

Python と Colab で作る - RAW 現像ソフト作成入門

目次

1. はじめに
2. カメラ画像処理について
3. 基本的な処理
 - 3.1 準備
 - 3.2 簡易デモザイク処理
 - 3.3 ブラックレベル調整
 - 3.4 ホワイトバランス調整
 - 3.5 ガンマ補正
4. 重要な処理
 - 4.1 この章について
 - 4.1 線形補間デモザイク
 - 4.2 欠陥画素補正
 - 4.3 カラーマトリクス
 - 4.4 レンズシェーディング補正
5. 画質を良くする処理
 - 5.1 この章について
 - 5.2 ノイズフィルター
 - 5.3 エッジ強調
 - 5.4 トーンカーブ補正
6. 応用編
 - 6.2 線形補間デモザイクの周波数特性
 - 6.1 高度なデモザイク処理
7. まとめ
8. Appendix
 - 8.1. Google Colab の使い方
 - 8.2. RAW 画像の撮影方法

1. はじめに

この本について

この本では、カメラ画像処理・RAW 画像現像の内容を実際の動作レベルで解説し、なるべくスクラッチから Python 上で実行してみる事を目的としています。

そのために、解説する処理は最も重要なものにしぼり、使用するアルゴリズムは一部の例外（デモザイク）を除き、もっとも簡単なものを選びました [1]。

またすべての処理は Google Colab 上で行うことができ、読者の皆さんにはパワフルな PC や特殊なソフトウェアをよういせずとも、ブラウザ上で全ての処理を試してみることができます。

記事の最後ではラズベリーパイのカメラで撮影した RAW 画像からこんな RGB 画像が作れるようになります。

(TODO: 画像を入れる)

対象読者

この本は以下のようないくつかの読者を対象としています。- カメラ内部の画像処理または RAW 現像の内容に興味がある。

また次のような知識があれば、内容を深く理解する助けになります。- Python プログラミングの基本的な内容について知っている。- 高校で学ぶ程度の数学の知識がある。

Colab について

Google Colab (グーグル・コラボ) とは Google が提供するサービスの一つで、ブラウザ上で実行可能な Python のインタラクティブ環境です。

Google Colab を利用することにより、Python の環境を無料でブラウザ上で利用することができます。

詳しくは Google 自身による Colab の解説をご覧ください。

この本で扱うもの

この本で扱う内容は基本的に以下のとおりです。

- 基本的な RAW 現像処理・カメラ画像処理の流れ
- Bayer 画像から RGB 画像出力までの各アルゴリズムのうち、基本的な最低限のものの解説
- 解説した基本的なアルゴリズムの Python による実装と処理例

一部例外はありますが、そういったものは関連した話題として触れられるだけにとどまります。

この記事で扱わないもの

この本で基本的に扱わない物は以下のとおりです。ただし、記事の解説上最低限必要なものについては触れることがあります。

- 3 A (オートフォーカス、オートホワイトバランス、オート露出)などを始めとするカメラコントロールアルゴリズム
- 高度なカメラ画像処理アルゴリズム
- 画像評価、カリブレーション、及びチューニング
- 画像圧縮

環境について

この記事で解説する内容は一般的なものですが、使用した画像ファイルは特定のカメラに依存しています。他のカメラでもわずかな変更で同等の処理ができるとは予想されますが、検証はしていません。

使用カメラ

- Sony Alpha 7 III
- Raspberry Pi Camera v2.1

なお、使用したファイルは Github からダウンロードできるので、これらのカメラをお持ちでなくとも、紹介した処理の内容を実行することは可能です。

実行環境

この内容を再現するには通常の PC 環境に加えて以下の環境等が必用です。

- Google Colab にアクセスできるブラウザ
- 多くのブラウザが対応できると思われます。なお、本書の内容は Chrome によって確かめられています。
- Colab にアクセスできる Google アカウント

必須ではありませんが、次の物があれば、本書の内容を更に深く理解することができます

- PC 上で実行できる Perl 環境 (ExifTool の実行に必用)
- exiftool

2. カメラ画像処理について

この章について

この章ではカメラのしくみや RAW 画像からはじめて、カメラ画像処理や RAW 現像ソフトの中でどのような処理が行われているのかを説明します。

この章を含む全ての内容は Colab ノートブックとして公開しています。この章のノートブックを見るには目次ページから参照するか、以下のリンクを使ってアクセスしてください。

https://colab.research.google.com/github/moizumi99/camera_raw_processing/blob/master/camera_raw_chapter.ipynb

カメラのしくみ

(TODO: 追加)

RAW 画像、RAW ファイルとは？

RAW ファイルや RAW データというのは厳密な定義はないのですが、カメラ処理で RAW というと Bayer フォーマットの画像データを指すことが多いようです。したがって多くの場合、RAW データは Bayer フォーマットの画像データ、RAW ファイルはその RAW データを含んだファイルということになります。

まず前提として、今使われているカメラの画像センサーの殆どは Bayer 配列というものを使ってフルカラーを実現しています。

画像センサーは、碁盤の目状にならんだ小さな光センサーの集まりでできています。一つ一つの光センサーはそのままでは色の違いを認識できません。そこで色を認識するためには、3 原色のうち一色を選択して光センサーにあてて、その光の強度を測定する必要があります。方法としてはまず、分光器を使って光を赤、青、緑に分解して、3 つの画像センサーにあてて、それぞれの色の画像を認識し、その後その 3 枚をあわせることでフルカラーの画像を合成するという方法がありました。これは 3 板方式などとよばれることがあります。この方法は手法的にもわかりやすく、また、余計な処理が含まれないためフルカラーの画像がきれい、といった特徴があり高級ビデオカメラなどで採用されていました。欠点としては分光器と 3 つの画像センサーを搭載するためにサイズが大きくなるという点があります。

これに対して、画像センサー上的一つ一つの光センサーの上に、一部の波長の光だけを通す色フィルターを載せ、各画素が異なる色を取り込むという方法もあります。この方法では 1 枚の画像センサーでフルカラー画像を取り込めるため、3 板方式に対して单板方式とよばれることもあります。3 板方式とくらべた利点としては分光器が不要で 1 枚のセンサーで済むのでサイズが小さい。逆に欠点としては、1 画素につき 1 色の情報しか無いので、フルカラーの画像を再現するには画像処理が必要になる、という点があります。

単版方式の画像センサーの上に載る色フィルターの種類としては、3原色を通す原色フィルターと、3原色の補色（シアン・マゼンダ・イエロー）を通す補色フィルター¹というものがあります。補色フィルターは光の透過率が高いためより明るい画像を得ることができます。それに対して原色フィルターは色の再現度にすぐれています。Bayer配列はこの単版原色フィルター方式のうち最もポピュラーなものです。

こういうわけで Bayer 配列の画像センサーの出力では 1 画素につき一色しか情報をもちません。Bayer 配列のカラーフィルターはこの図左のように、 2×2 ブロックの中に、赤が 1 画素、青が 1 画素、緑が 2 画素ならぶようになっています。緑は対角線上にならびます。緑が 2 画素あるのは、可視光の中でも最も強い光の緑色を使うことで解像度を稼ぐため、という解釈がなされています。Bayer というのはこの配列の発明者の名前です。

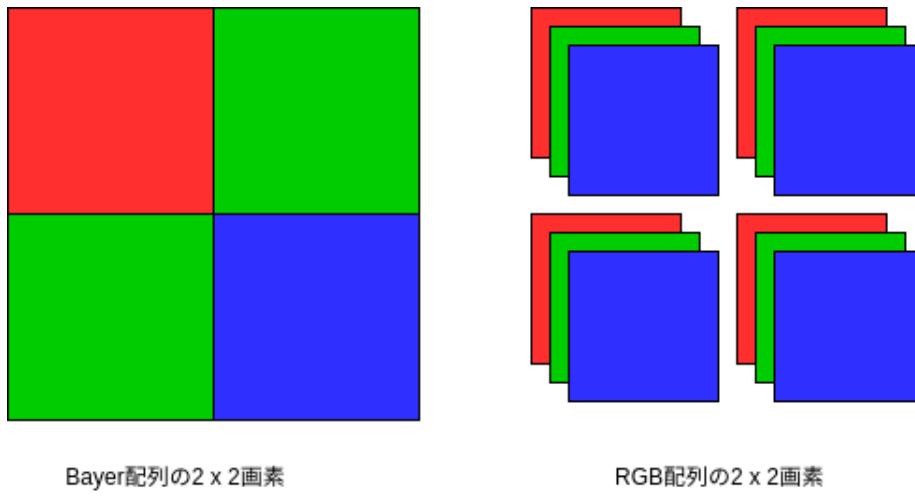


Figure 1: Bayer と RGB 配列

カメラ用センサーでは 2000 年代初頭までは、補色フィルターや 3 板方式もそれなりの割合で使われていたのですが、センサーの性能向上やカメラの小型化と高画質化の流れの中でほとんどが Bayer 方式にかわりました。今では、Sigma の Foveon のような意欲的な例外を除くと、DSLR やスマートフォンで使われているカラー画像センサーの殆どが Bayer 方式を採用しています。したがって、ほとんどのカメラの中では Bayer フォーマットの画像データをセンサーから受取り、フルカラーの画像に変換するという処理が行われている、ということになります。

こういった Bayer フォーマットの画像ファイルは、すなわちセンサーの出力に近いところで出力されたことになり、カメラが処理した JPEG に比べて以下の利点があります。

- ビット数が多い (RGB は通常 8 ビット。Bayer は 10 ビットから 12 ビットが普通。さらに多いものもある)

¹ 実際にはこの他に緑色の画素もあり、 2×2 の 4 画素のパターンになっているのが普通

- 信号が線形（ガンマ補正などがされていない）
- 余計な画像処理がされていない
- 非可逆圧縮がかけられていない（情報のロスがない）

したがって、優秀なソフトウェアを使うことで、カメラが output する JPEG よりもすぐれた画像を手に入れる事ができる「可能性」があります。

逆に欠点としては

- データ量が多い（ビット数が多い。通常非可逆圧縮がされていない）
- 手を加えないと画像が見れない
- 画像フォーマットの情報があまりシェアされていない
- 実際にはどんな処理がすでに行われているのか不透明

などがあります。最後の点に関して言うと、RAW データといつてもセンサー出力をそのままファイルに書き出すことはまずなく、欠陥画素除去など最低限の前処理が行われるのが普通です。しかし、実際にどんな前処理がおこなわれているのかは必ずしも公開されていません。

カメラ画像処理のあらまし

Bayer からフルカラーの画像を作り出す RAW 現像処理・カメラ画像処理のうち、メインになる部分の例はこんな感じになります。²

このうち、最低限必要な処理は、以下のものです。

- ブラックレベル補正
- ホワイトバランス補正（デジタルゲイン補正も含む）
- デモザイク（Bayer からフルカラー画像への変換）
- ガンマ補正

これらがないと、まともに見ることのできる画像を作ることができません。

さらに、最低限の画質を維持するには、通常は、

- 線形性補正
- 欠陥画素補正
- 周辺減光補正
- カラーマトリクス

が必要です。ただし、線形性補正や欠陥画素補正は、カメラが RAW データを出力する前に処理されていることが多いようです。また、センサーの特性がよければ線形性補正やカラーマトリクスの影響は小さいかもしれません。

次に、より良い画質を実現するものとして、

- ノイズ除去
- エッジ強調・テクスチャ補正

²これはあくまで一例です。実際のカメラや RAW 現像ソフト内で行われる処理はメーカー・機種ごとに異なります。

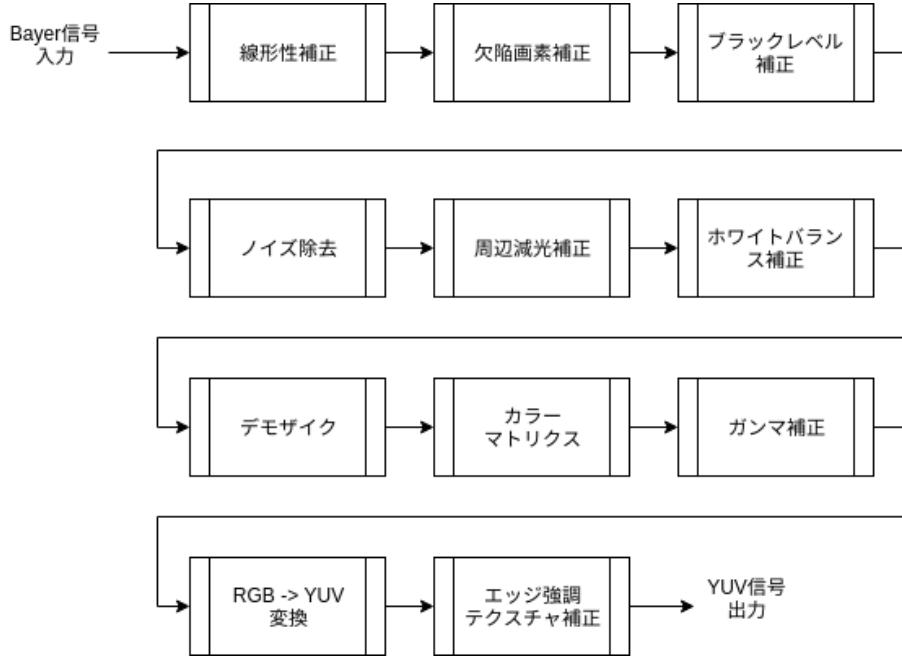


Figure 2: カメラ画像処理パイプライン

があります。RGB->YUV 変換は JPEG や MPEG の画像を作るのには必要ですが、RGB 画像を出力する分にはなくてもかまいません。

この他に、最近のカメラでは更に画質向上させるために

- レンズ収差補正
- レンズ歪補正
- 偽色補正
- グローバル・トーン補正
- ローカル・トーン補正
- 高度なノイズ処理
- 高度な色補正
- ズーム
- マルチフレーム処理

などの処理が行われるのが普通です。今回はベーシックな処理のみとりあげるので、こういった高度な処理は行いません。

結局、今回扱うのは次の部分のみです。

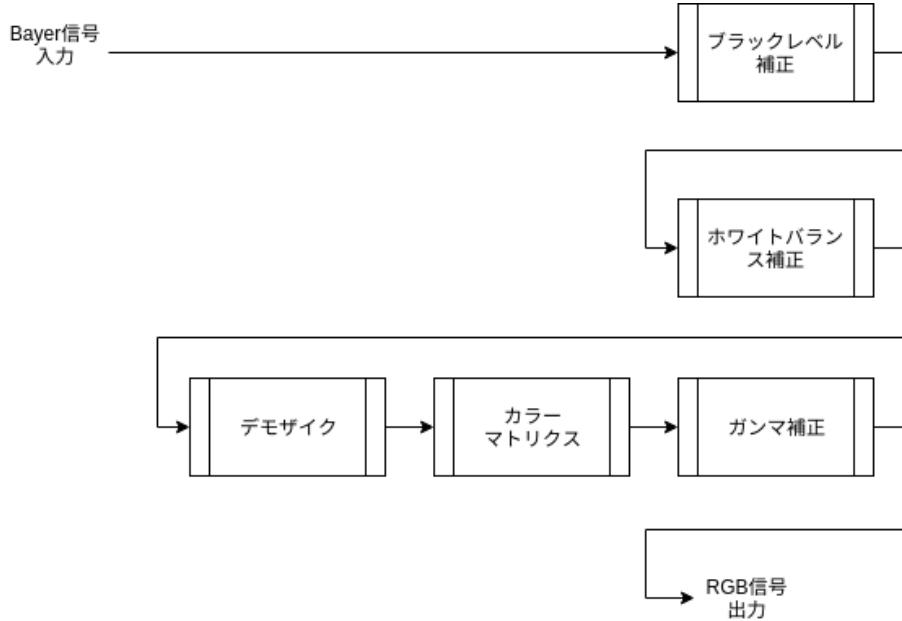


Figure 3: 単純化したカメラ画像処理パイプライン

この章のまとめ

カメラ内部の仕組みと RAW 現像処理・カメラ画像処理を、画像処理の観点から説明しました。

次の章

次はRAW 画像の準備を行います。

3.1 準備と簡易デモザイク

この節について

この節ではまず RAW 画像を準備し、簡易的なデモザイクを行ってみます。

この節の内容は Colab ノートブックとして公開しています。ノートブックを見るには目次ページから参照するか、以下のリンクを使ってアクセスしてください。

https://colab.research.google.com/github/moizumi99/camera_raw_processing/blob/master/camera_raw_ch

RAW 画像の準備

まず RAW 画像を用意します。今回は Sony 7 III で撮影したこの画像を使います。



Figure 4: 最終的な画像

ここで表示している画像は使用する RAW ファイルから作成した RGB 画像 (PNG ファイル) です。

元になる RAW ファイルはこの URL にあります。

https://raw.githubusercontent.com/moizumi99/raw_process/master/sample.ARW

では、colab 上にダウンロードしてみましょう。

```
!if [ ! -f sample.ARW ]; then wget
    https://raw.githubusercontent.com/moizumi99/camera_raw_process/master/sample.ARW;
fi

--2019-01-26 10:54:31--
https://raw.githubusercontent.com/moizumi99/raw_process/master/sample.ARW
Resolving raw.githubusercontent.com (raw.githubusercontent.com)...
151.101.188.133
Connecting to raw.githubusercontent.com
(raw.githubusercontent.com)|151.101.188.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 24746752 (24M) [application/octet-stream]
Saving to: 'sample.ARW.1'
```

```
sample.ARW.1          100%[=====] 23.60M 52.3MB/s
in 0.5s

2019-01-26 10:54:32 (52.3 MB/s) - 'sample.ARW.1' saved
[24746752/24746752]
```

自分で撮影した RAW データを使用する場合は次のコマンド利用してください。

```
# from google.colab import files
# uploaded = files.upload()
```

使いやすいように RAW ファイル名を変数に保存しておきます。

自分でアップロードしたファイルを使用する場合は、ファイル名を対象のファイルに書き換えてください。

```
raw_file = "sample.ARW"
```

RAW 画像の読み込み

まず必用なモジュールをインストールします。

まずインストールするのは rawpy です。

rawpy は python で RAW 画像を操作するためのモジュールです。

<https://pypi.org/project/rawpy/>

rawpy を使うと RAW 画像から RAW 画像データを取り出したり、画像サイズなどのパラメータを読み出したり、また簡易現像することができます。

rawpy の使用法については実際に使う時に説明します。

```
!pip install rawpy
```

```
Requirement already satisfied: rawpy in
  /home/moiz/anaconda3/lib/python3.7/site-packages (0.13.0)
Requirement already satisfied: numpy in
  /home/moiz/anaconda3/lib/python3.7/site-packages (from rawpy)
  (1.15.1)
```

colab ではコマンドの最初に! をつけると、linux コマンドが実行できます。pip は python のモジュール管理用のコマンドです。

次に imageio をインストールします。

imageio は画像の表示やロード・セーブなどを行うモジュールです。

```
!pip install imageio
```

```
Requirement already satisfied: imageio in  
/home/moiz/anaconda3/lib/python3.7/site-packages (2.4.1)
```

次にこれらのモジュールをインポートします。

```
import rawpy, imageio
```

他に必用なモジュールがある場合はその都度 import することにします。

では先程ダウンロードした RAW ファイルを rawpy を使って読み出してみましょう。

imread()は raw データをファイルから読み込む rawpy のメソッドです。

```
raw = rawpy.imread(raw_file)
```

読み込みがうまくいったか確認を兼ねて RAW データの情報を見てみましょう。

まず、画像サイズを確認します。読み込んだ RAW データは、sizes というアトリビュートでサイズ確認ができます。

```
print(raw.sizes)
```

```
ImageSizes(raw_height=4024, raw_width=6048, height=4024, width=6024,  
          top_margin=0, left_margin=0, iheight=4024, iwidth=6024,  
          pixel_aspect=1.0, flip=0)
```

raw_height と raw_width は RAW データのサイズです。この画像のサイズは縦 4024 ライン、横 6048 画素、ということになります。

height と width は、画像処理後の出力画像のサイズです。

他の値については rawpy のページで解説されています。

<https://letmaik.github.io/rawpy/api/rawpy.RawPy.html#rawpy.RawPy.sizes>

この画像を処理しやすくするために、numpy を使用します。

numpy は python 用の数値計算ライブラリーです。行列処理の機能が豊富なので画像処理にも向いています。

まず、numpy を np という名前でインポートします。

```
import numpy as np
```

次に raw 画像データから数値データのみを numpy の配列に読み込みます。

```
raw_array = raw.raw_image
```

raw_image は RAW 画像データを numpy の配列して渡すアトリビュートです。

このままでは 1 次元配列で扱いにくいので、reshape を使って 2 次元配列に変換します。

```
h, w = raw.sizes.raw_height, raw.sizes.raw_width  
raw_array = raw_array.reshape((h, w))
```

これで raw_array は 4024 x 6048 の 2 次元配列になりました。

画像データを表示するコマンド imshow を使って、画像として確認してみましょう。

```
import matplotlib.pyplot as plt
from matplotlib.pyplot import imshow

# raw_array の中のデータをグレースケールで表示します。
imshow(raw_array, cmap='gray')
# 軸を非表示にします。
plt.axis('off')
# 実際に表示します。
plt.show()
```



Figure 5: png

ここで matplotlib は numpy 用描画ライブラリーです。その中で pyplot は各種グラフを表示するモジュールです。ここでは plt という名前でインポートしています。

この節のまとめ

必用なモジュールをインポートして RAW 画像を colab 上に読み込みました。次は簡易デモザイク処理を行います。

3.2 簡易デモザイク処理

この節について

この節では、RAW データの簡易でモザイク処理を行い、画像をフルカラーで表示してみます。

この節の内容は Colab ノートブックとして公開しています。ノートブックを見るには目次ページから参照するか、以下のリンクを使ってアクセスしてください。

https://colab.research.google.com/github/moizumi99/camera_raw_processing/blob/master/camera_raw_ch

準備

まず前節で行ったライブラリのインストールと、モジュールのインポート、画像のダウンロード、及び RAW 画像の読み込みを行います。内容については前節を参照ください。

```
# rawpyとimageioのインストール
!pip install rawpy
!pip install imageio

# rawpy, imageio, numpy, pyplot, imshowのインポート
import rawpy, imageio
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.pyplot import imshow

# 画像をダウンロードします。
!if [ ! -f sample.ARW ]; then wget
    https://raw.githubusercontent.com/moizumi99/camera_raw_process/master/sample.ARW;
fi

#
# 自分で撮影した画像を使用する場合は以下のコメントを取り除きアップロードします。
# from google.colab import files
# uploaded = files.upload()

# RAWファイルの名前。
# アップロードしたファイルを使う場合はその名前に変更。
raw_file  = "sample.ARW"
raw = rawpy.imread(raw_file)
raw_array = raw.raw_image
h, w = raw.sizes.raw_height, raw.sizes.raw_width
raw_array = raw_array.reshape((h, w))
```

```

Requirement already satisfied: rawpy in
    /home/moiz/anaconda3/lib/python3.7/site-packages (0.13.0)
Requirement already satisfied: numpy in
    /home/moiz/anaconda3/lib/python3.7/site-packages (from rawpy)
    (1.15.1)
Requirement already satisfied: imageio in
    /home/moiz/anaconda3/lib/python3.7/site-packages (2.4.1)
--2019-01-26 15:56:51--
    https://raw.githubusercontent.com/moizumi99/raw_process/master/sample.ARW
Resolving raw.githubusercontent.com (raw.githubusercontent.com)...
    151.101.188.133
Connecting to raw.githubusercontent.com
    (raw.githubusercontent.com)|151.101.188.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 24746752 (24M) [application/octet-stream]
Saving to: 'sample.ARW.1'

sample.ARW.1      100%[=====] 23.60M 52.3MB/s
    in 0.5s

2019-01-26 15:56:51 (52.3 MB/s) - 'sample.ARW.1' saved
[24746752/24746752]

```

RAW 画像の確認

読み込んだ RAW 画像を表示してみます。

```

# raw_array の中のデータをグレースケールで表示します。
imshow(raw_array, cmap='gray')
# 軸を非表示にします。
plt.axis('off')
# 実際に表示します。
plt.show()

```

拡大して見てみましょう。

```

# pyplot のコマンド figure() を使って表示サイズを調整。
# ここでは figsize=(8, 8) で、8inch x
# 8inch を指定（ただし実際の表示サイズはディスプレイ解像度に依存）
plt.figure(figsize=(8, 8))

# RAW 画像の中から (1310, 2620) から 60x60 の領域を表示。
imshow(raw_array[1310:1370, 2620:2680], cmap='gray')
# 軸非表示
plt.axis('off')

```



Figure 6: png

```
# 画像表示  
plt.show()
```

明るいところが緑、暗いところが赤や青の画素のはずです。

疑似カラー化

Bayer の赤の部分を赤、青を青、緑を緑で表示してみましょう。

まず、RAW 画像の配列を確認しておきます。

```
print(raw.raw_pattern)
```

```
[[0 1]  
 [3 2]]
```

raw_pattern は rawpy のアトリビュートで、Bayer 配列の 2x2 行列を示します。

ここで、各番号と色の関係は以下のようになっています。カッコ内は略称です

| 番号 | 色 |
|----|--------|
| 0 | 赤 (R) |
| 1 | 緑 (Gr) |

| 番号 | 色 |
|----|--------|
| 2 | 青 (B) |
| 3 | 緑 (Gb) |

ここで緑に Gr と Gb があるのは、赤の行の緑と青の行の緑を区別するためです。カメラ画像処理では両者を区別することが多々あり、両者を Gr と Gb と表す事があります。

両者を区別する必要が無い場合はどちらも G であらわします。

この対応関係を考えると、この画像の各画素の色は、左上から

赤 緑

緑 青

のように並んでいることがわかります。これを図示するところになります。

では、これに対応する RGB 画像を作ってみましょう。

```
# raw_arrayと同じ大きさで、3色のデータを持つnumpyの行列を作る。
# zerosは指定された大きさの0行列を作るコマンド。
raw_color = np.zeros((h, w, 3))

# 偶数列、偶数行の画素は赤なので、赤チャンネル(0)にコピー。
raw_color[0::2, 0::2, 0] = raw_array[0::2, 0::2]
# 奇数列、偶数行の画素は緑なので、緑チャンネル(1)にコピー。
raw_color[0::2, 1::2, 1] = raw_array[0::2, 1::2]
# 偶数列、奇数行の画素は緑なので、緑チャンネル(1)にコピー。
raw_color[1::2, 0::2, 1] = raw_array[1::2, 0::2]
# 奇数列、奇数行の画素は赤なので、青チャンネル(2)にコピー。
raw_color[1::2, 1::2, 2] = raw_array[1::2, 1::2]

# 0から1の範囲にノーマライズ
raw_color[raw_color < 0] = 0
# max()はnumpy行列の最大値を得る関数。
raw_color = raw_color / raw_color.max()
```

これで Bayer に対応する RGB 画像ができたはずです。表示してみましょう。

```
plt.figure(figsize=(8, 8))
# RAW画像表示。
imshow(raw_color)
# 軸非表示
plt.axis('off')
# 画像表示
plt.show()
```

さらに拡大してみます。

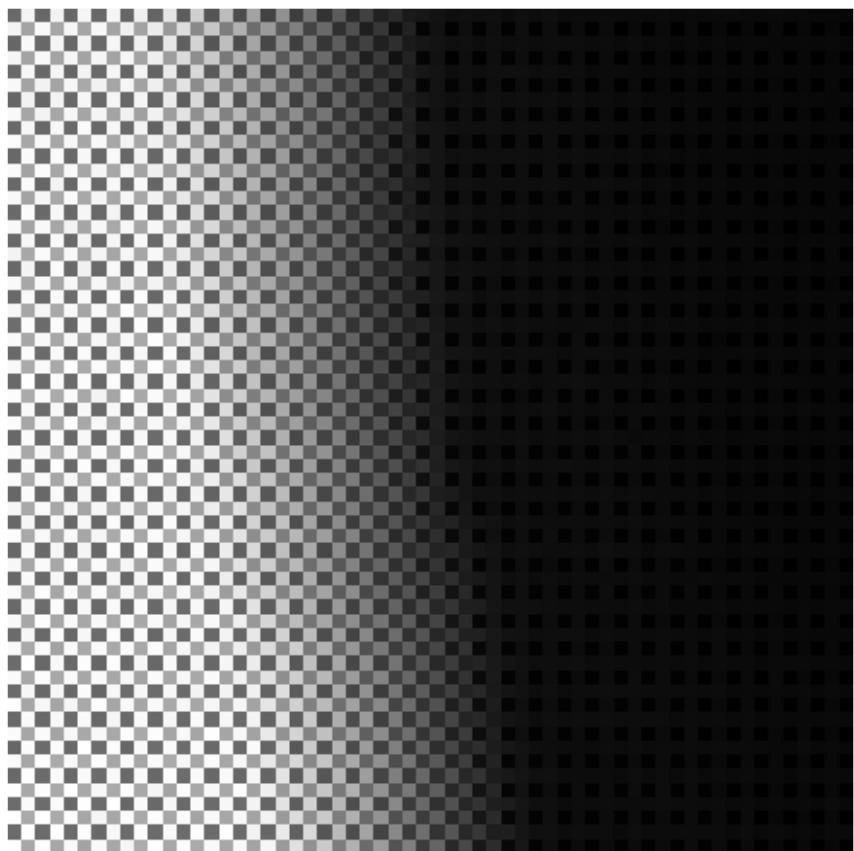


Figure 7: png

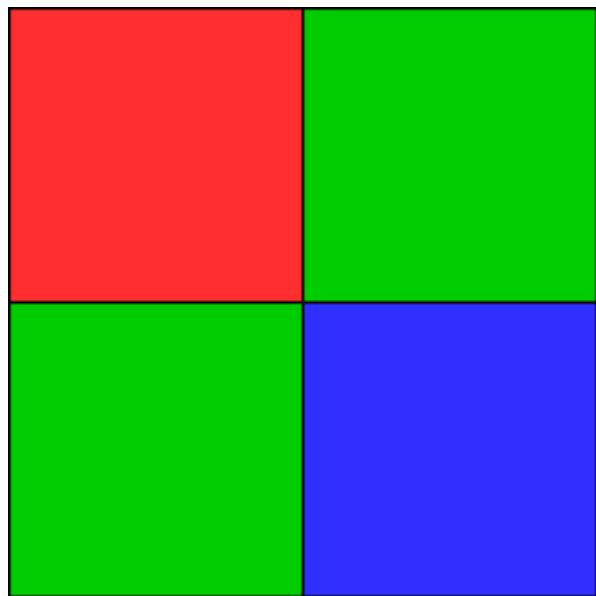


Figure 8: この画像の Bayer 配列

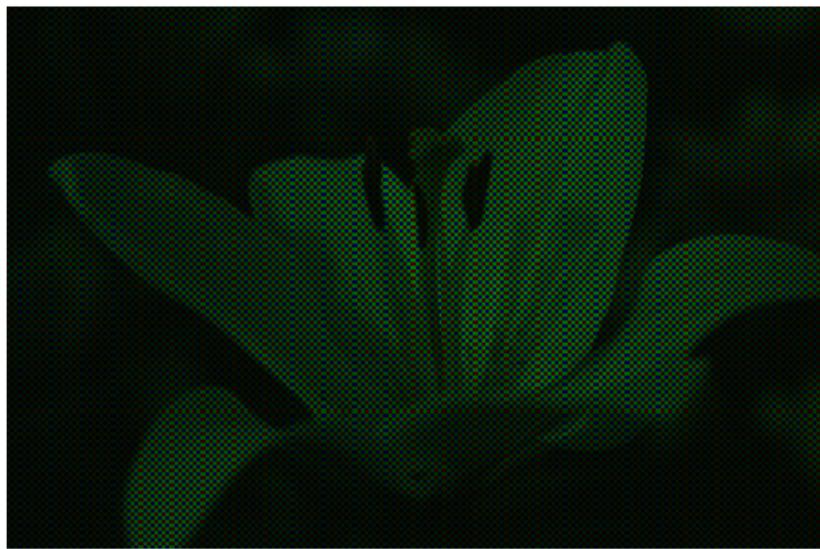


Figure 9: png

```
plt.figure(figsize=(8, 8))
# RAW画像の中から(1310, 2620)から32x32の領域を表示。
imshow(raw_color[1310:1342, 2620:2652])
# 軸非表示
plt.axis('off')
# 画像表示
plt.show()
```

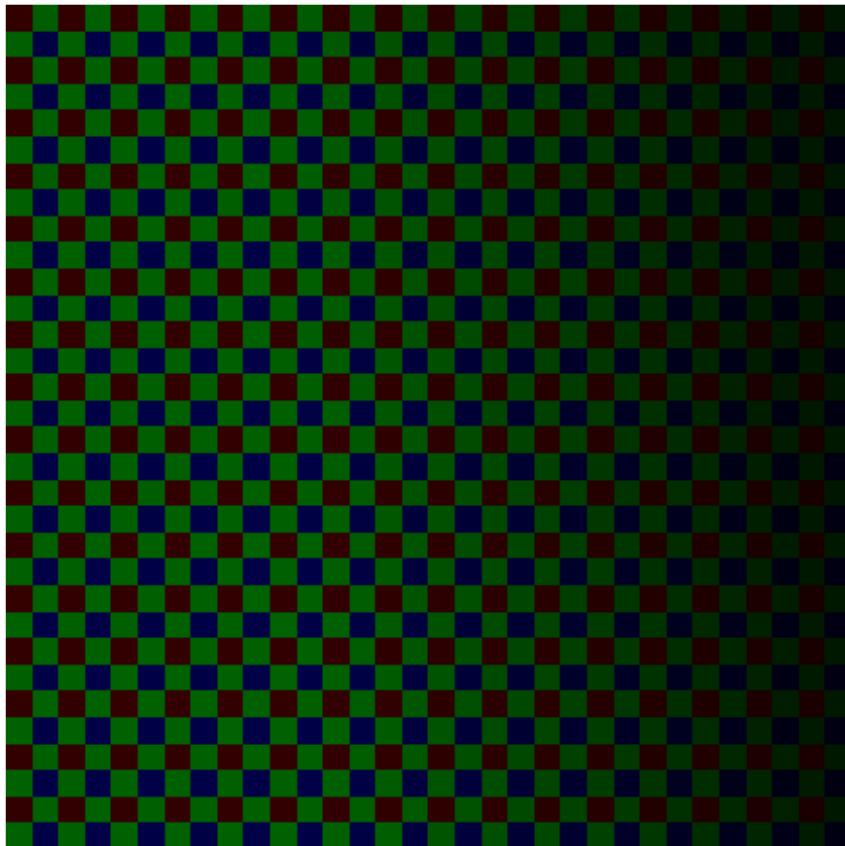


Figure 10: png

これではなんだかよくわかりません。そういうわけで Bayer をフルカラーの RGB に変換する処理が必用になるわけです。

簡易デモザイク処理

それでは Bayer 配列からフルカラーの画像を作ってみましょう。

この処理はデモザイクと呼ばれることが多いです。本来デモザイクはカメラ画像処理プロセス (ISP) の肝になる部分で、画質のうち解像感や、偽色などの不快なアーティファクトなどを大きく左右します。したがって手を抜くべきところではないのですが、今回は簡易処理なので、考えうる限りでもっとも簡単な処理を採用します。

その簡単な処理というのは、3 色の情報を持つ最小単位の 2x2 のブロックから、1 画素のみをとりだす、というものです。

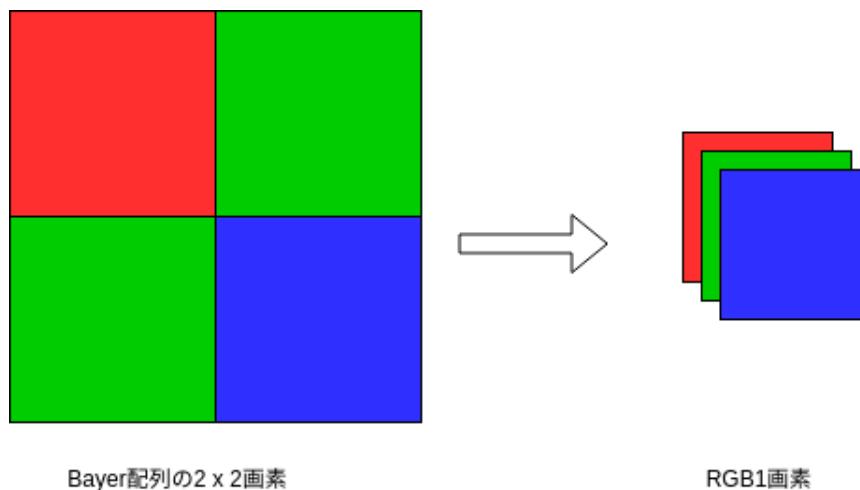


Figure 11: Bayer 配列から RGB 1 画素への簡単な変換

結果として得られる画像サイズは 1/4 になりますが、もとが 24M もあるので、まだ 6M 残っていますので簡易処理としては十分でしょう。

なお、解像度低下とともにないデモザイクアルゴリズムは応用編以降でとりあげます。

では、簡易デモザイク処理してみましょう。なお、2x2 ピクセルの中に 2 画素ある緑は平均値をとります。

今回の処理では 2 つの緑画素は同じものとして扱うので、bayer 配列を 0, 1, 2 で表しておきましょう。

```
bayer_pattern = raw.raw_pattern
# Bayer配列を0, 1, 2, 3から0, 1, 2表記に変更
bayer_pattern[bayer_pattern==3] = 1
# 表示して確認
print(bayer_pattern)
```

```
[[0 1]
 [1 2]]
```

では、 2×2 画素毎に平均をとって RGB 画像を作ります。

```
# RGB画像を容易。サイズは縦横ともRAWデータの半分。
dms_img = np.zeros((h//2, w//2, 3))

# 各画素毎に処理。y, xはRAW画像での位置。
for y in range(0, h, 2):
    for x in range(0, w, 2):
        # bayer_pattern[0, 0]は2x2ブロックの左上の画素の色を示す
        dms_img[y // 2, x // 2, bayer_pattern[0, 0]] += raw_array[y
            + 0, x + 0]
        # bayer_pattern[0, 1]は2x2ブロックの右上の画素の色を示す
        dms_img[y // 2, x // 2, bayer_pattern[0, 1]] += raw_array[y
            + 0, x + 1]
        # bayer_pattern[1, 0]は2x2ブロックの左下の画素の色を示す
        dms_img[y // 2, x // 2, bayer_pattern[1, 0]] += raw_array[y
            + 1, x + 0]
        # bayer_pattern[1, 1]は2x2ブロックの右下の画素の色を示す
        dms_img[y // 2, x // 2, bayer_pattern[1, 1]] += raw_array[y
            + 1, x + 1]
        # 緑画素は2つあるので平均を取る
        dms_img[y // 2, x // 2, 1] /= 2
```

できあがった画像を見てみましょう。

```
# 画像を0と1の間でノーマライズ
dms_img[dms_img < 0] = 0
dms_img = dms_img / dms_img.max()
# 表示
plt.figure(figsize=(8, 8))
imshow(dms_img)
plt.axis('off')
plt.show()
```

このように RGB のフルカラー画像を作ることができました。

まだ色が正しくない、全体的に暗い、などの問題があります。次の節でこのあたりを修正していきます。

処理の高速化

上記のコードは、画像処理とコードの対応がわかりやすいように各画素ごとの処理をループを使って記述してあります。

これは処理の内容はわかりやすいのですが、numpy の高速性を十分に活用しておらず、かなり遅い処理になっています。このコードを numpy の機能を利用して書き直すところのようになります。

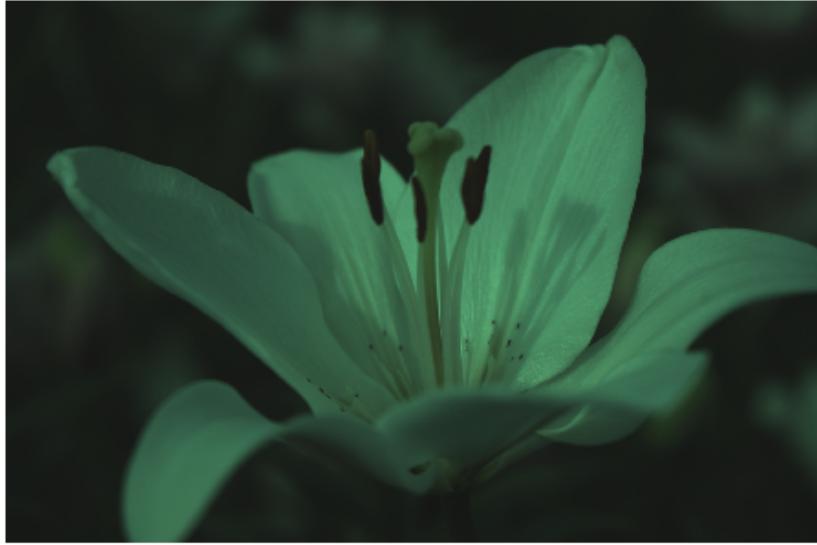


Figure 12: png

```
def simple_demosaic(raw_array, bayer_pattern):
    """
    簡易デモザイク処理を行う。

    Parameters
    -----
    raw_array: numpy array
        入力BayerRAW画像データ
    bayer_pattern: int[2, 2]
        ベイヤーパターン。0:赤、1:緑、2:青、3:緑。

    Returns
    -----
    dms_img: numpy array
        出力RGB画像。サイズは入力の縦横共に1/2。
    """
    height, width = raw_array.shape
    dms_img = np.zeros((height//2, width//2, 3))
    bayer_pattern[bayer_pattern == 3] = 1
    dms_img[:, :, bayer_pattern[0, 0]] = raw_array[0::2, 0::2]
    dms_img[:, :, bayer_pattern[0, 1]] += raw_array[0::2, 1::2]
    dms_img[:, :, bayer_pattern[1, 0]] += raw_array[1::2, 0::2]
```

```
dms_img[:, :, bayer_pattern[1, 1]] += raw_array[1::2, 1::2]
dms_img[:, :, 1] /= 2
return dms_img
```

処理の内容としては最初のループを使ったコードと同じですが、速度は格段に上がっていきます。

同じ画像になったか確認してみましょう。

```
dms_img = simple_demosaic(raw_array, raw.raw_pattern)

# 画像を0と1の間でノーマライズ
dms_img[dms_img < 0] = 0
dms_img = dms_img / dms_img.max()
# 表示
plt.figure(figsize=(8, 8))
imshow(dms_img)
plt.axis('off')
plt.show()
```

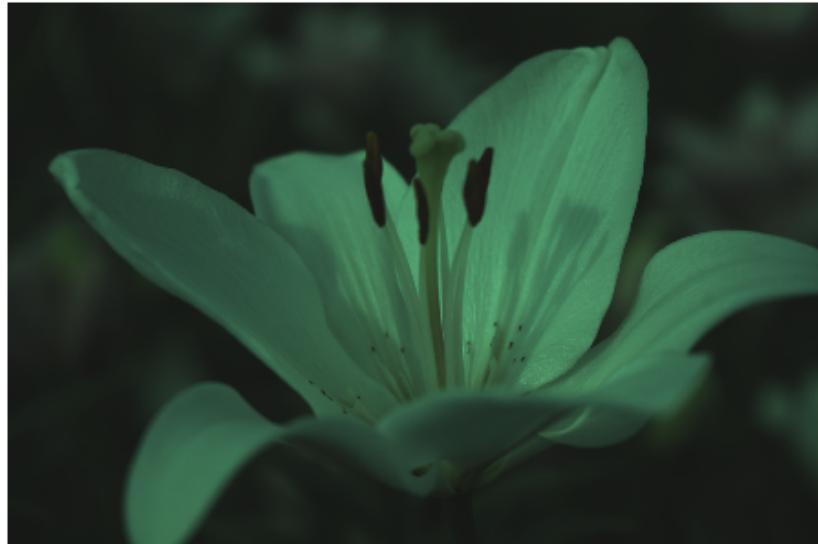


Figure 13: png

同様の画像が出力されたようです。

このsimple_demosaic()関数を含んだモジュールがraw_process.pyとしてgithubにアップロードされています。使用する場合は、

```
!wget https://raw.githubusercontent.com/moizumi99/raw_process/master/raw_process.py
```

としてダウンロードした後、

```
from raw_process import simple_demosaic
```

としてインポートしてください。

この節のまとめ

RAW 画像に対して簡易デモザイク処理を行いました。次はホワイトバランス補正を行います。

3.3 ホワイトバランス補正

この節について

この節では、ホワイトバランス補正を行い画像の色を修正します。

この節の内容は Colab ノートブックとして公開しています。ノートブックを見るには目次ページから参照するか、以下のリンクを使ってアクセスしてください。

https://colab.research.google.com/github/moizumi99/camera_raw_processing/blob/master/camera_raw_ch3.ipynb

準備

まず 3.1 節で行ったライブラリーのインストールと、モジュールのインポート、画像のダウンロード、及び RAW 画像の読み込みを行います。内容については 3.1 節を参照ください

```
# rawpyとimageioのインストール
!pip install rawpy
!pip install imageio

# rawpy, imageio, numpy, pyplot, imshowのインポート
import rawpy, imageio
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.pyplot import imshow

# 前節までに作成したモジュールのダウンロード
!if [ ! -f raw_process.py ]; then wget
    https://raw.githubusercontent.com/moizumi99/camera_raw_process/master/raw_process.py;
fi

from raw_process import simple_demosaic
```

```

# 画像をダウンロードします。
!if [ ! -f sample.ARW ]; then wget
    https://raw.githubusercontent.com/moizumi99/camera_raw_process/master/sample.ARW;
fi

#
# 自分で撮影した画像を使用する場合は以下のコメントを取り除きアップロードします。
# from google.colab import files
# uploaded = files.upload()

# RAWファイルの名前。
# アップロードしたファイルを使う場合はその名前に変更。
raw_file = "sample.ARW"
raw = rawpy.imread(raw_file)
raw_array = raw.raw_image
h, w = raw.sizes.raw_height, raw.sizes.raw_width
raw_array = raw_array.reshape((h, w))

Requirement already satisfied: rawpy in
    /home/moiz/anaconda3/lib/python3.7/site-packages (0.13.0)
Requirement already satisfied: numpy in
    /home/moiz/anaconda3/lib/python3.7/site-packages (from rawpy)
    (1.15.1)
Requirement already satisfied: imageio in
    /home/moiz/anaconda3/lib/python3.7/site-packages (2.4.1)

```

ホワイトバランス補正とは

ホワイトバランス補正とは、センサーの色ごとの感度や、光のスペクトラムなどの影響を除去して、本来の白を白として再現するための処理です。そのためには各色の画素に、別途計算したゲイン値をかけてあげます。

実際のカメラではホワイトバランスの推定は AWB（オートホワイトバランス）と呼ばれる複雑な処理によって行いますが、今回はカメラが撮影時に計算したゲイン値を RAW ファイルから抽出して使います。

ホワイトバランス補正処理

まずはどんなホワイトバランス値かみてみましょう。RAW ファイルの中に記録されたゲインを見てみましょう。

rawpy のアトリビュート camera_whitebalance を使います

```

wb = raw.camera_whitebalance
print(wb)

```

```
[2288.0, 1024.0, 1544.0, 1024.0]
```

これは、赤、緑（赤と同じ行）、青、緑（青と同じ行）のゲインがそれぞれ、2288、1024、1544、1024に比例することをあらわしています。

通常もっとも感度の良い緑に対するゲインを1.0倍として処理するのが普通なので、今回もこのゲインを1024で正規化します。

結局、赤、青に対して、 $2288/1024$ 倍、 $1544/1024$ 倍のゲインを与えるべき事がわかります。

処理してみましょう。

```
# 緑画素のゲインでノーマライズします。
norm = wb[1]

# 元のRAWデータをコピーします。
wb_img = raw_array.copy()
# RAWデータのベイヤーパターン。
bayer_pattern = raw.raw_pattern
for y in range(0, h):
    for x in range(0, w):
        # cは画素に対応する色チャンネル
        c = bayer_pattern[y % 2, x % 2]
        # 画素の色に対応するゲイン
        gain = wb[c] / norm
        # 各画素にゲインをかけ合わせる
        wb_img[y, x] *= gain
```

これでホワイトバランスがそろったかみてみましょう。3.2で使用した簡易デモザイクを使って表示します。

```
# 簡易デモザイク。
# 詳細は3.2節を参照
dms_img = np.zeros((h//2, w//2, 3))
bayer_pattern[bayer_pattern==3] = 1
dms_img[:, :, bayer_pattern[0, 0]] = wb_img[0::2, 0::2]
dms_img[:, :, bayer_pattern[0, 1]] += wb_img[0::2, 1::2]
dms_img[:, :, bayer_pattern[1, 0]] += wb_img[1::2, 0::2]
dms_img[:, :, bayer_pattern[1, 1]] += wb_img[1::2, 1::2]
dms_img[:, :, 1] /= 2

# 画像を0と1の間でノーマライズ
dms_img[dms_img < 0] = 0
dms_img = dms_img / dms_img.max()
# 表示
plt.figure(figsize=(8, 8))
imshow(dms_img)
```

```
plt.axis('off')
plt.show()
```



Figure 14: png

これでだいぶ色がよくなりました。

まだ赤みが強い画像になっています。ブラックレベルの補正がされていないためだと思われます。ブラックレベル補正是次の節で扱います。

処理の高速化

先程扱ったホワイトバランスの処理は、コードの読みやすさを優先したものなので低速です。

numpy の機能を利用して高速化した関数は次のようにになります。

```
def white_balance(raw_array, wb_gain, raw_colors):
    """
    ホワイトバランス補正処理を行う。

    Parameters
    -----
    raw_array: numpy array
        入力BayerRAW画像データ。
    wb_gain: float[4]
```

```

    ホワイトバランスゲイン。
raw_colors: int[h, w]
    RAW画像のカラーチャンネルマトリクス。

Returns
-----
wb_img: numpy array
    出力RAW画像。
"""

norm = wb_gain[1]
gain_matrix = np.zeros(raw_array.shape)
for color in (0, 1, 2, 3):
    gain_matrix[raw_colors == color] = wb_gain[color] / norm
wb_img = raw_array * gain_matrix
return wb_img

```

この関数を使ってホワイトバランス処理を行うにはこのように書きます。

```
wb_img = white_balance(raw_array, raw.camera_whitebalance,
                      raw.raw_colors)
```

簡易デモザイク処理を行って確認しましょう。

```

# 簡易デモザイク。
dms_img = simple_demosaic(wb_img, raw.raw_pattern)

# 画像を0と1の間でノーマライズ
dms_img[dms_img < 0] = 0
dms_img = dms_img / dms_img.max()
# 表示
imshow(dms_img)
plt.axis('off')
plt.show()

```

同様の画像が出力されたようです。

このwhite_balance()関数はraw_process.pyモジュールの一部として github からダウンロードできます。使用する場合は、

```
!wget https://raw.githubusercontent.com/moizumi99/raw_process/master/raw_process.py
```

としてダウンロードした後、

```
from raw_process import white_balance
```

としてインポートしてください。



Figure 15: png

この節のまとめ

この節ではホワイトバランスの調整を行いました。次はブラックレベル補正を行い、色再現を向上します。

3.4 ブラックレベル補正

この節について

この節では、ブラックレベル補正を行い画像の明るさと色を修正します。

この節の内容は Colab ノートブックとして公開しています。ノートブックを見るには目次ページから参照するか、以下のリンクを使ってアクセスしてください。

https://colab.research.google.com/github/moizumi99/camera_raw_processing/blob/master/camera_raw_ch

準備

まず 3.1 節で行ったライブラリーのインストールと、モジュールのインポート、画像のダウンロード、及び RAW 画像の読み込みを行います。内容については各節を参照ください。

```

# rawpyとimageioのインストール
!pip install rawpy;
!pip install imageio;

# rawpy, imageio, numpy, pyplot, imshowのインポート
import rawpy, imageio
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.pyplot import imshow

# 前節までに作成したモジュールのダウンロード
!if [ ! -f raw_process.py ]; then wget
    https://raw.githubusercontent.com/moizumi99/camera_raw_process/master/raw_process.py;
fi

from raw_process import simple_demosaic, white_balance

# 画像をダウンロードします。
!if [ ! -f sample.ARW ]; then wget
    https://raw.githubusercontent.com/moizumi99/camera_raw_process/master/sample.ARW;
fi

#
# 自分で撮影した画像を使用する場合は以下のコメントを取り除きアップロードします。
# from google.colab import files
# uploaded = files.upload()

# RAWファイルの名前。
# アップロードしたファイルを使う場合はその名前に変更。
raw_file = "sample.ARW"
raw = rawpy.imread(raw_file)
raw_array = raw.raw_image
h, w = raw.sizes.raw_height, raw.sizes.raw_width
raw_array = raw_array.reshape((h, w));

Requirement already satisfied: rawpy in
    /home/moiz/anaconda3/lib/python3.7/site-packages (0.13.0)
Requirement already satisfied: numpy in
    /home/moiz/anaconda3/lib/python3.7/site-packages (from rawpy)
    (1.15.1)
Requirement already satisfied: imageio in
    /home/moiz/anaconda3/lib/python3.7/site-packages (2.4.1)

```

ブラックレベル補正とは

RAW データの黒に対応する値は通常 0 より大きくなっています。

これは、センサーの読み出しノイズがマイナスの値を取ることがあるために、画像の値を 0 以上にしてしまうと、ノイズのクリッピングがおきて非常に暗い領域で色ズレが起きてしまうためです。

したがって、正しい画像処理を行うにはブラックレベルを調整して置かなくてはなりません。これをやって置かないと黒が十分黒くない、カスミがかかったような眠い画像になってしまいますし、色もずれてしまいます。

ブラックレベル補正処理

まずはどんなブラックレベル値かみてみましょう。

まず、rawpy のアトリビュートを使ってブラックレベルを確認しましょう。

```
blc = raw.black_level_per_channel  
print(blc)
```

```
[512, 512, 512, 512]
```

これは全チャンネルでブラックレベルは 512 であるという事をしめしています。

今回は全チャンネルで同じ値でしたが、他の RAW ファイルでもこのようになっているとは限りません。各画素ごとのチャンネルに対応した値を引くようにしておきましょう。

```
# ベイダー配列パターンを変数に保存  
bayer_pattern = raw.raw_pattern  
  
# RAWデータを符号付き整数としてコピー。  
blc_raw = raw_array.astype('int')  
# 各画素毎に対応するブラックレベルを参照して引いていく。  
for y in range(0, h, 2):  
    for x in range(0, w, 2):  
        blc_raw[y + 0, x + 0] -= blc[bayer_pattern[0, 0]]  
        blc_raw[y + 0, x + 1] -= blc[bayer_pattern[0, 1]]  
        blc_raw[y + 1, x + 0] -= blc[bayer_pattern[1, 0]]  
        blc_raw[y + 1, x + 1] -= blc[bayer_pattern[1, 1]]
```

処理が正常に行われたか、最大値と最小値を比較しておきましょう。

```
print("ブラックレベル補正前: 最小値=", raw_array.min(), ", 最大値=",  
      raw_array.max())  
print("ブラックレベル補正前: 最小値=", blc_raw.min(), ", 最大値=",  
      blc_raw.max())
```

```
ブラックレベル補正前: 最小値= 0 , 最大値= 8180  
ブラックレベル補正前: 最小値= -512 , 最大値= 7668
```

どうやら正常に処理が行われたようです。

ブラックレベル後の画像の確認

ホワイトバランスと簡易デモザイク処理を行って、ブラックレベルが正常に補正されたか確認しましょう。

```
# 最初に定義したwhite_balance()関数を使って、ホワイトバランス調整。  
wb_img = white_balance(blc_raw, raw.camera_whitebalance,  
                       raw.raw_colors)  
# simple_demosaic()関数を使って、簡易デモザイク処理。  
dms_img = simple_demosaic(wb_img, raw.raw_pattern)
```

では、処理の結果を見てみましょう。

```
# 表示  
plt.figure(figsize=(8, 8))  
#  
    imshowでは画像は0から1.0の値をとる必用があるので、ノーマライズする。  
dms_img[dms_img<0] = 0  
dms_img /= dms_img.max()  
imshow(dms_img)  
plt.axis('off')  
plt.show()
```

だいぶきれいになりました。前回問題だった赤みがかった色も集成されています。

ただし、だいぶ暗い画像になっています。これはガンマ補正がされていないためです。次はガンマ補正をかけてみましょう。

処理の高速化

今回のブラックレベル補正処理もコードの読みやすさを優先して、非常に遅いものになっています。

numpy の機能を利用して高速化すると次のようになります。

```
def black_level_correction(raw_array, blc, bayer_pattern):  
    """  
    ブラックレベル補正処理を行う。  
  
    Parameters  
    -----  
    raw_array: numpy array
```



Figure 16: png

```
    入力BayerRAW画像データ。  
    blc: float[4]  
        各カラーチャンネルごとのブラックレベル。  
    bayer_pattern: int[2, 2]  
        ベイヤーパターン。0:赤、1:緑、2:青、3:緑。  
  
Returns  
-----  
blc_raw: numpy array  
    出力RAW画像。  
"""  
# 符号付き整数として入力画像をコピー  
blc_raw = raw_array.astype('int')  
# 各カラーチャンネル毎にブラックレベルを引く。  
blc_raw[0::2, 0::2] -= blc[bayer_pattern[0, 0]]  
blc_raw[0::2, 1::2] -= blc[bayer_pattern[0, 1]]  
blc_raw[1::2, 0::2] -= blc[bayer_pattern[1, 0]]  
blc_raw[1::2, 1::2] -= blc[bayer_pattern[1, 1]]  
return blc_raw
```

簡易デモザイク処理を行って確認しましょう。

```
# 上記のblack_level_correction関数を使用してブラックレベル補正。  
blc_raw = black_level_correction(raw_array, blc, raw.raw_pattern)
```

```

# 最初に定義したwhite_balance()関数を使って、ホワイトバランス調整。
wb_img = white_balance(blc_raw, raw.camera_whitebalance,
                       raw.raw_colors)
# simple_demosaic()関数を使って、簡易デモザイク処理。
dms_img = simple_demosaic(wb_img, raw.raw_pattern)

# 表示
plt.figure(figsize=(8, 8))
#
    imshowでは画像は0から1.0の値をとる必用があるので、ノーマライズする。
dms_img[dms_img<0] = 0
dms_img /= dms_img.max()
imshow(dms_img)
plt.axis('off')
plt.show()

```

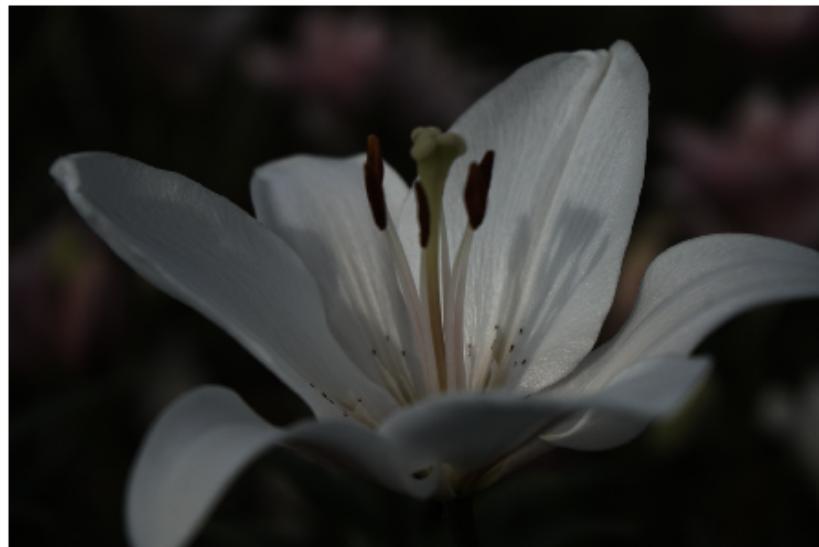


Figure 17: png

同様の画像がoutputされたようです。

このblack_level_correction()関数はraw_process.pyモジュールの一部としてgithubにアップロードされています。使用する場合は、

```

!wget https://raw.githubusercontent.com/moizumi99/raw_process/master/raw_process.py
としてダウンロードした後、

```

```
from raw_process import black_level_correction
```

としてインポートしてください。

まとめ

この節ではブラックレベル補正を行いました。次はガンマ補正を行い、明るさとトーンを補正します。

3.5 ガンマ補正

この節について

この節では、ガンマ補正を行い画像の明るさとトーンを修正します。

この節の内容は Colab ノートブックとして公開しています。ノートブックを見るには目次ページから参照するか、以下のリンクを使ってアクセスしてください。

https://colab.research.google.com/github/moizumi99/camera_raw_processing/blob/master/camera_raw_ch3.ipynb

準備

まずこれまで行ったライブラリーのインストールと、モジュールのインポート、画像のダウンロード、及び RAW 画像の読み込みを行います。内容については各節を参照ください。

```
# rawpyとimageioのインストール
!pip install rawpy;
!pip install imageio;

# rawpy, imageio, numpy, pyplot, imshowのインポート
import rawpy, imageio
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.pyplot import imshow

# 前節までに作成したモジュールのダウンロード
!if [ ! -f raw_process.py ]; then wget
    https://raw.githubusercontent.com/moizumi99/camera_raw_process/master/raw_process.py;
fi

from raw_process import simple_demosaic, white_balance,
    black_level_correction
```

```

# 画像をダウンロードします。
!if [ ! -f sample.ARW ]; then wget
    https://raw.githubusercontent.com/moizumi99/camera_raw_process/master/sample.ARW;
fi

#
    自分で撮影した画像を使用する場合は以下のコメントを取り除きアップロードします。
# from google.colab import files
# uploaded = files.upload()

# RAWファイルの名前。
# アップロードしたファイルを使う場合はその名前に変更。
raw_file = "sample.ARW"
raw = rawpy.imread(raw_file)
raw_array = raw.raw_image
h, w = raw.sizes.raw_height, raw.sizes.raw_width
raw_array = raw_array.reshape((h, w));

Requirement already satisfied: rawpy in
    /home/moiz/anaconda3/lib/python3.7/site-packages (0.13.0)
Requirement already satisfied: numpy in
    /home/moiz/anaconda3/lib/python3.7/site-packages (from rawpy)
    (1.15.1)
Requirement already satisfied: imageio in
    /home/moiz/anaconda3/lib/python3.7/site-packages (2.4.1)

```

ガンマ補正とは

ガンマ補正というのは、もともとテレビがブラウン管だった頃にテレビの出力特性と信号の強度を調整するために使われていたものです。

今でも残っているのは、ガンマ補正による特性が結果的に人間の目の非線形的な感度と相性が良かったからのようです。そんなわけで現在でもディスプレイの輝度は信号に対してブラウン管と似たような信号特性を持って作られており、画像にはガンマ補正をかけておかないと出力は暗い画像になってしまいます。

ガンマ特性自体は次の式で表されます

$$y = x^{2.2}$$

グラフで書くと次のようになります。

```

xs = np.arange(0.0, 1.0, 0.01)
ys = np.power(xs, 2.2)
plt.plot(xs, ys)
plt.show()

```

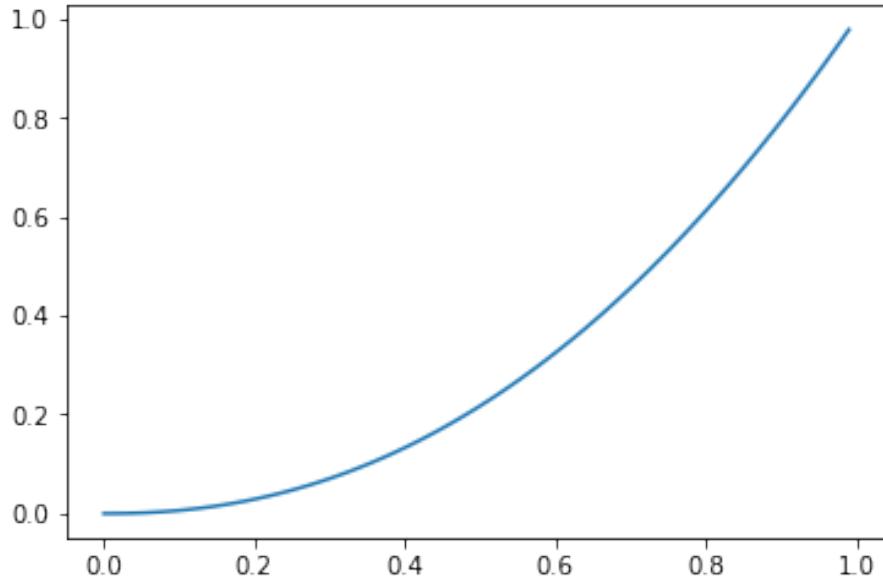


Figure 18: png

ガンマ補正はこれを打ち消す必要があるので、このような式になります。

$$y = x^{\frac{1}{2.2}}$$

グラフはこうなります。

```
xs = np.arange(0.0, 1.0, 0.01)
ys = np.power(xs, 1/2.2)
plt.plot(xs, ys)
plt.show()
```

ガンマ補正処理

それではガンマ補正をかけてみましょう。

ガンマをかけるのはデモザイクまで行った RGB 画像が対象ですので、まずブラックレベル補正、ホワイトバランス補正、簡易でモザイク処理をかけます。

```
# raw_processからインポートしたblack_level_correction関数を使用してブラックレベル補正。
blc_raw = black_level_correction(raw_array,
    raw.black_level_per_channel, raw.raw_pattern)
#
# raw_processからインポートしたwhite_balance()関数を使って、ホワイトバランス調整。
```

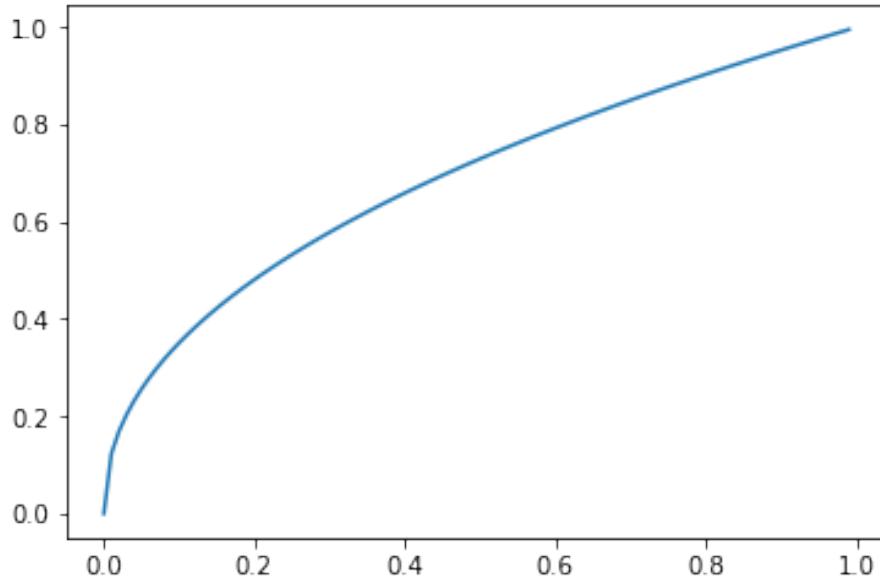


Figure 19: png

```

wb_img = white_balance(blc_raw, raw.camera_whitebalance,
                      raw.raw_colors)
#
      raw_processからインポートしたsimple_demosaic()関数を使って、簡易デモザイク処理。
dms_img = simple_demosaic(wb_img, raw.raw_pattern)

```

画像が正常に処理できているか確認しておきましょう。

```

# 表示
plt.figure(figsize=(8, 8))
#
      imshowでは画像は0から1.0の値をとる必用があるので、ノーマライズする。
dms_img[dms_img<0] = 0
dms_img /= dms_img.max()
imshow(dms_img)
plt.axis('off')
plt.show()

```

デモザイクまでの処理は正常に行われたようなので実際にガンマ補正をかけてみましょう。

```

# デモザイク後の画像をfloatタイプとしてコピー。
gamma_img = dms_img.astype(float)
# ガンマ関数は0-1の範囲で定義されているので、その範囲に正規化する。

```



Figure 20: png

```
gamma_img[gamma_img < 0] = 0
gamma_img = gamma_img/gamma_img.max()
# numpyのpower関数を使って、ガンマ関数を適用。
gamma_img = np.power(gamma_img, 1/2.2)
```

処理の結果を見てみましょう。

```
# 表示
plt.figure(figsize=(8, 8))
imshow(gamma_img)
plt.axis('off')
plt.show()
```

ガンマ補正により明るさが適正になりました。

処理のモジュール化

今回のガンマ補正もモジュールの一部としておきましょう。

```
def gamma_correction(input_img, gamma):
    """
    ガンマ補正処理を行う。
```



Figure 21: png

```
Parameters
-----

```

この`gamma_correction()`関数は`raw_process.py`モジュールの一部としてgithub

にアップロードされています。使用する場合は、

```
!wget https://raw.githubusercontent.com/moizumi99/raw_process/master/raw_process.py  
としてダウンロードした後、  
from raw_process import gamma_correction  
としてインポートしてください。
```

まとめ

この節ではガンマ補正を行いました。これで基本的な処理はすべておわりです。次は画像をきれいにする処理に移ります

4. 重要な処理

4.1 この章について

はじめに

この章ではカメラ画像処理の中でも重要な処理を紹介します。

この章の内容は Colab ノートブックとして公開しています。ノートブックを見るには目次ページから参照するか、以下のリンクを使ってアクセスしてください。

https://colab.research.google.com/github/moizumi99/camera_raw_processing/blob/master/camera_raw_ch

この章で扱う処理について

前章で RAW 画像に最低限の処理を行いフル RGB として表示することができました。処理が単純な割には以外のきれいな画像ができたのではないか？

しかし、これは多くの部分が元の RAW データが良い状態であったという恵まれた条件によるものです。前章で扱った RAW データは、フルサイズセンサーのカメラで非常に明るい屋外で撮影したもので、歪みもノイズも少なく、最小限の画像処理でもそこそこの画質を再現することができました。

しかし、RAW 現像やカメラ画像処理で扱う画像は常にこのような理想的な状態で撮影されるわけではありません。撮影は室内などの暗いところで扱われることも多いですし、センサーもスマートフォン向けを始め非常に小さい物を使う事が多々あります。そういった状況で撮影された RAW 画像に対しては前章で扱ったような RAW 現像だけでは太刀打ちできません。

この章ではそういった通常の RAW 画像を処理する上で最も重要な処理を紹介します。

とりあげるのは以下の処理です。 - デモザイク - 欠陥画素補正 - カラーマトリクス - レンズシェーディング補正

最初に扱うデモザイクでは、出力サイズが入力サイズと同じになる通常のデモザイク処理を取り上げます。

次の欠陥画素補正では、画像センサーにはつきものの欠陥画素を修正する方法を紹介します。

カラーマトリクスは色再現性を向上する処理です。

レンズシェーディング補正是画像の周辺で明るさが低下する周辺減光・レンズシェーディングと呼ばれる現象を補正します。

また、これらの処理の効果を見るために、この章以降では、ラズベリーパイのカメラ(v1.2)で撮影したRAW画像を使用します。

まとめ

この章で扱う内容について概要を説明しました。次はデモザイク処理です。

4.2 線形補間デモザイク

この節について

この節では、画像サイズを変えないデモザイク処理を解説します。

この節の内容は Colab ノートブックとして公開しています。ノートブックを見るには目次ページから参照するか、以下のリンクを使ってアクセスしてください。

https://colab.research.google.com/github/moizumi99/camera_raw_processing/blob/master/camera_raw_chapter2.ipynb

準備

まず3章で行ったライブラリーのインストールと、モジュールのインポートを行います。内容については各節を参照ください。

```
# rawpyとimageioのインストール
!pip install rawpy;
!pip install imageio;

# rawpy, imageio, numpy, pyplot, imshowのインポート
import rawpy, imageio
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.pyplot import imshow
```

```

# 前節までに作成したモジュールのダウンロード
!if [ ! -f raw_process.py ]; then wget
    https://raw.githubusercontent.com/moizumi99/camera_raw_process/master/camera_raw_process.py;
fi

from raw_process import simple_demosaic, white_balance,
black_level_correction, gamma_correction

Requirement already satisfied: rawpy in
/home/moiz/anaconda3/lib/python3.7/site-packages (0.13.0)
Requirement already satisfied: numpy in
/home/moiz/anaconda3/lib/python3.7/site-packages (from rawpy)
(1.15.1)
Requirement already satisfied: imageio in
/home/moiz/anaconda3/lib/python3.7/site-packages (2.4.1)

```

次に画像のダウンロードと読み込みを行います。

今回はラズベリーパイで撮影したこの画像を使用します。

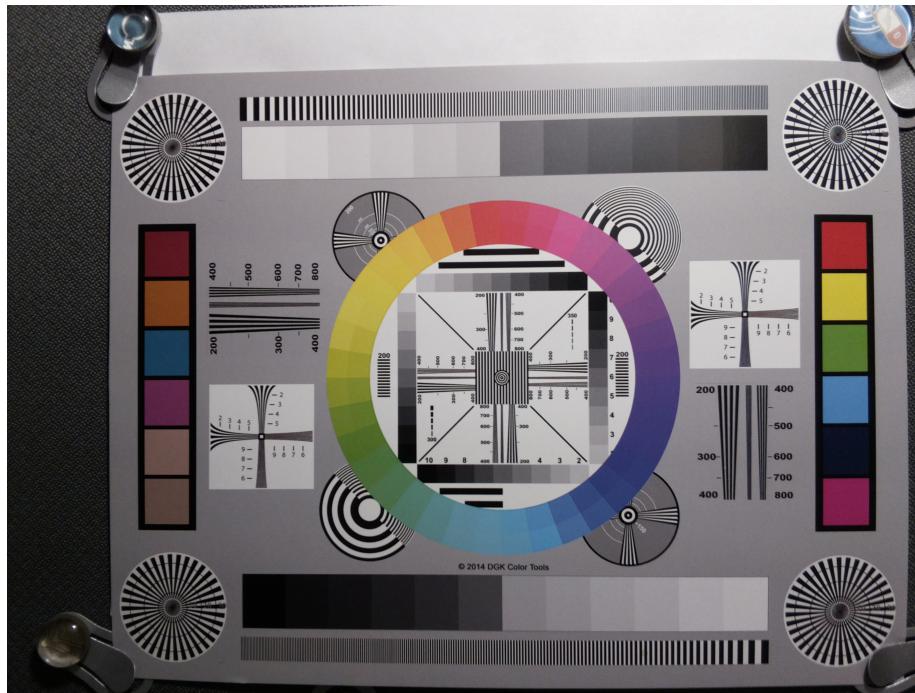


Figure 22: チャート画像

```
# 画像をダウンロードします。
```

```

!if [ ! -f chart.jpg ]; then wget
    https://raw.githubusercontent.com/moizumi99/camera_raw_process/master/chart.jpg;
fi

#
# 自分で撮影した画像を使用する場合は以下のコメントを取り除きアップロードします。
# from google.colab import files
# uploaded = files.upload()

# RAWファイルの名前。
# アップロードしたファイルを使う場合はその名前に変更。
raw_file = "chart.jpg"
raw = rawpy.imread(raw_file)
raw_array = raw.raw_image
h, w = raw.sizes.raw_height, raw.sizes.raw_width
raw_array = raw_array.reshape((h, w));

```

ラズベリーパイによる RAW 画像の撮影方法については付録を参照ください。

簡易デモザイク処理の問題点

前節では、デモザイク処理 (Bayer 配列の画像からフルカラーの画像を作り出す処理) として、簡単的な画像サイズが 1/4 になるものを使いました。単純な処理の割に意外なほどきれいな出力が得られるのですが、いかんせん画像が小さくなるのは問題です。また、出力画像が 1/4 になるので、細かい部分は潰れてしまいます。

この点を確認するために、先程の画像を raw_process モジュールを使って RGB 画像に変換してみましょう。

```

#
# raw_process からインポートした black_level_correction() 関数を使用してブラックレベル補正。
blc_raw = black_level_correction(raw_array,
    raw.black_level_per_channel, raw.raw_pattern)
#
# raw_process からインポートした white_balance() 関数を使って、ホワイトバランス調整。
wb_raw = white_balance(blc_raw, raw.camera_whitebalance,
    raw.raw_colors)
#
# raw_process からインポートした simple_demosaic() 関数を使って、簡易デモザイク処理。
dms_img = simple_demosaic(wb_raw, raw.raw_pattern)
#
# raw_process からインポートした gamma_correction() 関数を使って、ガンマ補正。
gmm_img = gamma_correction(dms_img, 2.2)

```

表示してみます。

```
# サイズ設定
plt.figure(figsize=(16, 8))
imshow(gmm_img)
plt.axis('off')
plt.show()
```

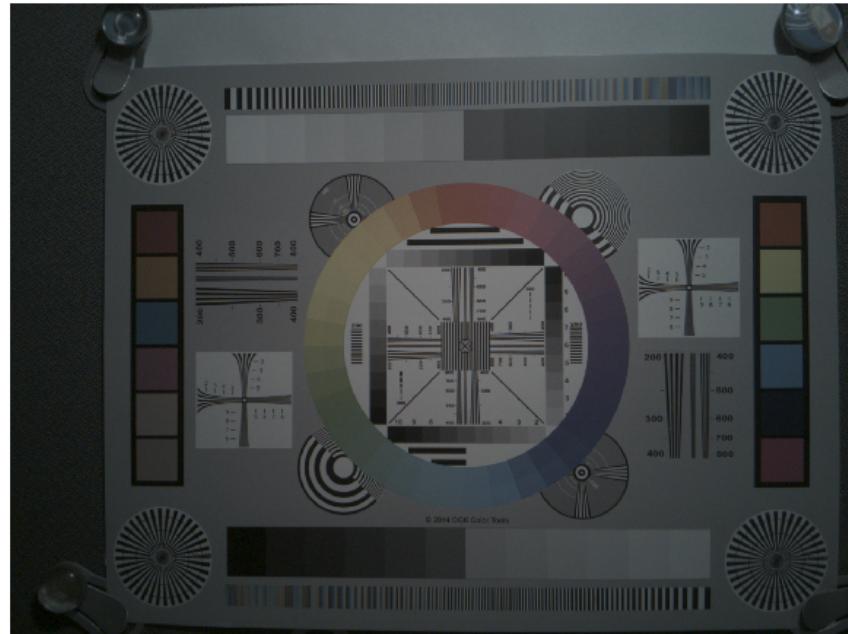


Figure 23: png

この画像のサイズと RAW データのサイズを見てみましょう。

```
print("現像後のサイズ = ", gmm_img.shape)
print("RAWデータのサイズ = ", raw_array.shape)
```

```
現像後のサイズ = (1232, 1640, 3)
RAWデータのサイズ = (2464, 3280)
```

この画像の大きさは縦 1232 ライン、横 1640 画素であることがわかります。それに対して元の RAW 画像のサイズは縦 2464 ライン、横 3280 画素です。ちょうど 2 分の 1 ずつになっているのがわかります。

最初に表示した JPEG 画像と大きさを合わせて並べてみましょう。まずは JPEG 画像を numpy の array として読み込みます。

```
# matplotlibのモジュールimageを使ってJPEG画像を読み込みます。
```

```

from matplotlib import image
jpg_img = image.imread("chart.jpg")
# 0-1の範囲で正規化します
jpg_img = jpg_img / jpg_img.max()
# 画像サイズを取得します
h2, w2, c = jpg_img.shape

```

次に、この JPEG から作ったデータと、先程簡易 RAW 現像したデータ同じ numpy array に代入してみます。

```

# JPEG画像の横幅の倍の幅を持つnumpy arrayを作成
two_img = np.zeros((h2, w2 * 2, c))
# numpy arrayの右半分にJPEG画像のデータをはめこむ。
two_img[0:, w2:, :] = jpg_img
# 左半分に簡易RAW現像したデータをはめこむ。
two_img[h//4:h//4+h//2, w//4:w//4+w//2, :] = gmm_img

```

ならべてた画像データを表示してみましょう。

```

plt.figure(figsize=(16, 8))
imshow(two_img)
plt.axis('off')
plt.show()

```



Figure 24: png

色合いが違う、明るさが違う、という点は無視しても、サイズの違いは明白です。

次は拡大してみましょう。こんどは表示サイズが同じになるように調整します。

```

# 表示サイズ設定
plt.figure(figsize=(16, 8))

# まずは簡易RAW現像したファイルの表示。
# 縦1列、横2列に表示領域を設定。

```

```

# そのうち 1つ目に画像表示。
plt.subplot(1, 2, 1)
# 簡易RAW現像した画像の表示したい範囲。
y1, x1 = 740, 835
dy1, dx1 = 100, 100
# 選択した範囲を表示
imshow(gmm_img[y1:y1+dy1, x1:x1+dx1])
plt.axis('off')

# 次にJPEG画像の表示。
# 縦 1列、横 2列のうち 2つめに表示。
plt.subplot(1, 2, 2)
# 画像位置を簡易RAW現像のものに合わせる
y2, x2 = y1 * 2, x1 * 2
dy2, dx2 = dy1 * 2, dx1 * 2
imshow(jpg_img[y2:y2+dy2, x2:x2+dx2])
plt.axis('off')

# 実際に表示。
plt.show()

```

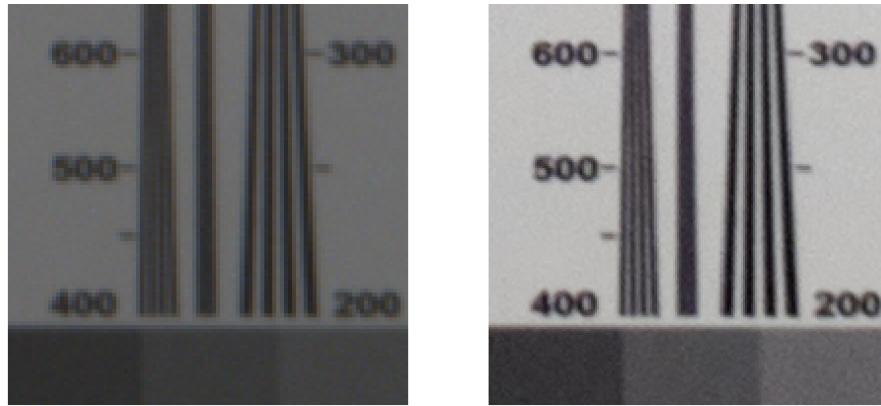


Figure 25: png

明るさやコントラストの違いがまっさきに目につきますが、それは次回以降考えましょう。解像度に注目すると、意外なほど健闘はしているのですが、縦のラインの分解能が低かったりする点がわかると思います。このあたりは簡易デモザイクによる画像サイズの低下の影響があるといえるでしょう。

現代のカメラ内部のデモザイクはかなり高度な処理をしているはずなので、右側の JPEG 画像並みの解像度を得るのは難しいと思いますが、せめてもとの画像サイズを取り戻せるような処理を導入してみましょう。

線形補完法

デモザイクアルゴリズムの中で、縮小する方法の次に簡単なのは、線形補間法です。線形補間といふものらしいですが、ようするに、距離に応じて間の値をとるわけです。たとえば、緑の画素ならこうなります。

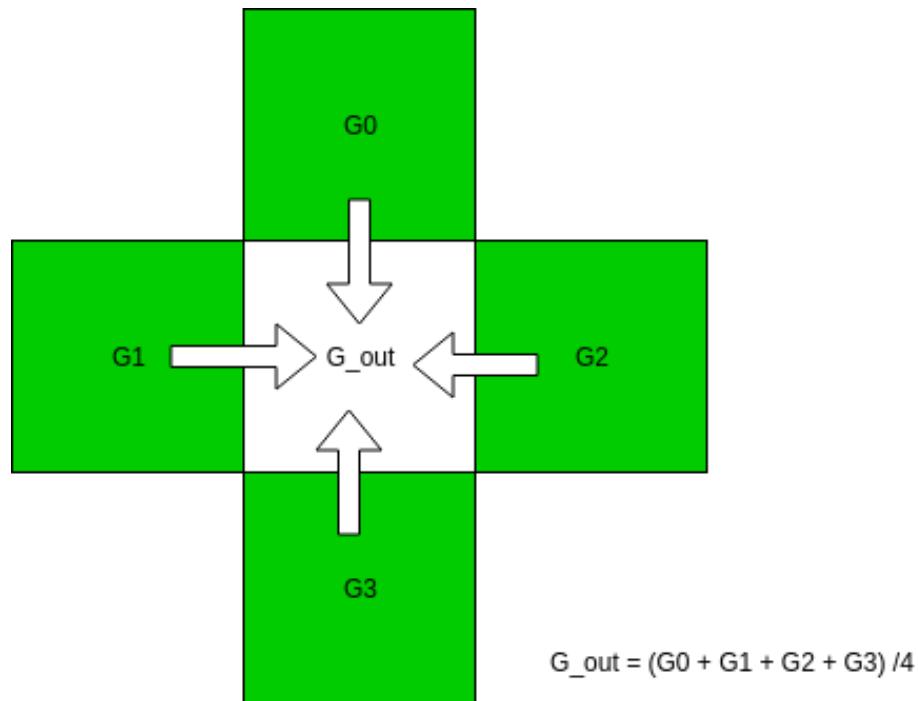


Figure 26: 緑画像の線形補間

赤の画素ではこうです。

青の画素でも、赤の場合と同じような補完を行います。

では実際やってみましょう。

```
# 画像のヘリの部分で折り返すためのヘルパー関数
def mirror(x, min, max):
    if x < min:
        return min - x
    elif x >= max:
        return 2 * max - x - 2
    else:
        return x

dms_img = np.zeros((h, w, 3))
```

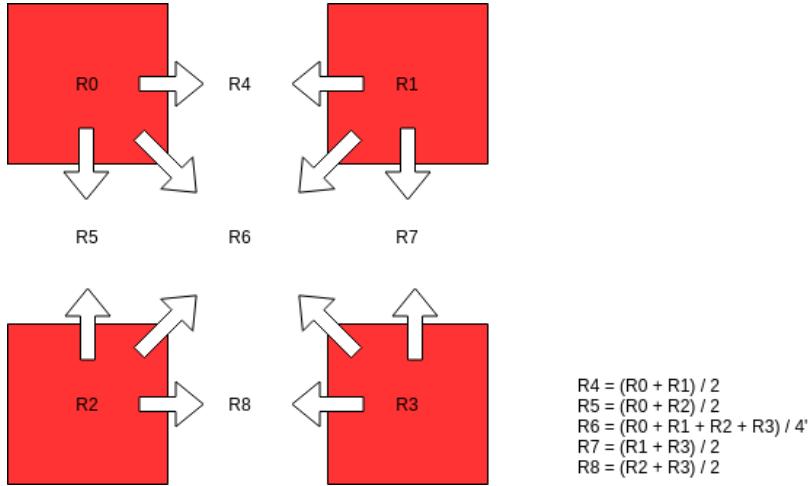


Figure 27: 赤画像の線形補間

```

bayer_pattern = raw.raw_pattern
for y in range(0, h):
    for x in range(0, w):
        color = bayer_pattern[y % 2, x % 2]
        y0 = mirror(y-1, 0, h)
        y1 = mirror(y+1, 0, h)
        x0 = mirror(x-1, 0, w)
        x1 = mirror(x+1, 0, w)
        if color == 0:
            dms_img[y, x, 0] = wb_raw[y, x]
            dms_img[y, x, 1] = (wb_raw[y0, x] + wb_raw[y, x0] +
                wb_raw[y, x1] + wb_raw[y1, x])/4
            dms_img[y, x, 2] = (wb_raw[y0, x0] + wb_raw[y0, x1] +
                wb_raw[y1, x0] + wb_raw[y1, x1])/4
        elif color == 1:
            dms_img[y, x, 0] = (wb_raw[y, x0] + wb_raw[y, x1]) / 2
            dms_img[y, x, 1] = wb_raw[y, x]
            dms_img[y, x, 2] = (wb_raw[y0, x] + wb_raw[y1, x]) / 2
        elif color == 2:
            dms_img[y, x, 0] = (wb_raw[y0, x0] + wb_raw[y0, x1] +
                wb_raw[y1, x0] + wb_raw[y1, x1])/4
            dms_img[y, x, 1] = (wb_raw[y0, x] + wb_raw[y, x0] +
                wb_raw[y, x1] + wb_raw[y1, x])/4
            dms_img[y, x, 2] = wb_raw[y, x]
        else:
            dms_img[y, x, 0] = (wb_raw[y0, x] + wb_raw[y1, x]) / 2
            dms_img[y, x, 1] = wb_raw[y, x]
    
```

```
dms_img[y, x, 2] = (wb_raw[y, x0] + wb_raw[y, x1]) / 2
```

画像のサイズを確認します。

```
print(dms_img.shape)
```

```
(2464, 3280, 3)
```

元の RAW 画像の同じサイズになっているようです。

画像を確認してみましょう。まずは残っているガンマ補正処理を行います。

```
gmm_full_img = gamma_correction(dms_img, 2.2)
```

表示します。

```
# サイズ設定
plt.figure(figsize=(16, 8))
imshow(gmm_full_img)
plt.axis('off')
plt.show()
```

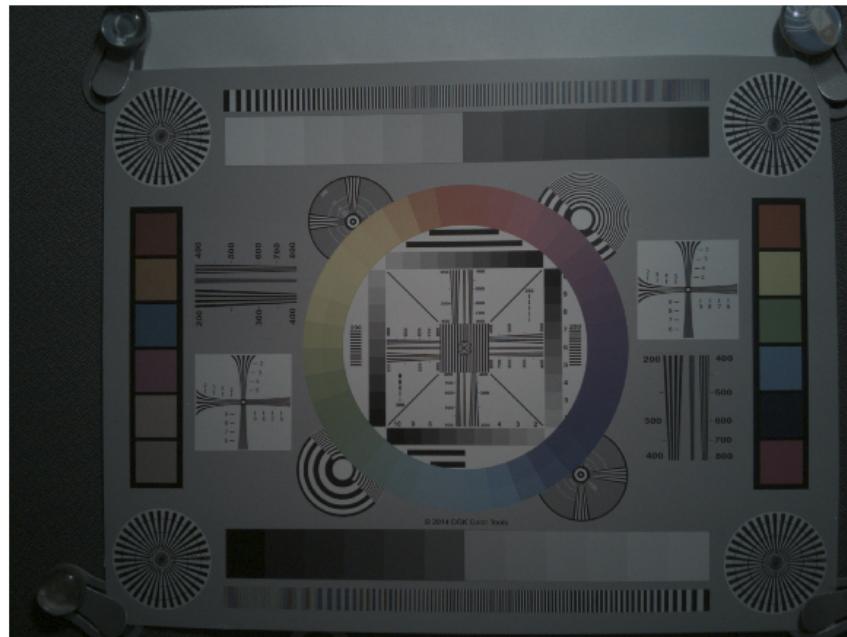


Figure 28: png

それでは、JPEG 画像と並べて表示してみましょう。

```

# 表示サイズ設定
plt.figure(figsize=(16, 8))

# まずは簡易RAW現像したファイルの描画。
plt.subplot(1, 2, 1)
y1, x1 = 740, 835
dy1, dx1 = 100, 100
imshow(gmm_img[y1:y1+dy1, x1:x1+dx1])
plt.axis('off')

# 今回RAW現像した画像の描画。
plt.subplot(1, 2, 2)
y2, x2 = y1 * 2, x1 * 2
dy2, dx2 = dy1 * 2, dx1 * 2
imshow(gmm_full_img[y2:y2+dy2, x2:x2+dx2])
plt.axis('off')

# 実際に表示。
plt.show()

```

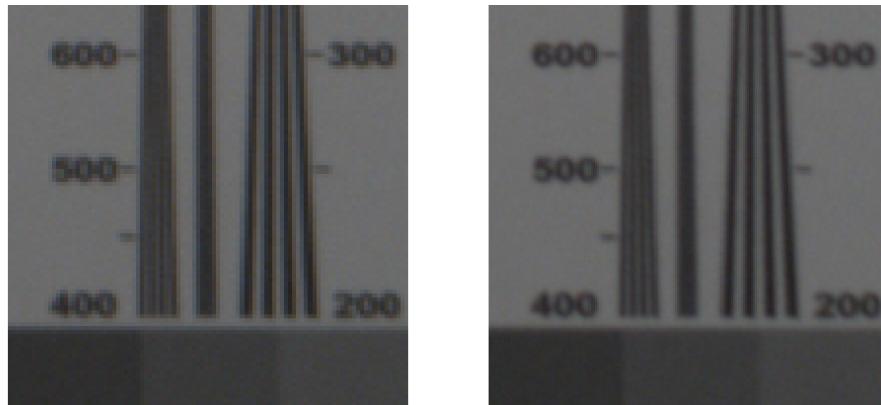


Figure 29: png

右側の線形補間した結果では、縦のラインがより細い部分まで分解されていることがわかります。とりあえずは成功としましょう。

なお、より高性能なデモザイクは6章の応用編でとりあげます。

処理の高速化

今回のデモザイクもコードの読みやすさを優先させてあります。高速化しておきましょう。

高速化に当たっては、数値計算ライブラリ scipy の signal モジュールを使います。

```
from scipy import signal

def demosaic(raw_array, raw_colors):
    """
    線形補間でデモザイクを行う

    Parameters
    -----
    raw_array: numpy array
        入力BayerRAW画像データ。
    raw_colors: numpy array
        RAW画像のカラーチャンネルマトリクス。
        通常Raupyのraw_colorsを用いて与える。

    Returns
    -----
    dms_img: numpy array
        出力RAW画像。
    """
    h, w = raw_array.shape
    dms_img = np.zeros((h, w, 3))

    # 緑画素の処理
    # 元のRAW画像から緑画素だけ抜き出す
    green = raw_array.copy()
    green[(raw_colors == 0) | (raw_colors == 2)] = 0
    # 緑画素の線形補間フィルター
    # [[0, 1, 0]]
    # [1, 4, 1]
    # [0, 1, 0]] / 4.0
    g_flt = np.array([[0, 1, 0], [1, 4, 1], [0, 1, 0]]) / 4.0
    # フィルターの適用
    # boundary='symm': 画像のヘリで折り返す。
    # mode='same': 出力画像サイズは入力画像と同じ。
    dms_img[:, :, 1] = signal.convolve2d(green, g_flt,
                                         boundary='symm', mode='same')

    # 元のRAW画像から赤画素だけ抜き出す
    red = raw_array.copy()
    red[raw_colors != 0] = 0
    # 赤画素の線形補間フィルター
    # [[1, 2, 1]]
    # [2, 4, 2]
    # [1, 2, 1]] / 4.0
```

```

rb_flt = np.array([[1 / 4, 1 / 2, 1 / 4], [1 / 2, 1, 1 / 2], [1
    / 4, 1 / 2, 1 / 4]])
# フィルターの適用
dms_img[:, :, 0] = signal.convolve2d(red, rb_flt,
    boundary='symm', mode='same')

# 元のRAW画像から青画素だけ抜き出す
blue = raw_array.copy()
blue[raw_colors != 2] = 0
# 青画素の線形補間フィルターは赤と共に通
# フィルターの適用
dms_img[:, :, 2] = signal.convolve2d(blue, rb_flt,
    boundary='symm', mode='same')
return dms_img

```

ここで使った convolve2d は、線形フィルターを画像などの2次元データに畳み込む処理です。

上記の場合は、入力画素の周辺 3x3 画素を取り出し、その一つ一つの画素とフィルターの値を掛け合わせた上で合計し出力するという処理をしています。

例えばこのコードの場合、緑画素なら、*gin*、*gout* を緑画素の入力、出力とすると

$$g = \frac{1}{4} \begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

$$gout_{x,y} = \sum_{i=-1}^{+1} \sum_{j=-1}^{+1} gin_{x+i,y+i} g_{i,j}$$

という処理を行います。これで入力画素が緑画素の場合、上下左右には緑画素がないので入力画素がそのまま出力されます。そうでない場合は、上下左右の緑がその平均値が出力されます。

結果として先程行った1画素毎に処理するプログラムと同じ結果がえられます。試してみましょう。

```

dms2_img = demosaic(wb_raw, raw.raw_colors)
print(dms2_img.shape)

```

```
(2464, 3280, 3)
```

処理が終了して画像サイズが元の RAW データと同じであることがわかります。

ガンマ補正を行って表示してみましょう。

```

gmm2_full_img = gamma_correction(dms2_img, 2.2)
# サイズ設定
plt.figure(figsize=(16, 8))
imshow(gmm2_full_img)
plt.axis('off')
plt.show()

```

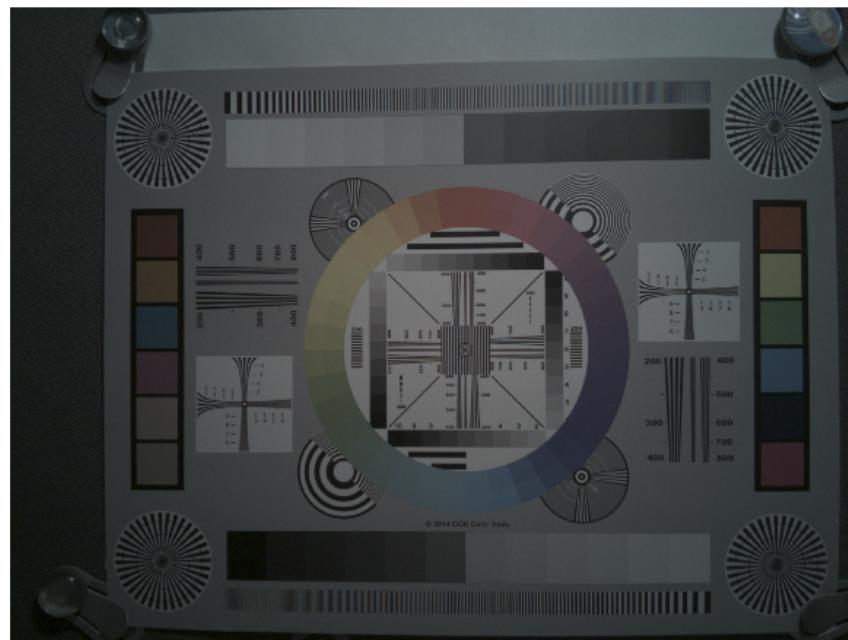


Figure 30: png

同様の画像が出力されたようです。

このdemosaic()関数はraw_process.pyモジュールの一部としてgithubにアップロードされています。使用する場合は、

```
!wget https://raw.githubusercontent.com/moizumi99/raw_process/master/raw_process.py
としてダウンロードした後、
```

```
from raw_process import demosaic
```

としてインポートしてください。

まとめ

この節では線形補間によるデモザイク処理を行いました。次は欠陥画素補正を行います。