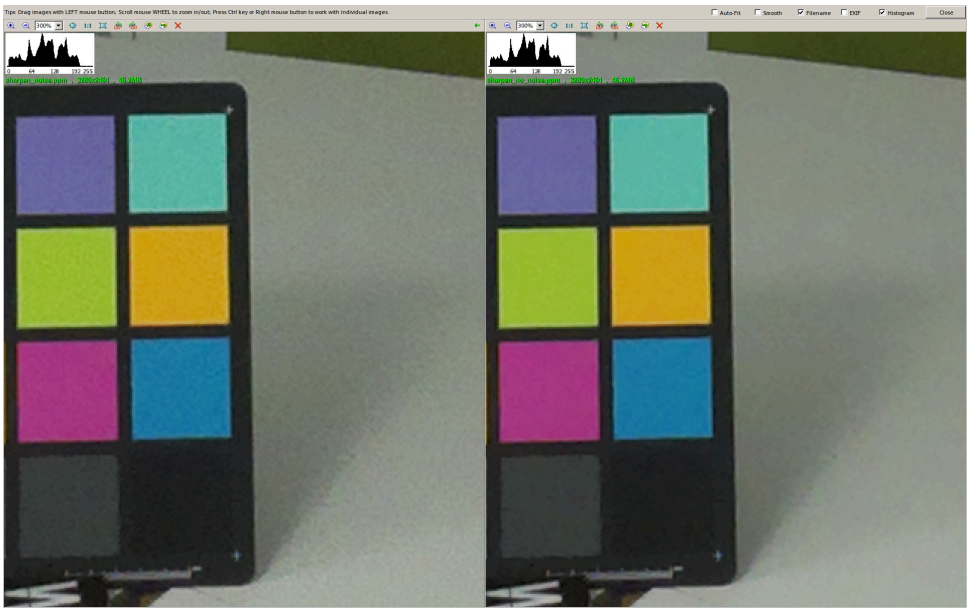

Figure 27: On the left too much residual noise is being sharpened because `threshold` was lowered.