

Python と Colab で作る - RAW 現像ソフト作成入門

注: この一連の記事は現時点未完成です (2/3/2019 時点)

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
 - 4.2 線形補間デモザイク
 - 4.3 欠陥画素補正
 - 4.4 カラーマトリクス補正
 - 4.5 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
 - 6.1 線形補間デモザイクの周波数特性
 - 6.2 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]。

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)

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。

全てのコードへのリンクはこちらの目次ページからたどることができます。

対象読者

この本は以下のような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

次の章

次はカメラ画像処理について簡単に解説します。

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、画像センサー上の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板方式に対して単板方式とよばれることもあります。3版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。

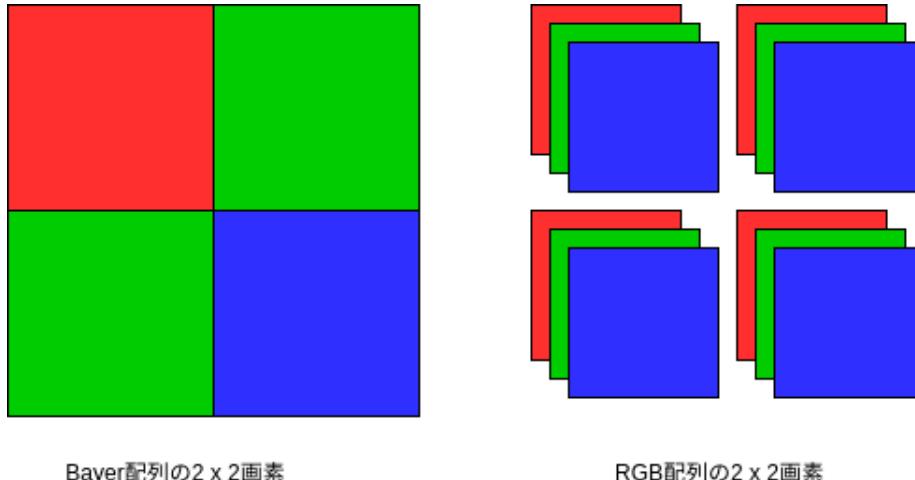


Figure 1: Bayer と RGB 配列

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。

¹Bilateral Filtering for Gray and Color Images, C. Tomasi and R. Manduchi, Proceedings of the 1998 IEEE International Conference on Computer Vision, pp. 839–846, 1998

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 に比べて以下のような利点があります。

- 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)
- 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、カメラが出力する 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もなく、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

²ただし、処理の内容が一見わかりにくくなるというトレードオフがあります。これが理由で先程はベタ書きの処理を紹介しました。

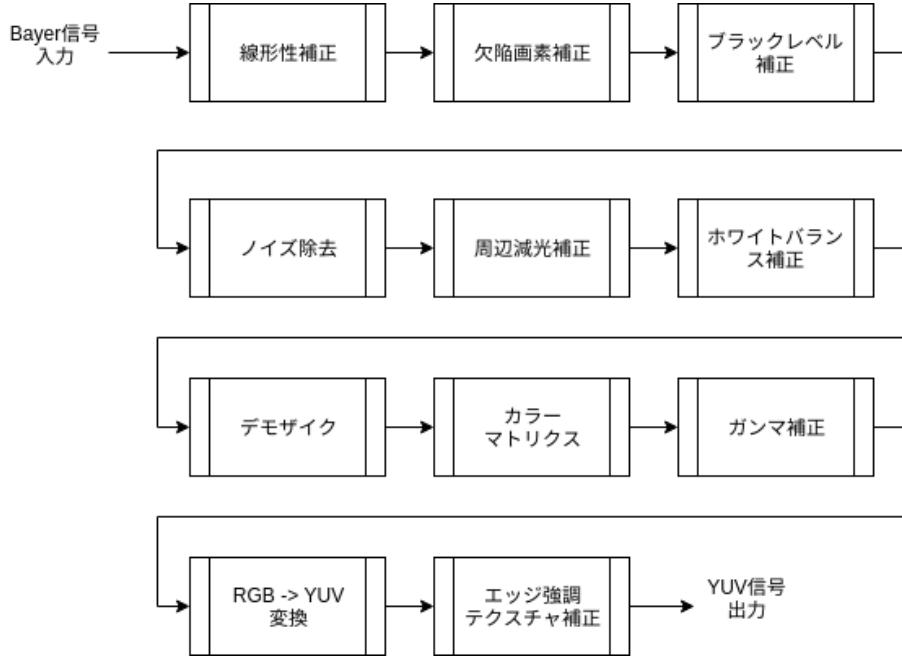


Figure 2: カメラ画像処理パイプライン

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

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ともかまいません。

この他に、最近のカメラでは更に画質を向上させるために

- 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。

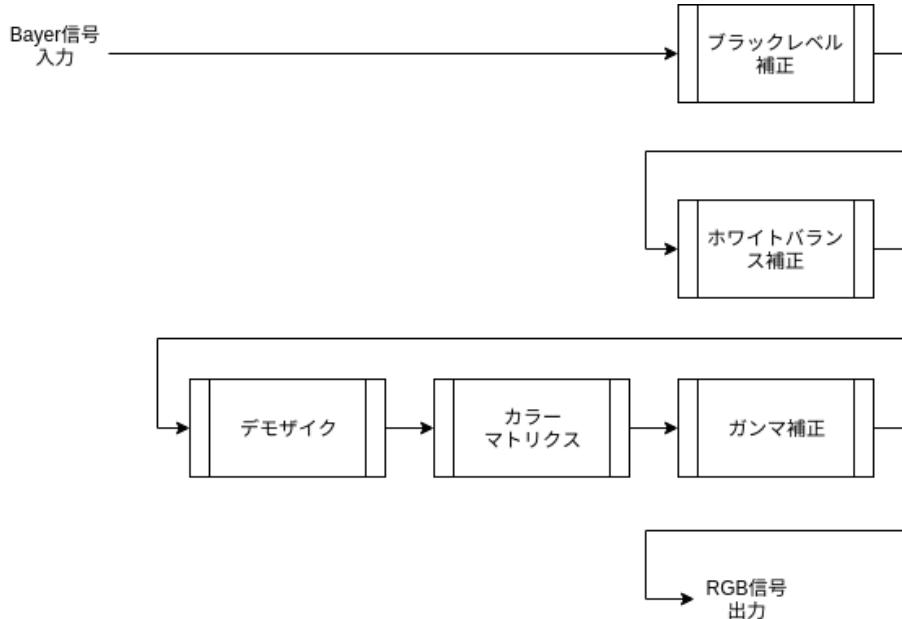


Figure 3: 単純化したカメラ画像処理パイプライン

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
```

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、また簡易現像を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 の配列として渡すアトリビュートです。

縦横の大きさを取得しておきます。

```
h, w = raw_array.shape  
print(h, w)
```

```
4024 6048
```

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。

まずキャプション用に日本語フォントを用意します。

```
# 日本語フォントをインストール  
!apt -y install fonts-ipafont-gothic  
  
# 画像表示用ライブラリ matplotlib のインポート。
```

```

import matplotlib.pyplot as plt
# 日本語フォントを設定
plt.rcParams['font.family'] = 'IPAPGothic'

E: Could not open lock file /var/lib/dpkg/lock-frontend - open (13:
    Permission denied)
E: Unable to acquire the dpkg frontend lock
(/var/lib/dpkg/lock-frontend), are you root?

```

実際に画像を表示します。

```

# 画像表示サイズを設定。figsizeの中身は横サイズ、縦サイズ。
#
# 単位はインチだが実際の表示サイズはディスプレイ解像度によって異なる。
plt.figure(figsize=(8, 6))
# raw_arrayの中のデータをグレースケールで表示します。
plt.imshow(raw_array, cmap='gray')
# 軸を非表示にします。
plt.axis('off')
# 画像タイトルの設定
plt.title(u"Bayer画像をそのまま表示")
# 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。

もし日本語のタイトルが文字化けしている場合は、もし日本語が文字化けしている場合は！ `rm /content/.cache/matplotlib/fontList.json`を実行して、Runtime->Restart Runtime で再実行してみてください。

```
# もし日本語が文字化けしている場合次の行の#を削除して実行。  
# ! rm /content/.cache/matplotlib/fontList.json  
# その後、Runtime->Restart and run allで再実行
```

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  
  
# 日本語フォントの設定
```

```

!apt -y install fonts-ipafont-gothic
plt.rcParams['font.family'] = 'IPAPGothic'
# もし日本語が文字化けしている場合 `! rm
# /content/.cache/matplotlib/fontList.json` を実行して、
# Runtime->Restart Runtimeで再実行

# 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_array.shape

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)
E: Could not open lock file /var/lib/dpkg/lock-frontend - open (13:
    Permission denied)
E: Unable to acquire the dpkg frontend lock
    (/var/lib/dpkg/lock-frontend), are you root?

```

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。
plt.imshow(raw_array, cmap='gray')
plt.axis('off')
plt.title(u"RAW画像の確認")
plt.show()

```



Figure 6: png

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。
plt.imshow(raw_array[1310:1370, 2620:2680], cmap='gray')
plt.axis('off')
plt.title(u"RAW画像の拡大表示")
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 の画素と色の関係を直感的に理解するために、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)
```

RAW画像の拡大表示

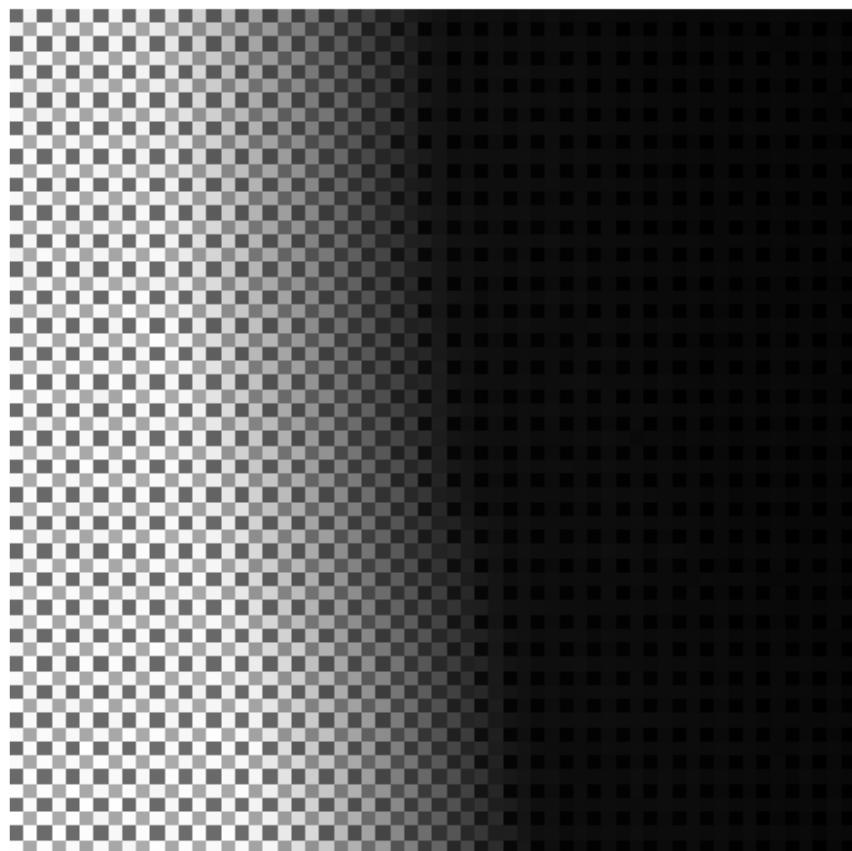


Figure 7: png

```
[[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)
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。

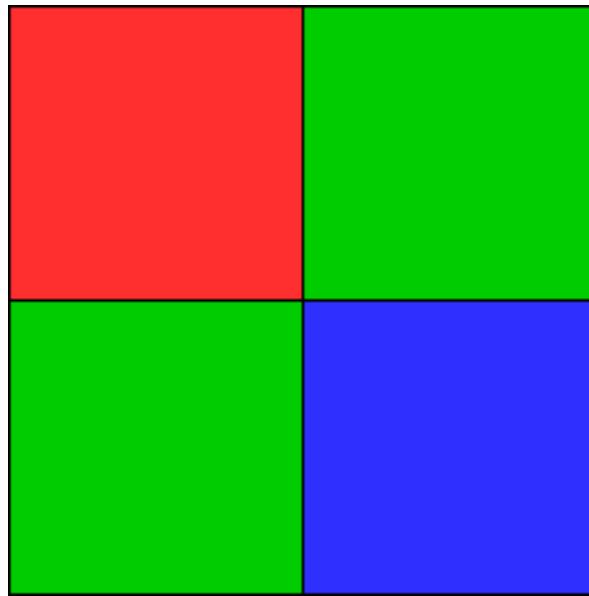


Figure 8: この画像の Bayer 配列

では、これに対応する RGB 画像を作つてみましょう。

```
# 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。

```
# RAW画像に色を割り振ったものを表示。
plt.figure(figsize=(8, 8))
plt.imshow(raw_color)
plt.axis('off')
plt.title(u"RAW画像の各画素に色を割り当てたもの")
plt.show()
```

さらに拡大してみます。

```
plt.figure(figsize=(8, 8))
# RAW画像の中から(1310, 2620)から32x32の領域を表示。
plt.imshow(raw_color[1310:1342, 2620:2652])
plt.axis('off')
plt.title(u"RAW画像の各画素に色を割り当てたものを拡大表示")
plt.show()
```

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つてみましょう。

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



Figure 9: png

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 2×2 のブロックから、1画素のみをとりだす、というものです。

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、解像度低下をともなわ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、 2×2 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]]
```

RAW画像の各画素に色を割り当てたものを拡大表示

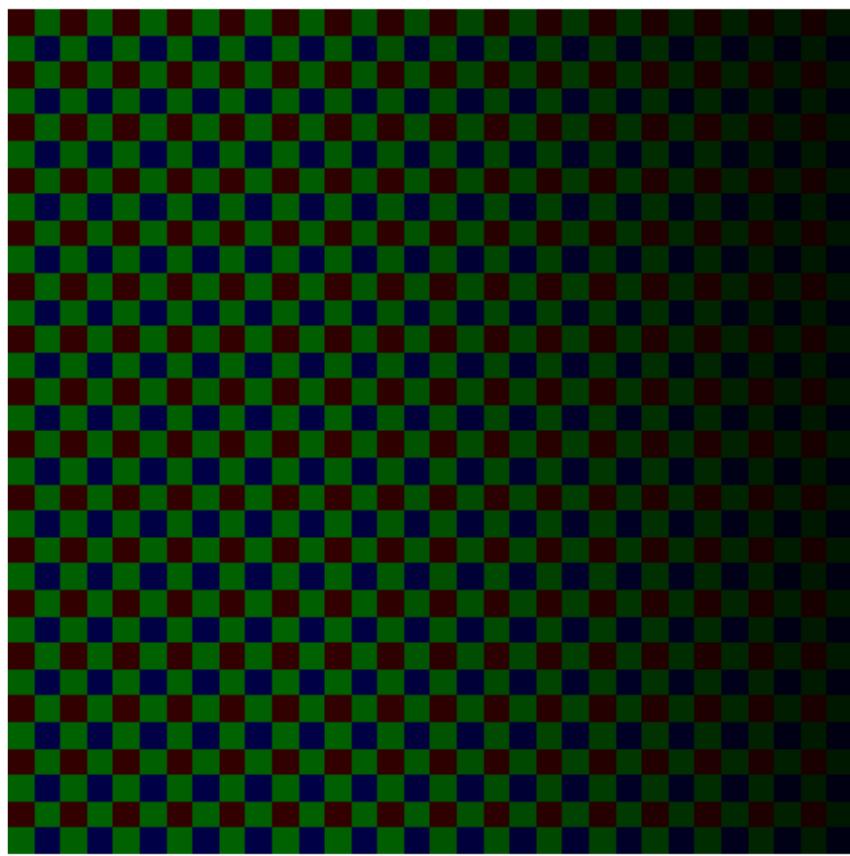


Figure 10: png

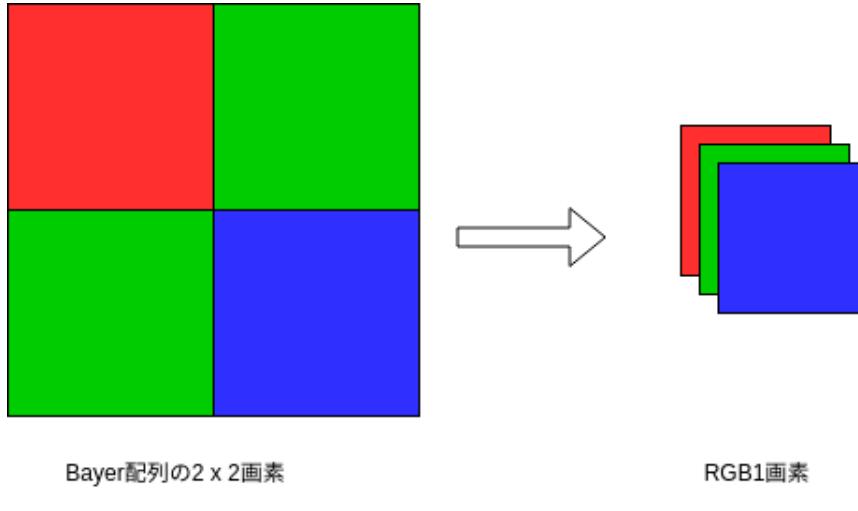


Figure 11: Bayer 配列から RGB 1 画素への簡単な変換

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))
plt.imshow(dms_img)
plt.axis('off')
plt.title(u"簡易デモザイク")
plt.show()
```



Figure 12: png

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います。

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になります。

```
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))
plt.imshow(dms_img)
plt.axis('off')
plt.title(u"simple_demosaic関数の出力")
plt.show()
```

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
```



Figure 13: png

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_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 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

# 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

# 日本語フォントの設定
!apt -y install fonts-ipafont-gothic
plt.rcParams['font.family'] = 'IPAPGothic'
# もし日本語が文字化けしている場合`! rm
    /content/.cache/matplotlib/fontList.json`を実行して、
# Runtime->Restart Runtimeで再実行

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_array.shape
```

```
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)
E: Could not open lock file /var/lib/dpkg/lock-frontend - open (13:
  Permission denied)
E: Unable to acquire the dpkg frontend lock
  (/var/lib/dpkg/lock-frontend), are you root?
```

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、 $1544/1024$ 倍のゲインを与えればよい事がわかります。

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))
plt.imshow(dms_img)
plt.axis('off')
plt.title(u"ホワイトバランス後の画像")
plt.show()

```

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。

ホワイトバランス後の画像



Figure 14: png

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
plt.imshow(dms_img)
plt.axis('off')
plt.title(u"white_balance関数を使った出力")
plt.show()
```

white_balance関数を使った出力

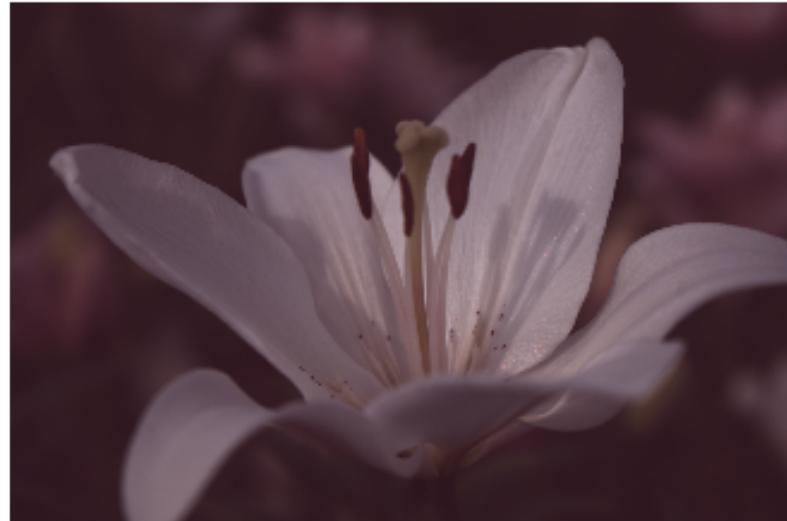


Figure 15: png

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。

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

# 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

# 日本語フォントの設定
!apt -y install fonts-ipafont-gothic
plt.rcParams['font.family'] = 'IPAPGothic'
# もし日本語が文字化けしている場合 `! rm
    /content/.cache/matplotlib/fontList.json` を実行して、
# Runtime->Restart Runtimeで再実行

# 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_array.shape

```

```

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)
E: Could not open lock file /var/lib/dpkg/lock-frontend - open (13:
    Permission denied)
E: Unable to acquire the dpkg frontend lock
    (/var/lib/dpkg/lock-frontend), are you root?

```

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お

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()
plt.imshow(dms_img)
plt.axis('off')
plt.title(u"ブラックレベル補正後の画像")
plt.show()
```

ブラックレベル補正後の画像

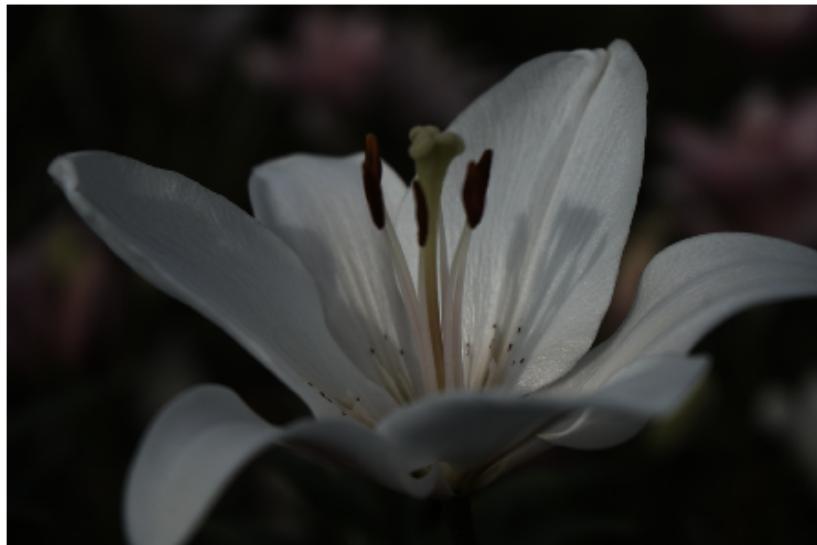


Figure 16: png

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。
次の節で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
        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()
plt.imshow(dms_img)
plt.axis('off')
plt.title(u"black_level_correction関数を使った出力")
plt.show()

```



Figure 17: png

同様の画像が出力されたようです。

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

# 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

# 日本語フォントの設定
!apt -y install fonts-ipafont-gothic
plt.rcParams['font.family'] = 'IPAGothic'
# もし日本語が文字化けしている場合`! rm
    /content/.cache/matplotlib/fontList.json`を実行して、
```

```

# Runtime->Restart Runtimeで再実行

# 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_array.shape

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)
E: Could not open lock file /var/lib/dpkg/lock-frontend - open (13:
    Permission denied)
E: Unable to acquire the dpkg frontend lock
    (/var/lib/dpkg/lock-frontend), are you root?

```

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.title(u"ガンマカーブ")
plt.show()
```

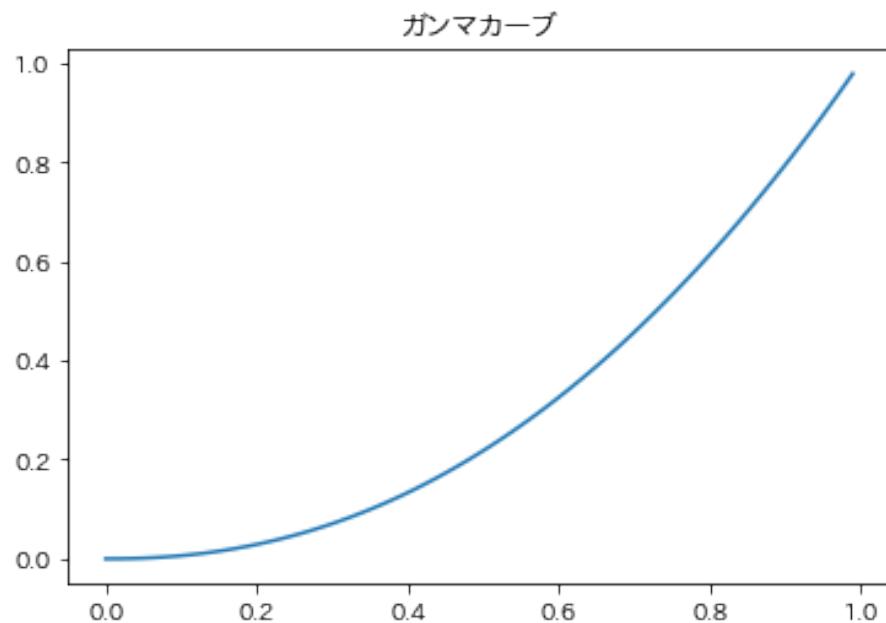


Figure 18: png

モニターなどの出力の強さは入力に対してこのような特性になるので、入力の方をこれに合わせて調整しておく必用があります。これがガンマ補正です。

ガンマ補正是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.title(u"ガンマ補正カーブ")
plt.show()
```

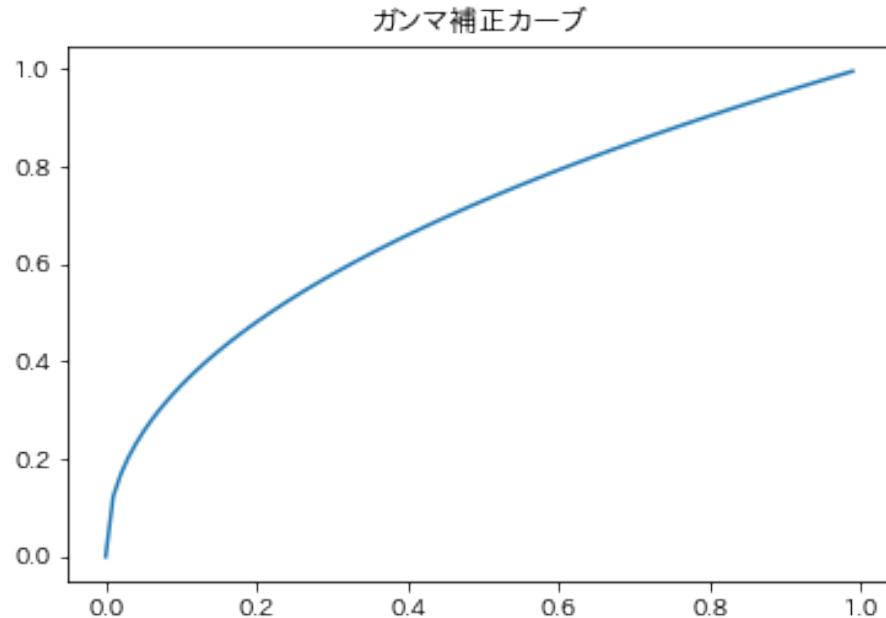


Figure 19: png

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。
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。
img = dms_img.copy()
img[img<0] = 0
img /= img.max()
img[img>1] = 1
plt.imshow(img)
plt.axis('off')
plt.title(u"ガンマ補正前")
plt.show()

```



Figure 20: png

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。
gamma_img[gamma_img < 0] = 0
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))
plt.imshow(gamma_img)
plt.axis('off')
plt.title(u"ガンマ補正後")
plt.show()
```



Figure 21: png

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。

    Parameters
    -----
    input_img: numpy array [h, w, 3]
        入力RGB画像データ。
        0-1の範囲で正規化されていること。
    
```

```

gamma: float
    ガンマ補正值。通常は2.2。

Returns
-----
gamma_img: numpy array [h, 2, 3]
    出力RGB画像。
"""

# デモザイク後の画像をfloatタイプとしてコピー。
gamma_img = input_img.copy()
gamma_img[gamma_img < 0] = 0
gamma_img[gamma_img > 1] = 1.0
# numpyのpower関数を使って、ガンマ関数を適用。
gamma_img = np.power(gamma_img, 1/gamma)
return gamma_img

```

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。次の章ではその他の重要な処理を扱います。

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でしょうか？

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。そういった画像データにはさまざまな理想的でない特性があり、そういうものは、各種の補正処理を行わないと最終的な画質は低品質になってしまいます。

また、前章で扱ったデモザイクは簡易的なもので出力画像が入力画像の 4 分の 1 の大きさになるという重大な問題があります。これも解決しなくてはなりません。

この章ではそういう通常の RAW 画像を処理する上で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 - カラーマトリクス補正
- 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。

カラーマトリクス補正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_ch

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

# 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

# 日本語フォントの設定
!apt -y install fonts-ipafont-gothic
plt.rcParams['font.family'] = 'IPAPGothic'
# もし日本語が文字化けしている場合`! rm
    /content/.cache/matplotlib/fontList.json`を実行して、
# Runtime->Restart Runtimeで再実行

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)
E: Could not open lock file /var/lib/dpkg/lock-frontend - open (13:
    Permission denied)
E: Unable to acquire the dpkg frontend lock
    (/var/lib/dpkg/lock-frontend), are you root?
```

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。

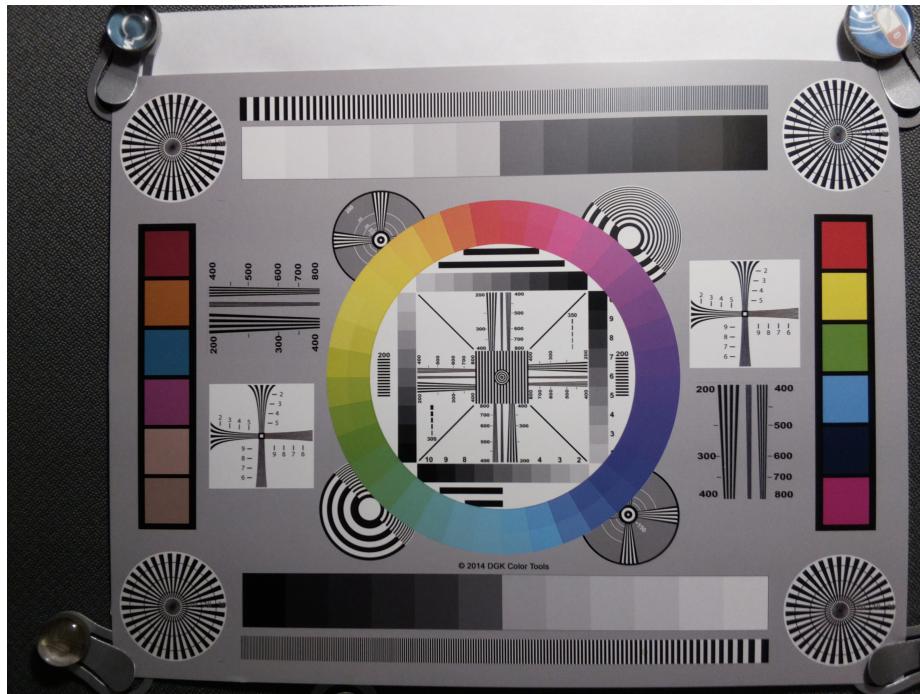


Figure 22: チャート画像

```
# 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_array.shape
```

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、簡易的な画像サイズが 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)
# ラズベリーパイの RAW 画像は 10bit なので、1024 で正規化しておく。
white_level = 1024.0
dms_img = dms_img / white_level
#
# 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))
plt.imshow(gmm_img)
plt.axis('off')
plt.title(u"簡易デモザイクを使ったRAW現像結果")
plt.show()
```

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 ずつになっています。

簡易デモザイクを使ったRAW現像結果

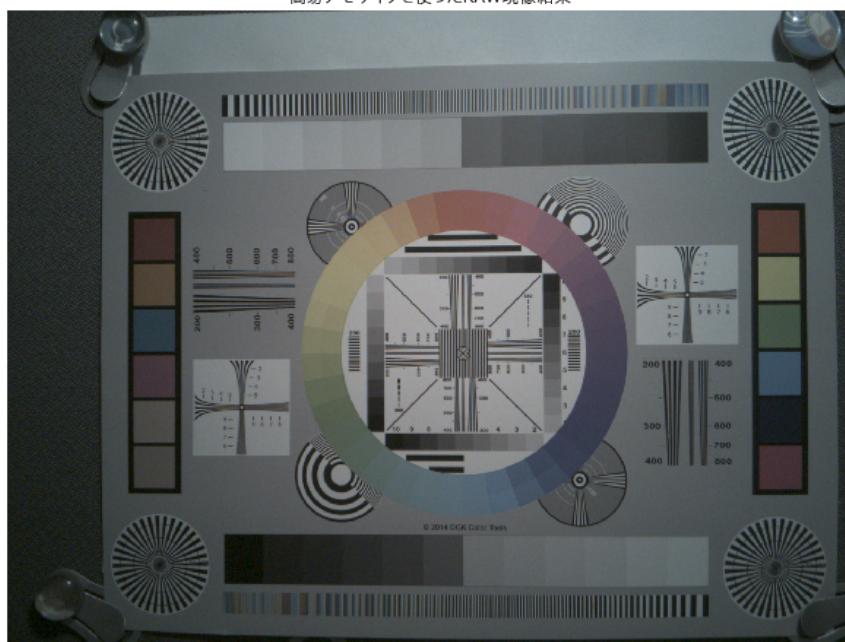


Figure 23: png

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))
plt.imshow(two_img)
plt.axis('off')
plt.title(u"簡易RAW現像結果（左）とJPEG画像（右）")
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
plt.imshow(gmm_img[y1:y1+dy1, x1:x1+dx1])
plt.axis('off')
plt.title("簡易デモザイク結果")

# 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
plt.imshow(jpg_img[y2:y2+dy2, x2:x2+dx2])
plt.axis('off')
plt.title("JPEG画像")
# 実際に表示。
plt.show()

```

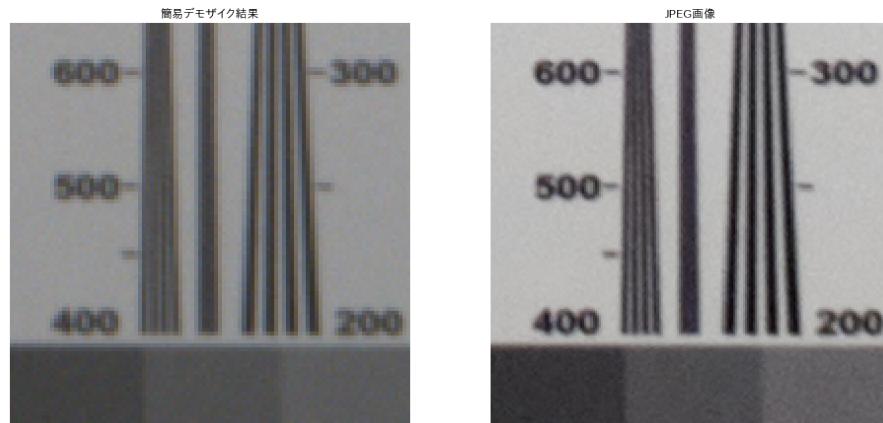


Figure 25: png

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。線形補間というもののものしいですが、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。

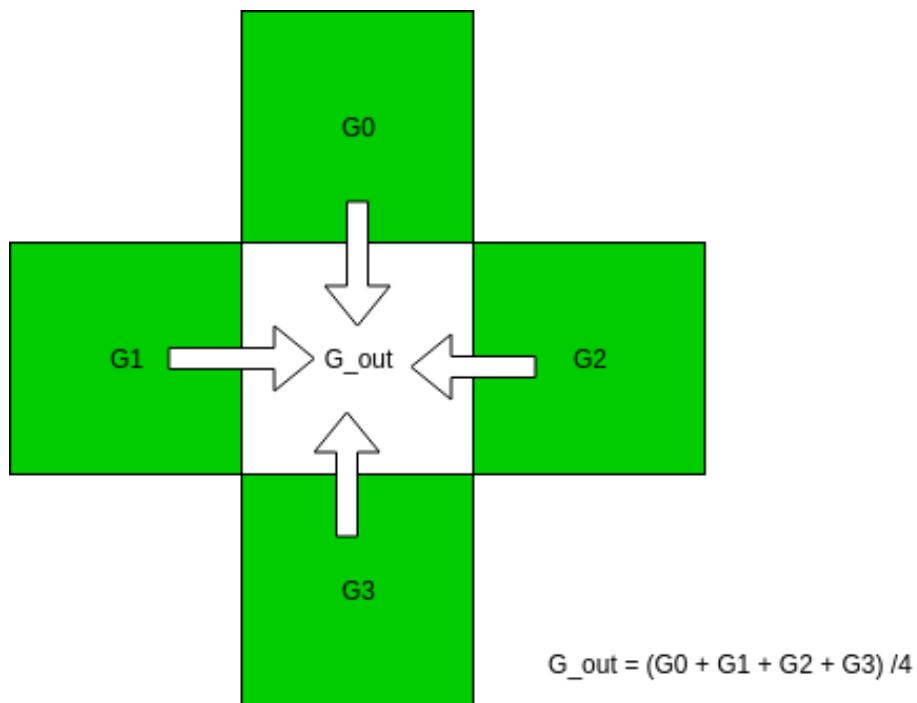


Figure 26: 緑画像の線形補間

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:
```

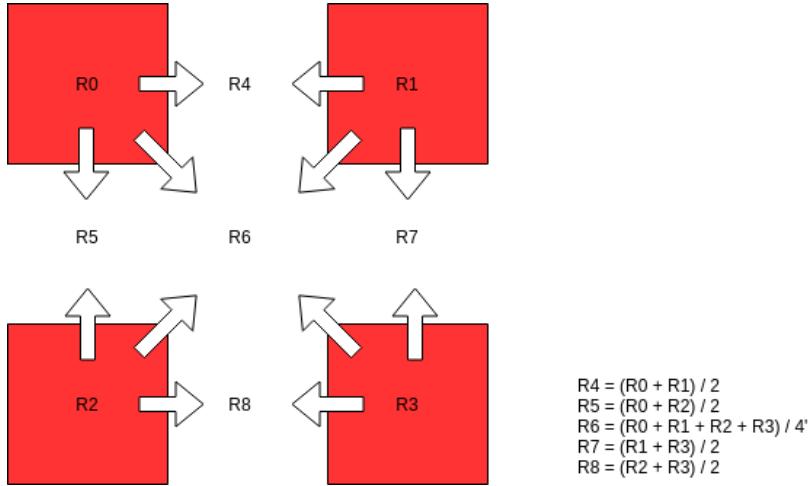


Figure 27: 赤画像の線形補間

```

    return min - x
elif x >= max:
    return 2 * max - x - 2
else:
    return x

dms_img = np.zeros((h, w, 3))
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, x0] + wb_raw[y0, x1] +
                wb_raw[y1, x0] + wb_raw[y1, x1])/4
            dms_img[y, x, 2] = (wb_raw[y0, x0] + wb_raw[y0, x1] +
                wb_raw[y1, x0] + wb_raw[y1, x1])/4
    
```

```

        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 画像と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。

```
# ガンマ補正
gmm_full_img = gamma_correction(dms_img / white_level, 2.2)
```

表示します。

```
# サイズ設定
plt.figure(figsize=(16, 8))
plt.imshow(gmm_full_img)
plt.axis('off')
plt.title(u"線形補間デモザイク画像")
plt.show()
```

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
plt.imshow(gmm_img[y1:y1+dy1, x1:x1+dx1])
plt.axis('off')
plt.title("簡易デモザイク結果")

# 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
```

線形補間デモザイク画像

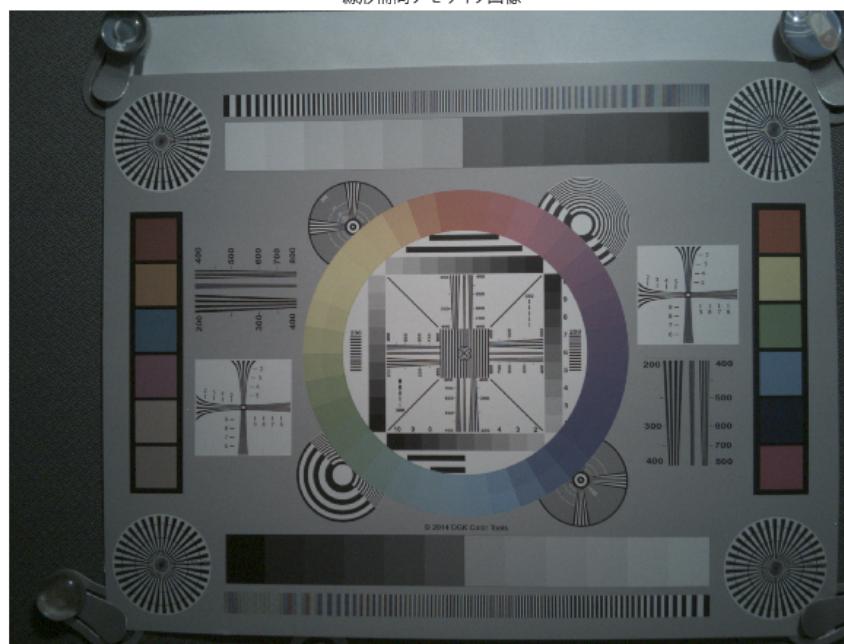


Figure 28: png

```

plt.imshow(gmm_full_img[y2:y2+dy2, x2:x2+dx2])
plt.axis('off')
plt.title(u"線形補間デモザイク画像")

# 実際に表示。
plt.show()

```

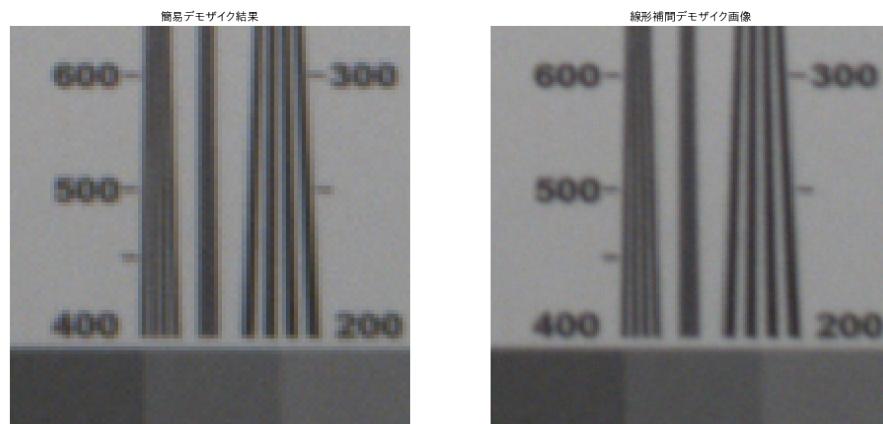


Figure 29: png

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通
# フィルターの適用
dms_img[:, :, 2] = signal.convolve2d(blue, rb_flt,
                                      boundary='symm', mode='same')

```

```

    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。

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。

4.3 欠陥画素補正

この節について

この節では、欠陥画素補正を解説します。

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_ch

準備

まず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、画像の読み込みを行います。今回もラズベリーパイで撮影したチャート画像を使用します。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

# 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, demosaic

# 日本語フォントの設定
!apt -y install fonts-ipafont-gothic
plt.rcParams['font.family'] = 'IPAGothic'
# もし日本語が文字化けしている場合`! rm
    /content/.cache/matplotlib/fontList.json`を実行して、
# Runtime->Restart Runtimeで再実行

# 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_array.shape
```

```

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)
E: Could not open lock file /var/lib/dpkg/lock-frontend - open (13:
    Permission denied)
E: Unable to acquire the dpkg frontend lock
    (/var/lib/dpkg/lock-frontend), are you root?

```

欠陥画素

ライブラリーを使って対象画像を現像して拡大表示すると、こんな部分があります。

```

#
# 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。
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 = demosaic(wb_raw, raw.raw_colors)
# ラズベリーパイのRAW画像は10bitなので、1024で正規化しておく。
white_level = 1024.0
dms_img = dms_img / white_level
#
# raw_processからインポートしたgamma_crrection()関数を使って、ガンマ補正。
gmm_img = gamma_correction(dms_img, 2.2)

# 画像の一部分を拡大表示。
plt.figure(figsize=(16, 8))
plt.imshow(gmm_img[2110:2160, 1150:1210, :])
plt.axis('off')
plt.title(u"欠陥画素")
plt.show()

```

これはいわゆる欠陥画素です。「欠陥」という名前がついていますが、製品の欠陥ではなく、一部の画素が正常な値を出力しない状態です。多くの場合このように、常に明るく見えますが（ホットピクセル）、常に暗い（デッドピクセルまたはコールドピクセル）こともあります。ともあれば、本来の信号とずれた値を示すという捕まえにくいケースもあります。

欠陷画素



Figure 30: png

数ミリ角のサイズに数百万から数千万の画素を作り込む現代の画像センサーでは、一部にこのような欠陥画素が含まれていることはごく普通のことです。

なぜこのような欠陥ができるかというのには、さまざまな原因が考えられます。たとえば製造過程でパーティクルが入り 1 特定の画素が反応しなくなったとか逆にショートして常に電流が流れるようになった、というのがまさしく思い描きます。まだ半導体中の結晶欠陥などのせいで基板側に電流が漏れている（または基板から漏れて入ってくる）のかもしれません。さらには宇宙線などが当たって製造後に欠陥が形成されるケースもあると聞きます。

このように、画像センサーに欠陥画素はつきもので、画像処理である程度対応していく必用があります。市販のスマートフォンのカメラはもちろん、一眼レフカメラなどでも欠陥画素補正処理は内部的に行われているはずです。

それでは、実際に簡単な補正処理を行ってみます。

欠陥画素補正には大きく分けて 2 つのステップがあります 2。

- 欠陥画素検出
- 欠陥画素修正

今回は、欠陥画素検出としては、周辺の画素の最大値よりある程度以上大きいか最小値よりある程度以上小さければ欠陥とみなす、という方針で行います。また修正としては、欠陥画素の上下左右 4 画素の平均をとる事にします。

それでは処理してみましょう。ブラックレベル補正後のデータ `blc_raw` から処理をはじめます。

各色毎に欠陥画素の検出と修正を行います。なお、ここでは Gr (Red と同列の Green) と Gb (Blue と同列の Green) を別の色として処理しています。まず、左上の色 (Blue) からです。処理しやすいように Blue のみを抜き出します。

```
dpc_raw = blc_raw.copy()
single_channel = dpc_raw[:, :, 2]
```

周辺の 5x5 画素の最大値と最小値を求めます。

最大値最小値を求めるのに `scipy` の `ndfilter` モジュールから、`maximum_filter` と `minimum_filter` を使います。

```
import scipy

# 5x5 の行列を作る、全成分を 1 にする。
footprint = np.ones((5, 5))
# 中心の値のみを 0 にする。
footprint[2, 2] = 0
#
# 入力画像の各画素周辺 5 x 5 の最大値をもとめて、画像と同じサイズの numpy
# array として保存。
local_max = scipy.ndimage.filters.maximum_filter(single_channel,
    footprint=footprint, mode='mirror')
```

```

# 入力画像の各画素周辺 5 x 5 の最小値をもとめて、画像と同じサイズのnumpy arrayとして保存。
local_min = scipy.ndimage.filters.minimum_filter(single_channel,
    footprint=footprint, mode='mirror')

```

ここで footprint によって、どの画素を最大値・最小値の計算の対象にするか決めることができます。

ここでは、5x5 の行列を作り、中心の成分のみをゼロ、他は 1 として footprint を作成しました。

$$footprint = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

これを使うことで、欠陥画素かどうか判定する対象を最大値最小値の計算から取り除くことができます。

この最大値と最小値を使って、欠陥画素判定を行います。最大値や最小値との差が threshold 値より大きい場合欠陥画素とみなす事にします。

```

threshold = 16
mask = (single_channel < local_min - threshold) + (single_channel >
    local_max + threshold)

```

これで欠陥画素の位置が mask に True として記録されました。(欠陥画素以外は False)

それでは修正しましょう。

まず、欠陥画素の上下左右の画素の平均値を計算しておきます。

```

# 上下左右の平均値を取るフィルター。
flt = np.array([[0, 1, 0], [1, 0, 1], [0, 1, 0]]) / 4
# scipyの機能で上下左右画素の平均値をとり、結果をnumpy arrayとして保存。
average = scipy.signal.convolve2d(single_channel, flt, mode='same')

```

次に欠陥画素をこの平均値で置き換えます。

欠陥画素は mask のうち True になっている部分です。

```
single_channel[mask] = average[mask]
```

これで Blue 面の欠陥画素補正ができました。

他の色の欠陥も補正するためにループ化します。

```

# ブラックレベル補正後の画像をコピー。
dpc_raw = blc_raw.copy()
# footprintとして5x5のマスクを作成
# [[1 1 1 1 1]
#  [1 1 1 1 1]
#  [1 1 0 1 1]
#  [1 1 1 1 1]
#  [1 1 1 1 1]]
footprint = np.ones((5, 5))
footprint[2, 2] = 0

# 各カラーごとの処理。左上(0, 0)、左下(1, 0), 右上(0, 1), 右下(1, 1)
for (yo, xo) in ((0, 0), (1, 0), (0, 1), (1, 1)):
    # 1カラーチャンネルを取り出す。
    single_channel = dpc_raw[yo::2, xo::2]
    # 上下左右の平均をとるフィルター
    flt = np.array([[0, 1, 0], [1, 0, 1], [0, 1, 0]]) / 4
    # 平均値をとった画像の作成
    average = scipy.signal.convolve2d(single_channel, flt,
        mode='same')
    #
    # 周辺画像の最大値を求める。footprintにより、対象となる画素は含めない。
    local_max = scipy.ndimage.filters.maximum_filter(single_channel,
        footprint=footprint, mode='mirror')
    #
    # 周辺画像の最小値を求める。footprintにより、対象となる画素は含めない。
    local_min = scipy.ndimage.filters.minimum_filter(single_channel,
        footprint=footprint, mode='mirror')
    #
    # 中心画素が最大値よりthreshold分以上大きい、または最小値よりthreshold分以上小さければ欠陥
    threshold = 16
    # 欠陥の位置をTrueとして保存。
    mask = (single_channel < local_min - threshold) +
        (single_channel > local_max + threshold)
    # 欠陥画素を平均値で置換。
    single_channel[mask] = average[mask]
    # single_channelはdpc_rawへの参照なので書き戻す必要がない

```

残りのホワイトバランス、デモザイク、ガンマ処理を行い、出力結果を確認します。

```

wb_raw = white_balance(dpc_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 = demosaic(wb_raw, raw.raw_colors)
# ラズベリーパイのRAW画像は10bitなので、1024で正規化しておく。

```

```

white_level = 1024.0
dms_img = dms_img / white_level
#
    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)

# 画像の一部分を拡大表示。
plt.figure(figsize=(16, 8))
plt.imshow(gmm_img)
plt.axis('off')
plt.title(u"欠陥画素補正後")
plt.show()

```

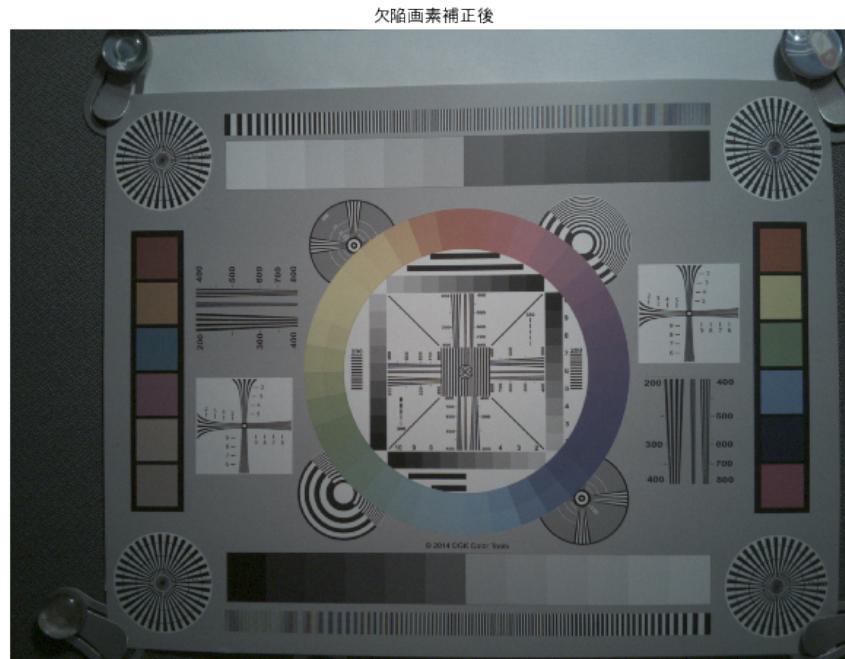


Figure 31: png

欠陥画素が修正されたか、先ほどと同じ部分を拡大して確認しましょう。

```

# 画像の一部分を拡大表示。
plt.figure(figsize=(16, 8))
plt.imshow(gmm_img[2110:2160, 1150:1210, :])
plt.axis('off')
plt.title(u"補正された欠陥画像")
plt.show()

```

補正された欠陥画像

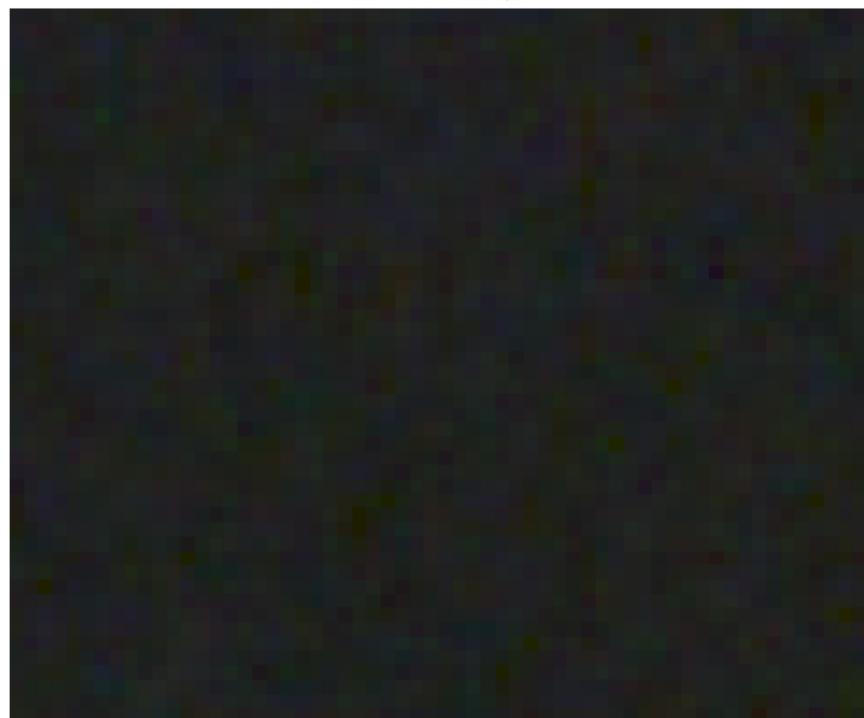


Figure 32: png

どうやら修正されたようです。

モジュールへの追加

この処理も関数としてモジュールへ追加しておきましょう。

```
def defect_correction(raw_array, threshold):
    """
    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。
    threshold: int
        欠陥画素判定の閾値。
        10bitRAW入力に対して典型的には16程度。

    Returns
    -----
    dpc_raw: numpy array
        出力RAW画像。
    """
    dpc_raw = raw_array.copy()
    # footprintとして5x5のマスクを作成
    # [[1 1 1 1 1]
    #  [1 1 1 1 1]
    #  [1 1 0 1 1]
    #  [1 1 1 1 1]
    #  [1 1 1 1 1]]
    footprint = np.ones((5, 5))
    footprint[2, 2] = 0
    # 各カラーごとの処理。左上(0, 0)、左下(1, 0)、右上(0, 1)、
    # 右下(1, 1)
    for (yo, xo) in ((0, 0), (1, 0), (0, 1), (1, 1)):
        single_channel = dpc_raw[yo::2, xo::2]
        # 上下左右の平均をとるフィルター
        flt = np.array([[0, 1, 0], [1, 0, 1], [0, 1, 0]]) / 4
        # 上下左右の平均値をとった画像の作成
        average = scipy.signal.convolve2d(single_channel, flt,
                                           mode='same')
        #
        # 周辺画像の最大値を求める。footprintにより、対象となる画素は含めない。
        local_max =
            scipy.ndimage.filters.maximum_filter(single_channel,
```

```

        footprint=footprint, mode='mirror')
#
# 周辺画像の最小値を求める。footprintにより、対象となる画素は含めない。
local_min =
    scipy.ndimage.filters.minimum_filter(single_channel,
        footprint=footprint, mode='mirror')
#
# 中心画素が最大値よりthreshold分以上大きい、または最小値よりthreshold分以上小さければ
# 欠陥の位置をTrueとして保存。
mask = (single_channel < local_min - threshold) +
    (single_channel > local_max + threshold)
#
# 欠陥画素を平均値で置換。
single_channel[mask] = average[mask]
# single_channelはdpc_rawへの参照なので書き戻す必用がない
return dpc_raw

```

このdefect_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 defect_correction
としてインポートしてください。
```

まとめ

この節では欠陥画素補正を行いました。次はカラーマトリクス補正を行います。

4.4 カラーマトリクス補正

この節について

この節では、カラーマトリクス補正を解説します。

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_ch

準備

まず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、画像の読み込みを行います。今回もラズベリーパイで撮影したチャート画像を使用します。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とインポート
!if [ ! -f raw_process.py ]; then wget raw_process.py; fi
from raw_process import simple_demosaic, white_balance,
    black_level_correction, gamma_correction, demosaic,
    defect_correction

# 日本語フォントの設定
!apt -y install fonts-ipafont-gothic
plt.rcParams['font.family'] = 'IPAPGothic'
# もし日本語が文字化けしている場合`! rm
    /content/.cache/matplotlib/fontList.json`を実行して、
# Runtime->Restart Runtimeで再実行

# 画像をダウンロードします。
!if [ ! -f chart.jpg ]; then wget 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_array.shape

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)
E: Could not open lock file /var/lib/dpkg/lock-frontend - open (13:
    Permission denied)
E: Unable to acquire the dpkg frontend lock
    (/var/lib/dpkg/lock-frontend), are you root?

```

カラーマトリクスとは

前節までに RAW 現像した画像と、JPEG 画像を並べて比較してみましょう。こちらが JPEG 画像を JPEG としてそのまま表示したものです。

```

# JPEG画像をnumpyのarrayとして取得
#
[4.2節参照](https://colab.research.google.com/github/moizumi99/camera_raw_processing/blob/ma
from matplotlib import image
import matplotlib
jpg_img = image.imread("chart.jpg")
jpg_img = jpg_img / jpg_img.max()

# JPEG画像表示
plt.figure(figsize=(16, 8))
imshow(jpg_img)
plt.axis('off')
plt.title(u"JPEG画像")
plt.show()

```

それに対してこちらが、前回までに現像したものです。

```

# ブラックレベル補正。
blc_raw = black_level_correction(raw_array,
    raw.black_level_per_channel, raw.raw_pattern)
# 前節で作成したdefect_correction関数を使って、欠陥画素補正。
dpc_raw = defect_correction(blc_raw, 16)
# 残りの現像処理
wb_raw = white_balance(dpc_raw, raw.camera_whitebalance,
    raw.raw_colors)
dms_img = demosaic(wb_raw, raw.raw_colors)
# ガンマ処理の前にwhite_levelで正規化
white_level = 1024
dms_img = dms_img / white_level
gmm_img = gamma_correction(dms_img, 2.2)

```

JPEG画像

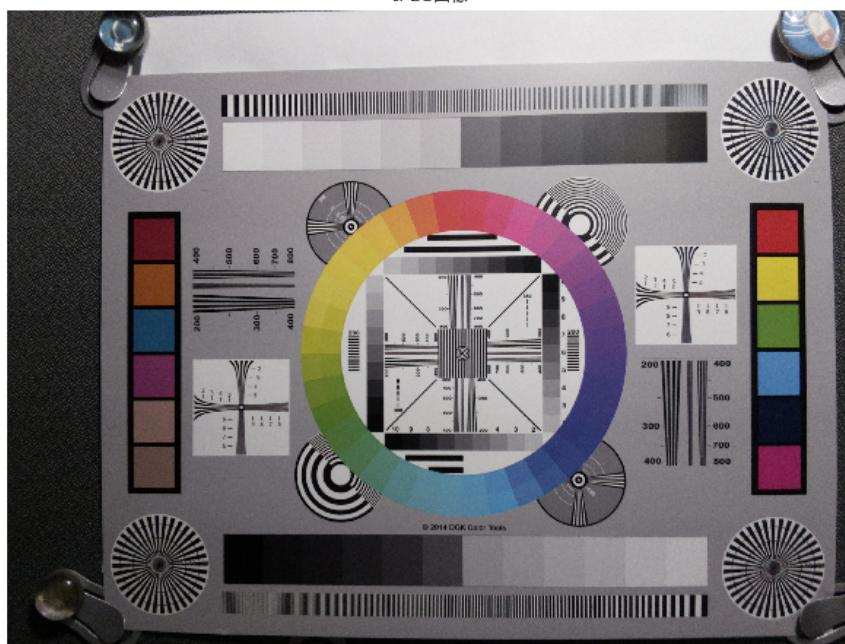


Figure 33: png

```
# 画像表示。
plt.figure(figsize=(16, 8))
imshow(gmm_img)
plt.axis('off')
plt.title(u"RAW現像した画像")
plt.show()
```

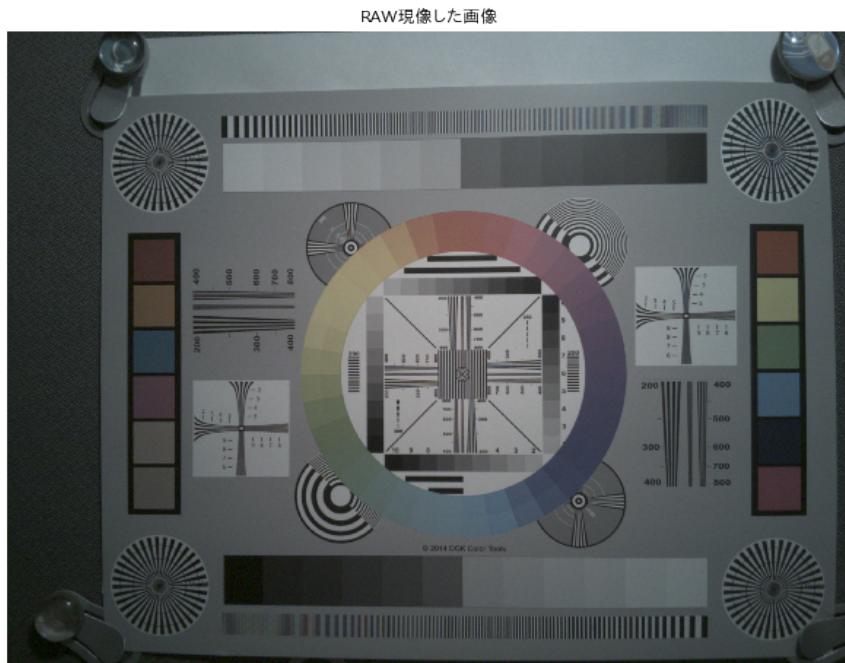


Figure 34: png

2つの画像を見比べると、色の鮮やかさが違うことがわかると思います。また色合いも完全には一致していません。実際の見た目も最初の JPEG の画像に近くなっています。

どうしてこうなってしまうかというと、最大の原因はカメラのセンサーの色ごとの感度が人間の目とは完全には一致しないことです。例えば人間の目はある光の周波数の範囲を赤、青、緑、と感じますが、センサーが緑を検知する範囲は人間が緑と感じる領域とは微妙に異なっています。同じように青や赤の範囲も違います。これは、センサーが光をなるべく沢山取り込むため、だとか、製造上の制限、などの理由があるようです。さらに、人間の目には、ある色を抑制するような領域まであります。これはセンサーで言えばマイナスの感度があるようなのですが、そんなセンサーは作れません。

こういったセンサー感度と人間の目の間隔となるべく小さくなるように、3 色を混ぜて、より人間の感覚に近い色を作り出す必要があります。この処理を通常は行列を使って行い、これをカラーマトリクス補正と呼びます。

カラーマトリクス補正というのは処理的には 3×3 の行列に、3 色の値を成分としたベクトルをかけるという処理になります。

$$\begin{pmatrix} R_{out} \\ G_{out} \\ B_{out} \end{pmatrix} = \begin{pmatrix} c_0 & c_1 & c_2 \\ c_3 & c_4 & c_5 \\ c_6 & c_7 & c_8 \end{pmatrix} \begin{pmatrix} R_{in} \\ G_{in} \\ B_{in} \end{pmatrix}$$

このような処理で、色の深み・鮮やかさ (Saturation と呼ばれます)、色合い (Hue と呼ばれます) をある程度修正することができます。

カラーマトリクス補正

それでは実際にカラーマトリクス補正をかけてみましょう。

まず、撮影された RAW 画像に保存されているカラーマトリクスの値を読んでみましょう。RAWPY にもカラーマトリクスを読み取る機能はあるのですが、実際に実行するところなってしまいます。

```
print(raw.color_matrix)
```

```
[[0. 0. 0. 0.]  
 [0. 0. 0. 0.]  
 [0. 0. 0. 0.]]
```

これでは仕方がないので、exiftool を使います。exiftool はコマンドラインから各種画像ファイルの情報を取り出すツールです。

まず colab の環境に exiftool をインストールします。

```
! apt install exiftool
```

```
E: Could not open lock file /var/lib/dpkg/lock-frontend - open (13:  
      Permission denied)  
E: Unable to acquire the dpkg frontend lock  
      (/var/lib/dpkg/lock-frontend), are you root?
```

Exiftool をつかって、カラーマトリクスの内容を見てみましょう。

Raspberry Pi の RAW 画像の場合、カラーマトリクスはメーカーノート情報（カメラメーカー独自のデータ）に含まれています³。メーカーノートは-EXIF:MakerNoteUnknownText -bオプションで読むことができます。

```
! exiftool -EXIF:MakerNoteUnknownText -b chart.jpg -b
```

³Bilateral Filtering for Gray and Color Images, C. Tomasi and R. Manduchi, Proceedings of the 1998 IEEE International Conference on Computer Vision, pp. 839–846, 1998

```

ev=-1 mlux=-1 exp=62998 ag=556 focus=255 gain_r=1.128 gain_b=2.546
greenness=3
ccm=6022,-2314,394,-936,4728,310,300,-4324,8126,0,0,0 md=0
tg=262 262 oth=0 0 b=0 f=262 262 fi=0 ISP Build Date: Oct 8
2018, 17:46:45 VC_BUILD_ID_VERSION:
656741eb5ba785fc4f1014a3a3b1c0e9c2cc8487 (clean)
VC_BUILD_ID_USER: dc4 VC_BUILD_ID_BRANCH: master

```

繋がっていて読みにくいですが、よく見てみるとこのような部分があるのがわかります。

```
ccm=6022,-2314,394,-936,4728,310,300,-4324,8126,0,0,0
```

これは、カラーマトリクス (CCM: Color Correction Matrix) が次のような値であることを示しています。

$$\begin{pmatrix} 6022 & -2314 & 394 \\ -936 & 4728 & 310 \\ 300 & -4324 & 8126 \end{pmatrix}$$

このままでは入力と掛け合せた時に非常に大きくなってしまうので、何かの値で正規化しなくてはなりません。Exif のメーカーノートには特に記載がありませんが、各行の和が 4096 に近いので、4096 で正規化しておくことにしましょう。

それではこの値を使ってカラーマトリクス補正を行ってみましょう。カラーマトリクス補正是線形な色空間で行う必用があるので、ガンマ補正の前に行います。

```

# カラーマトリクス
# [[6022,-2314,394]
# [-936,4728,310]
# [300,-4324,8126]] / 4096
color_matrix =
    np.array([[6022,-2314,394], [-936,4728,310], [300,-4324,8126]]) /
4096

# 出力先を作成しておく
ccm_img = np.zeros_like(dms_img)
# 実際に 1 画素毎に処理
# 入力はデモザイク処理後の画像。
for y in range(0, h):
    for x in range(0, w):
        pixel = dms_img[y, x, :]
        # numpy の dot は行列同士の掛け算を計算する関数
        pixel = np.dot(color_matrix, pixel)
        ccm_img[y, x, :] = pixel

```

残っているガンマ補正を行います。

```
ccm_gmm_img = gamma_correction(ccm_img, 2.2)
```

最終的な画像を表示して、カラーマトリクス補正なしのものと比較してみましょう。

```
# 表示領域設定
plt.figure(figsize=(12, 16))
# 2 x 1 (縦 2, 横 1)のうち 1つめの表示
plt.subplot(2, 1, 1)
# CCM補正なしの画像を表示
plt.imshow(gmm_img)
plt.axis('off')
plt.title(u"CCM補正なし")
# 2 x 1 (縦 2, 横 1)のうち 2つめの表示
plt.subplot(2, 1, 2)
# CCM補正ありの画像を表示
plt.imshow(ccm_gmm_img)
plt.axis('off')
plt.title(u"CCM補正あり")
# 画像を描画
plt.show()
```

CCM ありのほうが色が鮮やかになっているのがわかると思います。

モジュールへの追加

この処理も高速化して、関数としてモジュールへ追加しておきましょう。

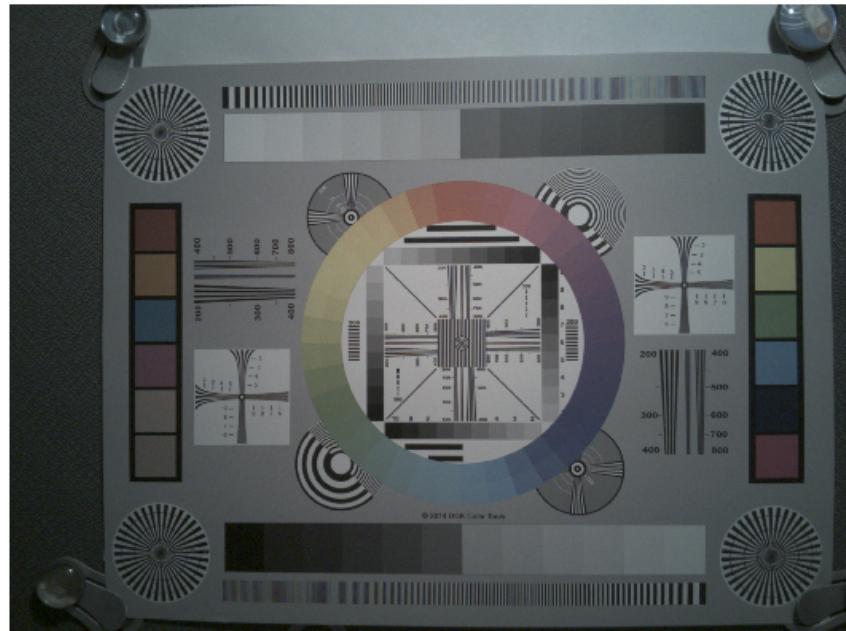
```
def color_correction_matrix(rgb_array, color_matrix):
    """
    カラーマトリクス補正を行う。

    Parameters
    -----
    rgb_array: numpy array
        入力RGB画像
    color_matrix: 2D (3x3) array like
        3x3 Color Correction Matrix
        Need to be normalized to 1.0

    Returns
    -----
    ccm_img: numpy array
        出力RGB画像
    """

    # 出力先を作成
    ccm_img = np.zeros_like(rgb_array)
    # CCMが3x3フォーマットでない場合、3x3に変換
```

CCM補正なし



CCM補正あり

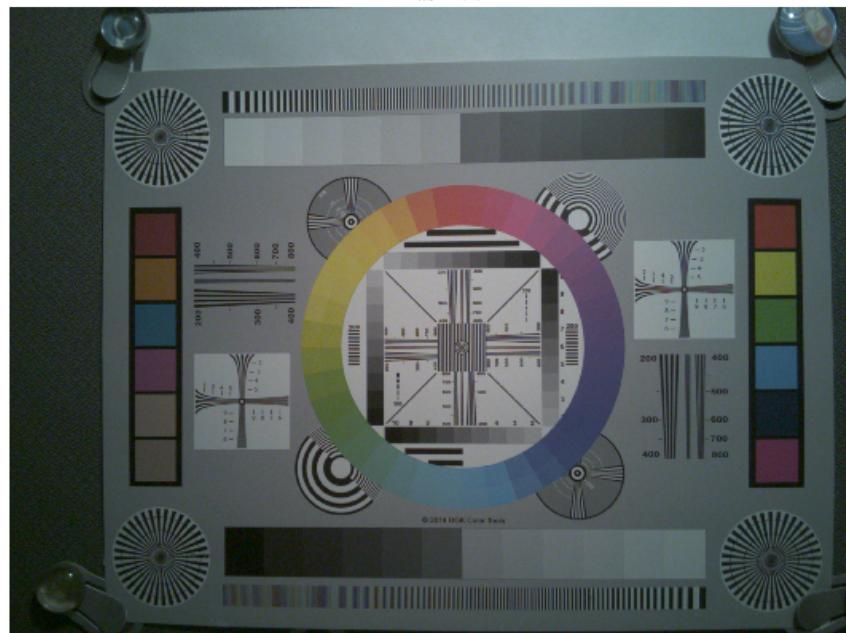


Figure 35: png
74

```

ccm = np.array(color_matrix).reshape((3, 3))
# 各色毎に処理。この方が各画素ごとに処理するよりも高速なようだ。
for color in (0, 1, 2):
    # 行列と入力画像の各色を掛け合わせる。
    ccm_img[:, :, color] = ccm[color, 0] * rgb_array[:, :, 0] + \
        ccm[color, 1] * rgb_array[:, :, 1] + \
        ccm[color, 2] * rgb_array[:, :, 2]
return ccm_img

```

このcolor_correction_matrix()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 color_correction_matrix
```

としてインポートしてください。

まとめ

この節ではカラーマトリクス補正を行いました。次はレンズシェーディング補正を行います。

4.5 シェーディング補正

この節について

この節では、シェーディング補正について解説します。

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_ch

準備

まず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、画像の読み込みを行います。今回もラズベリーパイで撮影したチャート画像を使用します。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とインポート
!if [ ! -f raw_process.py ]; then wget raw_process.py; fi
from raw_process import simple_demosaic, white_balance,
    black_level_correction, gamma_correction
from raw_process import demosaic, defect_correction,
    color_correction_matrix, lens_shading_correction

# 日本語フォントの設定
!apt -y install fonts-ipafont-gothic
plt.rcParams['font.family'] = 'IPAPGothic'
# もし日本語が文字化けしている場合`! rm
    /content/.cache/matplotlib/fontList.json`を実行して、
# Runtime->Restart Runtimeで再実行

# 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_array.shape

```

```

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)
E: Could not open lock file /var/lib/dpkg/lock-frontend - open (13:
    Permission denied)
E: Unable to acquire the dpkg frontend lock
    (/var/lib/dpkg/lock-frontend), are you root?

```

レンズシェーディング（周辺減光）とは

デジタルカメラに限らず、レンズを通して結像した画像は中央部より周辺部のほうが暗くなっています。

このような現象をレンズシェーディングや単にシェーディング、また日本語では周辺減光などといいます。英語では Lens Shading や Vignetting と呼ばれます。

このような事が起きてしまう第一の原因是、レンズを通してセンサーにあたる光の量が、センサーの中央部と、センサーの周辺部とで異なる事です。良く説明に出されるのが二枚のレンズを持つ単純な系の場合です。

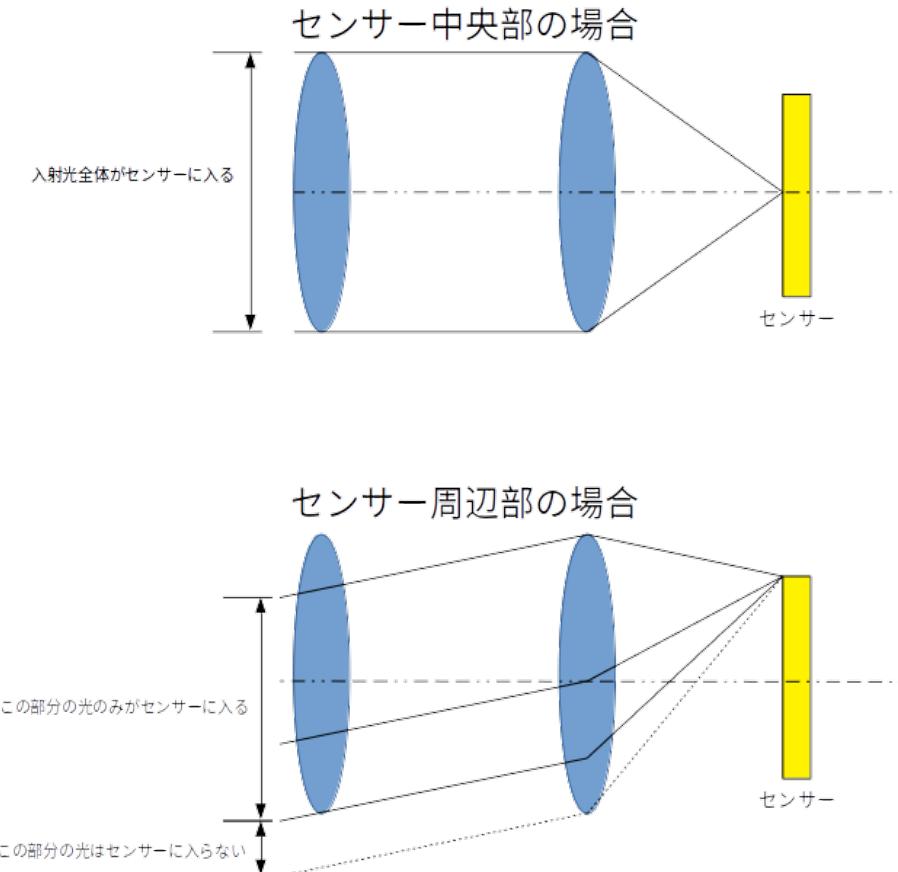


Figure 36: レンズシェーディングが起こるしくみ

この図の例では、センサー中央部分にあたる光はレンズの開口部全体を通ってくるのに対し、センサー周辺部にあたる光はレンズの一部分しか通過できません。結果的に中心部が明るく、周辺部が暗くなります。実際のカメラの光学系はこれより遥かに複雑ですが、同様の理由により画像の周辺部分が暗くなります。

この要因の他に、センサー周辺部ではレンズやカバーの縁に遮られる光の量も増えます。また、光の入射角度によってレンズ等のコーティングと干渉する可能性もあります。またデジタルカメラ独特の状況として、画像センサー上のフォトダイオードに到達する光の量が、光の入射角に依存するケースがあります。

シェーディングを起こす要因はこのように多岐にわたり、一概に、これが原因だとは言うことはできません。したがってモデル計算で減光量を求めるよりも、実際のレンズで測定した結果を元に画像補正する必要があります。

レンズシェーディングの確認

では実際にシェーディングの影響を見てみましょう。

本来ならば明るさを均一にしたグレイチャート（その名の通り灰色の大きなシート）などを撮影してテストするのですが普通の家庭にそのような物はないので、今回はラズベリーパイのレンズの上に白いコピー用紙を載せ、そこに後ろから光を当てることでなるべく一様な明るさの画像を撮影しました。ファイル名は flat.jpgです。このファイルはgithub にアップロードしております。

まずはダウンロードしてみます。

```
! wget flat.jpg

--2019-02-03 09:03:40-- flat.jpg
Resolving github.com (github.com)... 192.30.255.112, 192.30.255.113
Connecting to github.com (github.com)|192.30.255.112|:443...
    connected.
HTTP request sent, awaiting response... 302 Found
Location:
    https://raw.githubusercontent.com/moizumi99/camera_raw_processing/master/flat.jpg
    [following]
--2019-02-03 09:03:40--
    https://raw.githubusercontent.com/moizumi99/camera_raw_processing/master/flat.jpg
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: 14561271 (14M) [application/octet-stream]
Saving to: 'flat.jpg.5'

flat.jpg.5          100%[=====] 13.89M 40.4MB/s
    in 0.3s

2019-02-03 09:03:41 (40.4 MB/s) - 'flat.jpg.5' saved
[14561271/14561271]
```

このファイルから RAW 画像を取り出し現像してみましょう。まだレンズシェーディング補正は行いません。

```
# RAW画像の読み込み。
raw_file = "flat.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));

#RAW現像処理
blc_raw = black_level_correction(raw_array,
    raw.black_level_per_channel, raw.raw_pattern)
dpc_raw = defect_correction(blc_raw, 16)
wb_raw = white_balance(dpc_raw, raw.camera_whitebalance,
    raw.raw_colors)
dms_img = demosaic(wb_raw, raw.raw_colors)
color_matrix =
    np.array([6022,-2314,394,-936,4728,310,300,-4324,8126])/4096
ccm_img = color_correction_matrix(dms_img, color_matrix)
white_level = 1024.0
no_shading_img = gamma_correction(ccm_img / white_level, 2.2)

# 画像表示
plt.figure(figsize=(16, 8))
plt.imshow(no_shading_img)
plt.axis('off')
plt.title(u"レンズシェーディング補正前")
plt.show()

```

確かに周辺光量が落ちているのが確認できます。その他に画像中央部と周辺部とで色味が違うのもわかると思います。

画像の中で明るさがどう変わっているか見てみましょう。画面の高さ方向中央付近、上下32画素幅で左から右まで帯状の画像をとりだし、明るさがどう変わるグラフにしてみます。

```

# 基準となる画像中央部の位置
center_y, center_x = h // 2, w // 2
# 明るさの配列を保存する配列
shading_profile = [[], [], []]

# 画面中央の上下+/-16ライン幅で測定
y = center_y - 16
# 横方向32画素毎にサンプリング
for x in range(0, w - 32, 32):
    xx = x + 16
    # (x, y)から(x+32, y+32)を対角線とする正方形内部の画素の平均
    # 色ごとに保存する。
    shading_profile[0].append(no_shading_img[y:y+32, x:x+32,
        0].mean())

```

レンズシェーディング補正前

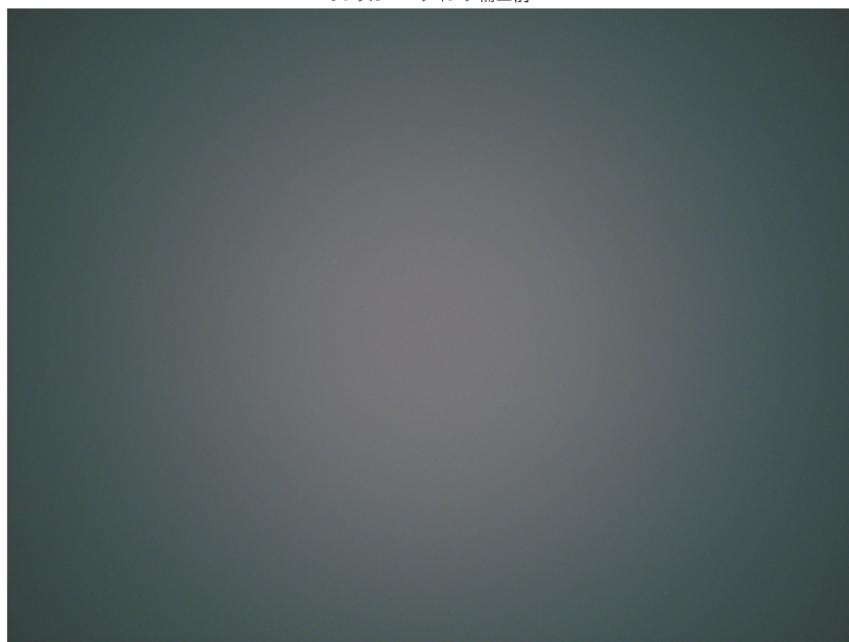


Figure 37: png

```

shading_profile[1].append(no_shading_img[y:y+32, x:x+32,
    1].mean())
shading_profile[2].append(no_shading_img[y:y+32, x:x+32,
    2].mean())
# 画像中央部の値で正規化する
shading_profile = [np.array(a) / max(a) for a in shading_profile]

plt.axis(ymin=0, ymax=1.1)
plt.plot(shading_profile[0], color='red')
plt.plot(shading_profile[1], color='green')
plt.plot(shading_profile[2], color='blue')
plt.title(u"明るさの横方向の変化")
plt.show()

```

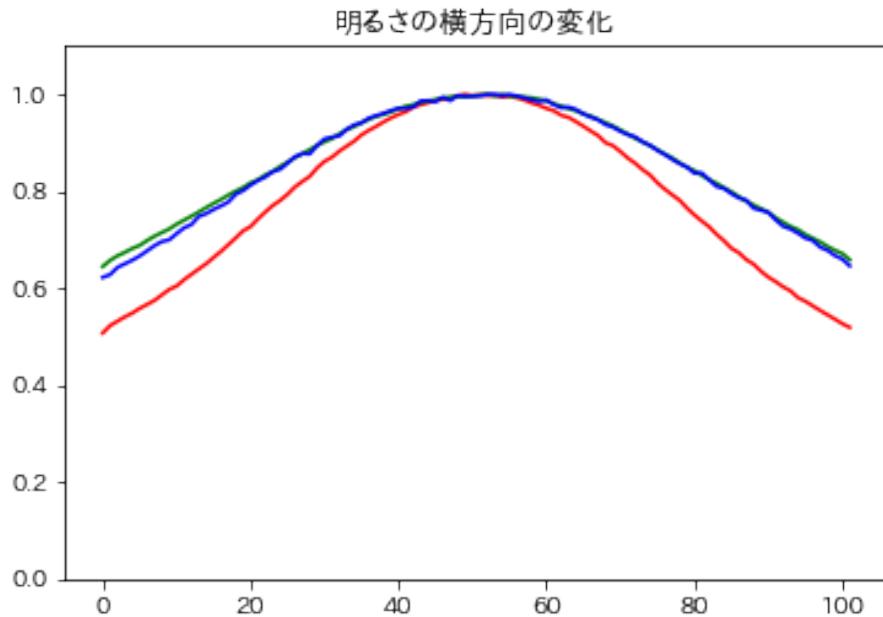


Figure 38: png

画像の左右端では中心部分に比べて 50% 程度の明るさに落ちていることがわかります。これはガンマ補正後の値なので、RAW 画像では明るさの違いはさらに大きいことが予想できます。

また、赤画素と、緑・青画素とではシェーディングの様子がだいぶ違います。画像の色が中央と周辺部でだいぶ違うのはこのせいでしょう。

レンズシェーディングのモデル化

レンズシェーディングを補正するために、まずどの程度の減光があるのか測定してみましょう。

画像を見てわかるとおり、光の量は中心から離れるに従って減っています。中央からの距離によってパラメータ化できそうです。また、対称性を考えると偶関数で近似できるはずです。

では各画素の明るさを、中央からの距離に応じてグラフにしてみましょう。

一画素毎に計算するのは計算量が大きくまたノイズによる誤差も入ってくるので、32 x 32 のブロックごとに測定します。また、今度は実際に補正する量の見積もりに使いたいので Bayer での値を使います。ゼロレベルを正しく取る必用があるので、測定する対象はブラックレベル補正後の画像です。

```
# 測定値を入れる配列を色の数分だけ準備する。
vals = [[], [], [], []]
# 中心からの距離を保存
radials = []

# 32x32のブロック毎に処理を行います。
# (x, y)から(x+32, y+32)を対角線とする正方形毎に処理。
for y in range(0, h, 32):
    for x in range(0, w - 32, 32):
        # (xx, yy)は正方形の中心。
        xx = x + 16
        yy = y + 16
        # 正方形の中心と画像の中心の間の距離を求め記録。
        r2 = (yy - center_y) * (yy - center_y) + (xx - center_x) *
              (xx - center_x)
        radials.append(r2)
        # 色ごとに正方形の中の画素の平均値を求める。
        vals[0].append(blc_raw[y:y+32:2, x:x+32:2].mean())
        vals[1].append(blc_raw[y:y+32:2, x+1:x+32:2].mean())
        vals[2].append(blc_raw[y+1:y+32:2, x:x+32:2].mean())
        vals[3].append(blc_raw[y+1:y+32:2, x+1:x+32:2].mean())
```

これで`vals[]`には色ごとの画素の明るさ、`radials`には中央からの距離の2乗が入っているはずです。

最大値でノーマライズしてグラフにして確認してみます。

```
# 扱いやすいようにnumpyの配列に変換。
rs = np.array(radials)
vs = np.array(vals)
# 最大値で正規化
norm = vs.max(axis=1)
vs[0, :] /= vs[0, :].max()
```

```

vs[1, :] /= vs[1, :].max()
vs[2, :] /= vs[2, :].max()
vs[3, :] /= vs[3, :].max()
# Pyplotの散乱図機能でグラフを描画。
plt.figure(figsize=(8, 8))
plt.scatter(rs, vs[0,:], color='blue')
plt.scatter(rs, vs[1,:], color='green')
plt.scatter(rs, vs[2,:], color='green')
plt.scatter(rs, vs[3,:], color='red')
plt.ylim(0, 1.05)
plt.title(u"RAW画像の明るさと中心からの距離")
plt.show()

```

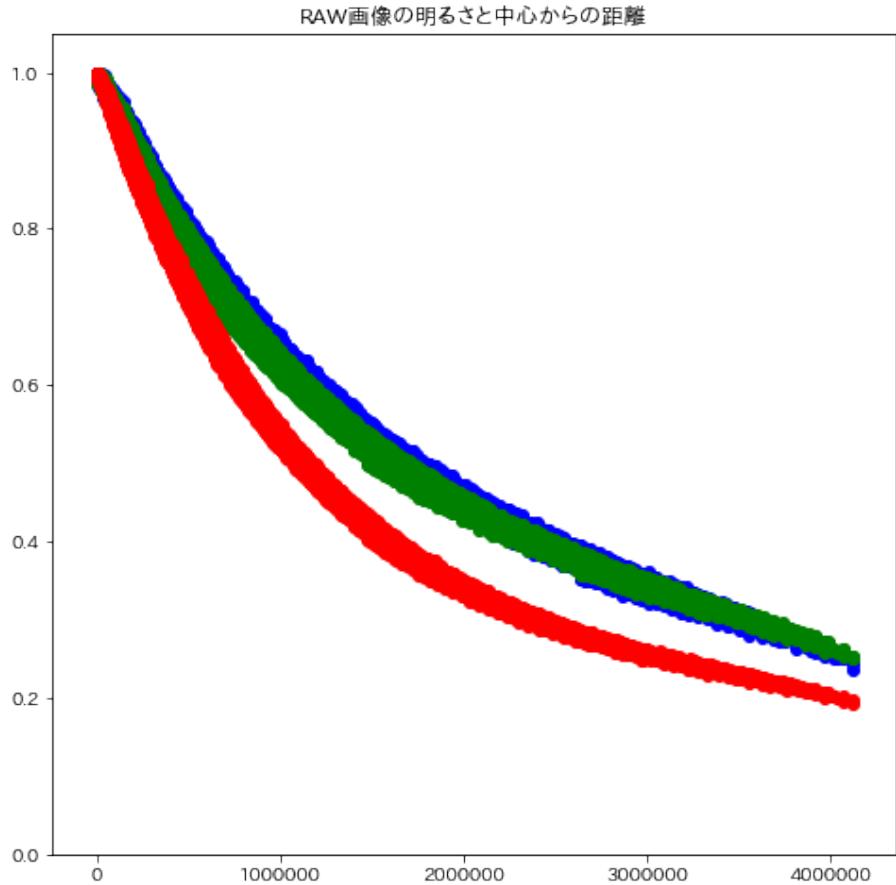


Figure 39: png

きれいに中心からの距離に応じて明るさが減少しています。また、この段階では周辺部

では中心部に比べて3分の1程度に暗くなっていることがわかります。

これを補正するには、明るさの減少率の逆数をかけてやればよい事になります。逆数のグラフを書いてみましょう。

```
# 明るさの逆数を求める。
gs = 1 / vs
# 明るさの逆数のグラフ。
plt.figure(figsize=(8, 8))
plt.scatter(rs, gs[0,:], color='blue')
plt.scatter(rs, gs[1,:], color='green')
plt.scatter(rs, gs[2,:], color='green')
plt.scatter(rs, gs[3,:], color='red')
plt.title(u"RAW画像の明るさと中心からの距離")
plt.ylim(0, 6)
plt.show()
```

これなら1次関数で近似できそうです。横軸は距離の二乗なので、距離の関数としては二次多項式になります。

やってみましょう。近似には、numpy の多項式近似機能 polyfit を使います。使い方は、

```
ps = polyfit(xs, ys, o)
```

で、xs は入力、ys は出力、o は次数です。ps には各次の係数が入ります。

```
# 係数をしまう配列を色の数分だけ準備。
par = [[], [], [], []]
# 各色ごとに1次式で近似。
for color in range(4):
    par[color] = np.polyfit(rs, gs[color, :], 1)
```

ここでpar[]には近似関数の傾きと切片が入っているはずです。確認してみましょう。

```
print(par)
```

```
[array([6.55713373e-07, 9.35608322e-01]), array([6.53200037e-07,
9.60626646e-01]), array([6.49653241e-07, 9.63338394e-01]),
array([1.00153497e-06, 9.09326661e-01])]
```

それらしい値が入っています。グラフにしてみてみましょう。

```
# 各色ごとの1次式の出力
es = [[], [], [], []]
# 各色ごとに、中心からの距離(rs)に応じた値を求める
for color in range(4):
    es[color] = par[color][0] * rs + par[color][1]
# 各色ごとにグラフ表示
plt.figure(figsize=(8, 8))
for color in range(4):
```

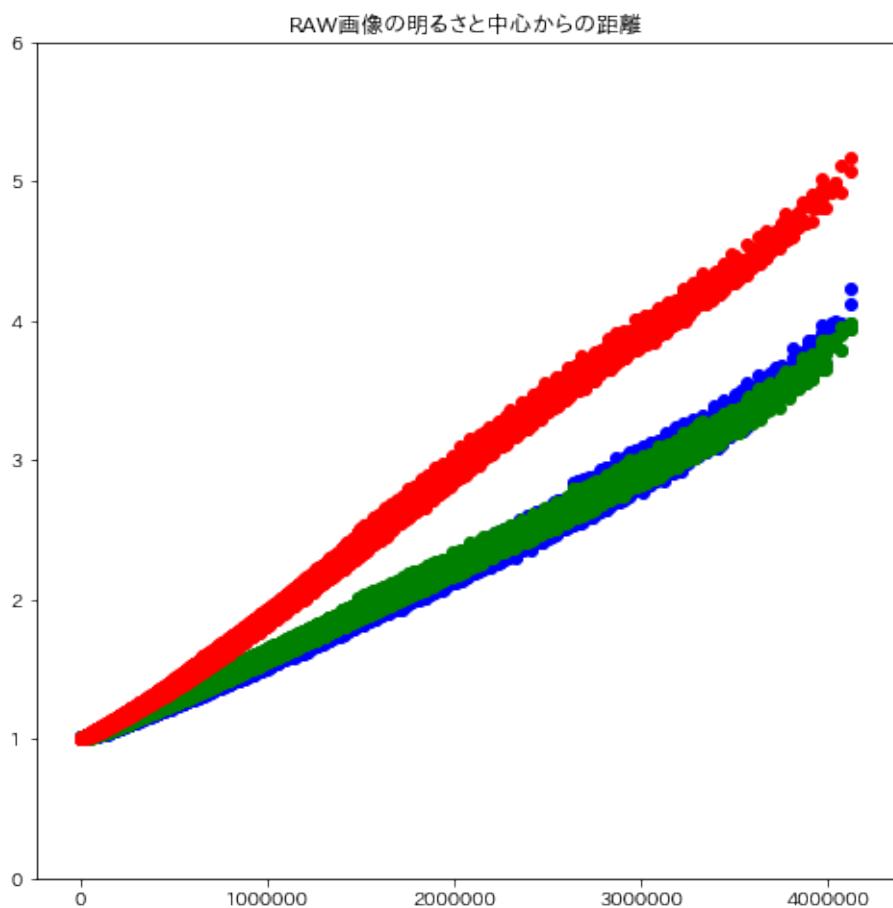


Figure 40: png

```
plt.scatter(rs, es[color])
plt.ylim(0, 6)
plt.title(u"近似した補正強度")
plt.show()
```

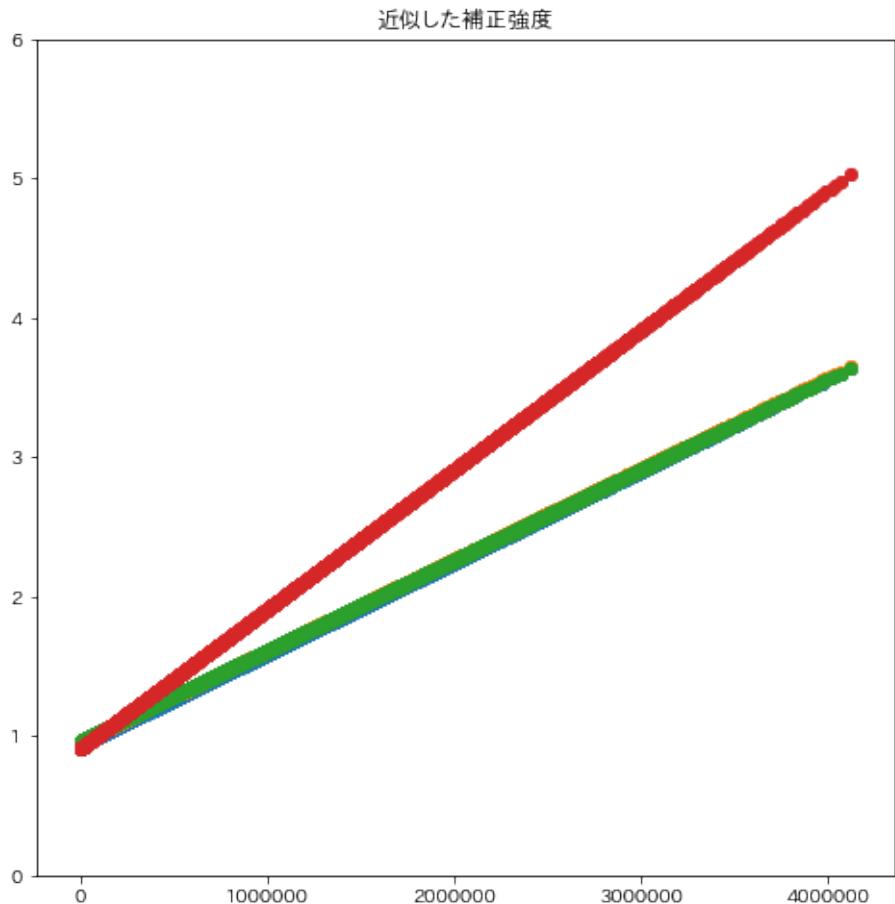


Figure 41: png

良さそうです。

レンズシェーディング補正

ではいよいよ、実際の画像のレンズシェーディングを補正してみましょう。

まず、レンズシェーディング補正前の、ブラックレベル補正のみをかけた RAW 画像がこちらです。

```
plt.figure(figsize=(8, 8))
plt.imshow(blc_raw, cmap='gray')
plt.axis('off')
plt.show()
```

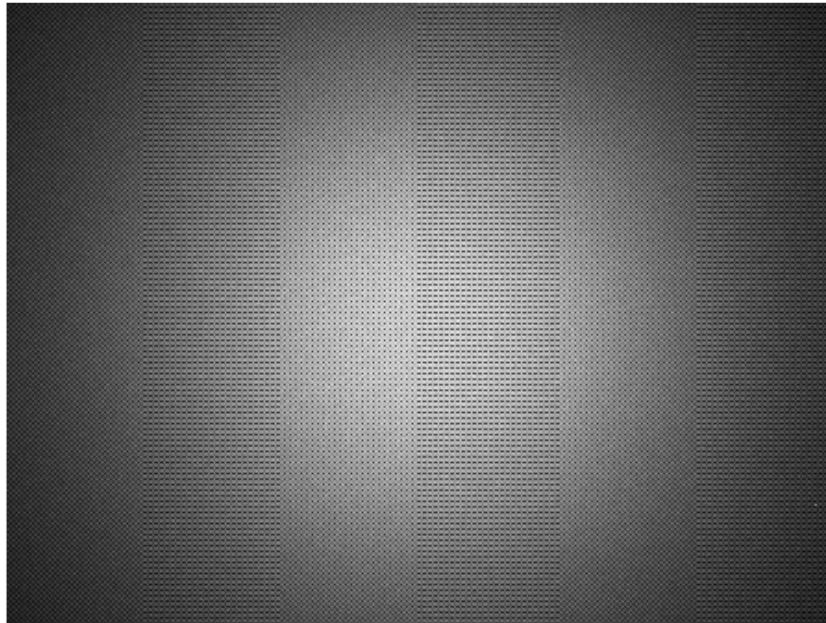


Figure 42: png

先に各画素ごとに掛け合わせるゲインを、先程の近似関数から計算しておきます。

```
gain_map = np.zeros((h, w))
center_y, center_x = h // 2, w // 2
for y in range(0, h, 2):
    for x in range(0, w, 2):
        r2 = (y - center_y) **2 + (x - center_x) **2
        gain = [par[i][0] * r2 + par[i][1] for i in range(4)]
        gain_map[y, x] = gain[0]
        gain_map[y, x+1] = gain[1]
        gain_map[y+1, x] = gain[2]
        gain_map[y+1, x+1] = gain[3]
```

このゲインをブラックレベル補正した画像にかけ合せます。

```
lsc_raw = blc_raw * gain_map
```

```
outimg = lsc_raw.copy()
outimg /= 1024
outimg[outimg < 0] = 0
outimg[outimg > 1] = 1
plt.figure(figsize=(8, 8))
plt.imshow(outimg, cmap='gray')
plt.axis('off')
plt.show()
```

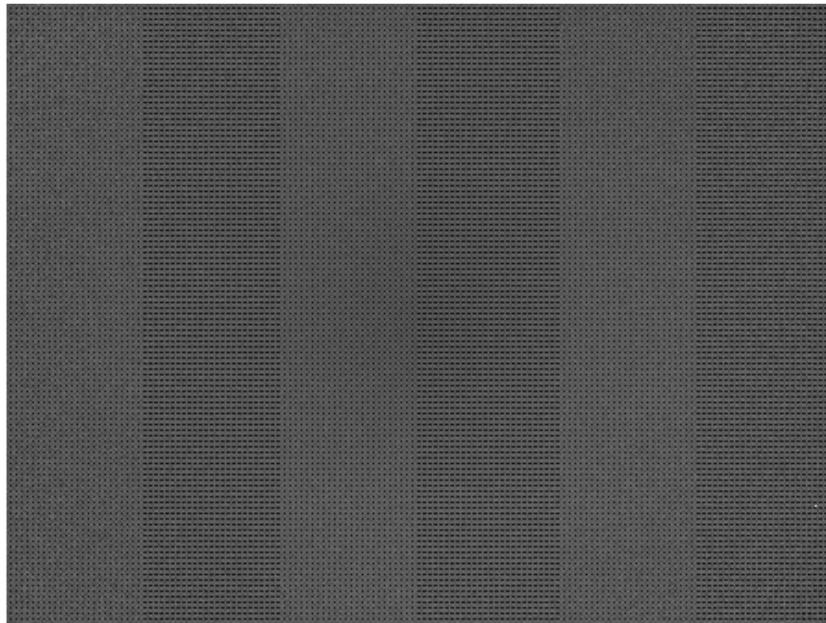


Figure 43: png

フラットな画像が出力されました！

残りの処理（ホワイトバランス補正、デモザイク、カラーマトリクス補正、ガンマ補正）を行ってフルカラー画像を出力してみましょう。

```
dpc_raw = defect_correction(lsc_raw, 16)
wb_raw = white_balance(dpc_raw, raw.camera_whitebalance,
                      raw.raw_colors)
dms_img = demosaic(wb_raw, raw.raw_colors)
color_matrix =
    np.array([6022,-2314,394,-936,4728,310,300,-4324,8126])/4096
ccm_img = color_correction_matrix(dms_img, color_matrix)
white_level = 1024
```

```

shading_img = gamma_correction(ccm_img / white_level, 2.2)

# 画像表示
plt.figure(figsize=(16, 8))
plt.imshow(shading_img)
plt.axis('off')
plt.title(u"レンズシェーディング補正後")
plt.show()

```



Figure 44: png

RGB 画像で効果が確認できました。

残っているシェーディング量を測定してみましょう。

```

#画素の明るさの横方向の分布の測定。
center_y, center_x = h // 2, w // 2
shading_after = [[], [], []]
y = center_y - 16
for x in range(0, w - 32, 32):
    xx = x + 16
    shading_after[0].append(shading_img[y:y+32, x:x+32, 0].mean())
    shading_after[1].append(shading_img[y:y+32, x:x+32, 1].mean())
    shading_after[2].append(shading_img[y:y+32, x:x+32, 2].mean())

```

```

shading_after = [np.array(a) / max(a) for a in shading_after]

plt.axis(ymin=0, ymax=1.1)
plt.plot(shading_after[0], color='red')
plt.plot(shading_after[1], color='green')
plt.plot(shading_after[2], color='blue')
plt.show()

```

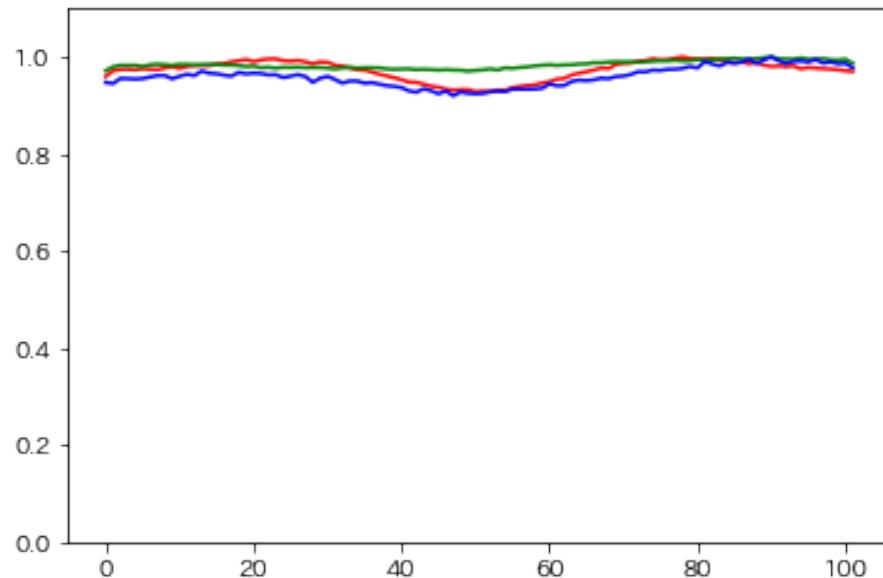


Figure 45: png

シェーディングがほぼなくなりフラットになりました。また、赤色と緑・青色との違いもほぼ消えました。成功のようです。

通常画像への適用

それではテスト用の平坦画像 (Flat Field) ではなく、実際の画像にレンズシェーディング補正を適用してみましょう。

今回もchart.jpgを使います。

```

# 画像をダウンロードします。
!if [ ! -f chart.jpg ]; then wget 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_array.shape

```

では RAW 画像データを取り出し、まずはレンズシェーディング補正なしで現像してみます。

```

#RAW現像処理
blc_raw = black_level_correction(raw_array,
    raw.black_level_per_channel, raw.raw_pattern)
dpc_raw = defect_correction(blc_raw, 16)
wb_raw = white_balance(dpc_raw, raw.camera_whitebalance,
    raw.raw_colors)
dms_img = demosaic(wb_raw, raw.raw_colors)
color_matrix =
    np.array([6022,-2314,394,-936,4728,310,300,-4324,8126])/4096
ccm_img = color_correction_matrix(dms_img, color_matrix)
white_level = 1024
no_shading_img = gamma_correction(ccm_img / white_level, 2.2)

# 画像表示
plt.figure(figsize=(16, 8))
plt.imshow(no_shading_img)
plt.axis('off')
plt.title(u"チャート画像レンズシェーディング補正前")
plt.show()

```

レンズシェーディングの影響に気をつけてみると、周辺部が若干暗くなっているのがわかると思います。また、全体に青みがかっているようです。

それでは次にレンズシェーディング補正を入れて処理してみます。補正パラメータは先程の平坦画像で計算したものを使います。

```

blc_raw = black_level_correction(raw_array,
    raw.black_level_per_channel, raw.raw_pattern)
#先程計算したゲインマップを使いシェーディング補正。
lsc_raw = blc_raw * gain_map
dpc_raw = defect_correction(lsc_raw, 16)
wb_raw = white_balance(dpc_raw, raw.camera_whitebalance,
    raw.raw_colors)
dms_img = demosaic(wb_raw, raw.raw_colors)

```

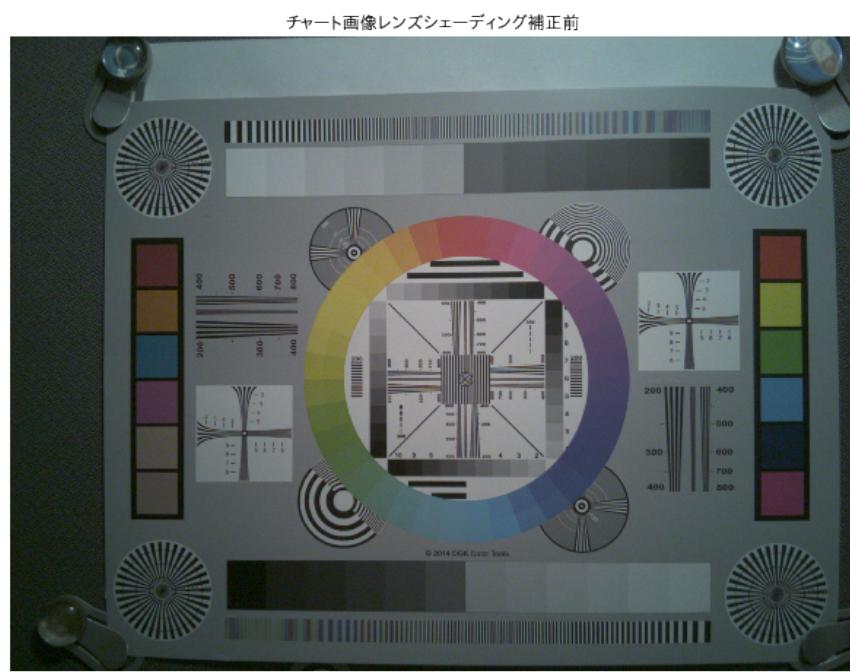


Figure 46: png

```

color_matrix =
    np.array([6022,-2314,394,-936,4728,310,300,-4324,8126])/4096
ccm_img = color_correction_matrix(dms_img, color_matrix)
shading_img = gamma_correction(ccm_img / white_level, 2.2)

# 画像表示
plt.figure(figsize=(16, 8))
plt.imshow(shading_img)
plt.axis('off')
plt.title(u"チャート画像レンズシェーディング補正後")
plt.show()

```

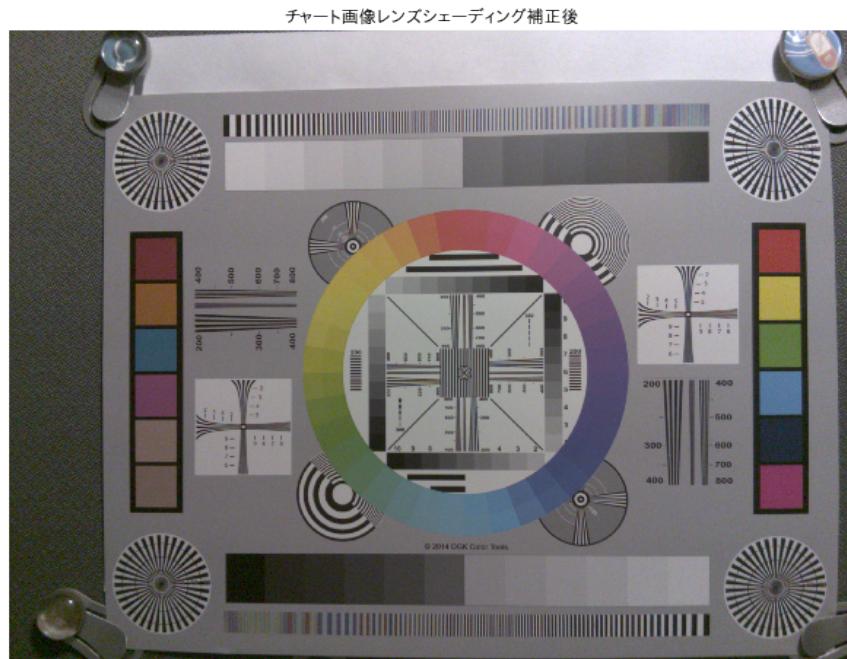


Figure 47: png

先程の画像に比べると明るさが均一になり、不自然な青みも消えました。成功のようです。

モジュールへの追加

この処理も関数としてモジュールへ追加しておきましょう。

```
def lens_shading_correction(raw_array, coef):
```

```

"""
レンズシェーディング補正を行う。

Parameters
-----
raw_array: numpy array
    Bayerフォーマット入力RAW画像
coef: array of size 4x2
    coef[c][1]: 定数項
    coef[c][0]: 傾き
    cはBayer配列内のカラーチャンネル
    c = 0: 左上
    c = 1: 右上
    c = 2: 左下
    c = 3: 右下

Returns
-----
lsc_raw: numpy array
    出力RAW画像
"""

# ゲインマップの保存場所。
gain_map = np.zeros(raw_array.shape)
# 起点となる画像の中心位置。
h, w = raw_array.shape
center_y, center_x = h // 2, w // 2
# 中心からの距離を配列に保存。
x = np.arange(0, w) - center_x
y = np.arange(0, h) - center_y
# numpyのmeshgridは, x, yを並べた配列を生成する。
# この場合範囲内の i, jに対し、xs[i, j] = x[j]
# 同様に i, jに対し、ys[i, j] = y[i]
xs, ys = np.meshgrid(x, y, sparse=True)
# 各点ごとの中心からの距離を計算
r2 = ys * ys + xs * xs
# 中心からの距離に係数をかけて各点のゲインを計算。
gain_map[::2, ::2] = r2[::2, ::2] * coef[0][0] + coef[0][1]
gain_map[1::2, ::2] = r2[1::2, ::2] * coef[1][0] + coef[1][1]
gain_map[::2, 1::2] = r2[::2, 1::2] * coef[2][0] + coef[2][1]
gain_map[1::2, 1::2] = r2[1::2, 1::2] * coef[3][0] + coef[3][1]
# 入力画像にゲインをかけ合わせる。
lsc_array = raw_array * gain_map
return lsc_array

```

正常に動作するか確認しておきます。

```

blc_raw = black_level_correction(raw_array,
    raw.black_level_per_channel, raw.raw_pattern)
lsc_raw = lens_shading_correction(blc_raw, par)
dpc_raw = defect_correction(lsc_raw, 16)
wb_raw = white_balance(dpc_raw, raw.camera_whitebalance,
    raw.raw_colors)
dms_img = demosaic(wb_raw, raw.raw_colors)
color_matrix =
    np.array([6022,-2314,394,-936,4728,310,300,-4324,8126])/4096
ccm_img = color_correction_matrix(dms_img, color_matrix)
shading_img = gamma_correction(ccm_img / white_level, 2.2)

# 画像表示
plt.figure(figsize=(16, 8))
plt.imshow(shading_img)
plt.axis('off')
plt.title(u"lens_shading_correction関数を使った画像")
plt.show()

```

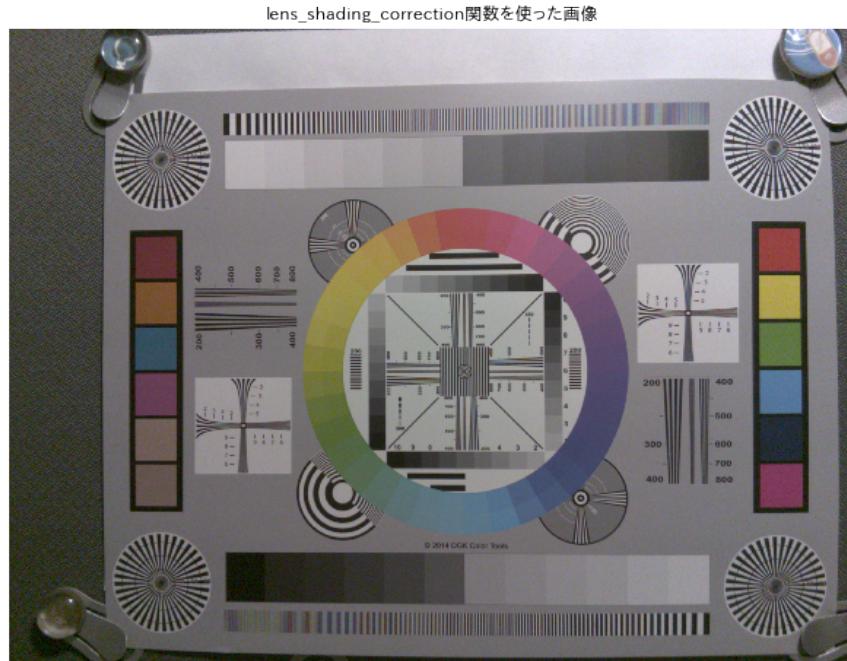


Figure 48: png

正常に処理できているようです。

まとめ

今回はレンズシェーディング補正（周辺減光補正）をとりあげました。おそらく、『カメラ』画像処理以外のいわゆる画像処理では取り上げることのない特殊な処理だと思います。

今回行ったのは、半径方向の二次多項式による補正ゲインの近似で、レンズシェーディング補正の中では最も単純なものです。実際のカメラの中では、より高次の関数による近似や、2次元ルックアップテーブルによる補正などが行われているのが普通です。また、補正パラメータも、明るさや光源の種類、オートフォーカスの場合はフォーカス位置、などにより調整します。

一見単純そうな見た目や効果と比べて、実際には確かに複雑で非常に重要な処理です。ある意味カメラの出力画像の画質を決める肝と言ってもよいと思います。

これで第4章は終わりです。次は第五章画像をきれいにする処理に入ります。

5 画質を良くする処理

5.1 この章について

はじめに

この章では画像の画質を良くする処理を紹介します。

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_chapter.ipynb

この章で扱う処理について

前章まででカメラ画像処理に最低限必要な処理の説明を行いました。

ここまででとりあえず見るに耐える画像はできましたが、画像の細部を見ていくとなんとなくぼやっとしていたり、平面のはずのところにノイズがのっていることがあります。この章では、こういった点を解決して見栄えの良い、よりきれいな画像を作り上げる処理をとりあげます。

とりあげるのは以下の処理です。- ノイズフィルター - エッジ強調 - トーンカーブ補正
最初のノイズフィルターとしてはよく知られているバイラテラルフィルターという処理をとりあげます。

次のエッジ強調では、これまたアナログの時代から知られているアンシャープマスク処理をとりあげます。

最後のトーンカーブ補正では、ヒストグラムについて簡単に説明したあとで、トーンカーブ補正処理を行ってみます。

この章でも、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。次はノイズフィルターです。

5.2 ノイズフィルター

この節未完成

この節について

この節では、ノイズフィルターについて解説します。

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、画像の読み込みを行います。今回もラズベリーパイで撮影したチャート画像を使用します。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とインポート
!if [ ! -f raw_process.py ]; wget raw_process.py; fi
from raw_process import simple_demosaic, white_balance,
    black_level_correction, gamma_correction
from raw_process import demosaic, defect_correction,
    color_correction_matrix, lens_shading_correction
```

```

# 日本語フォントの設定
!apt -y install fonts-ipafont-gothic
plt.rcParams['font.family'] = 'IPAGothic'
# もし日本語が文字化けしている場合`! rm
    /content/.cache/matplotlib/fontList.json`を実行して、
# Runtime->Restart Runtimeで再実行

# 画像をダウンロードします。
!if [ ! -f chart.jpg ]; then wget 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_array.shape

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)
/bin/sh: 1: Syntax error: "fi" unexpected (expecting "then")
E: Could not open lock file /var/lib/dpkg/lock-frontend - open (13:
    Permission denied)
E: Unable to acquire the dpkg frontend lock
    (/var/lib/dpkg/lock-frontend), are you root?

```

デジタル画像のノイズ

デジタル画像にも当然ながらノイズはあります。

ノイズ源には様々な物がありますが、ごくごく大雑把に言うと、センサーから画像データを読み出す際に付加されるリードノイズと、センサーの各セルに入ってくる光の量の統計的ゆらぎによる光ショットノイズが大きな要因になります。

もちろんこれは非常に ____ 大雑把 ____ な分類で、リードノイズには、熱ショットノイズ、固定パターンノイズ、暗電流、などがあります。

光ショットノイズは光の量に依存して、その分散は次のようなポアソン分布を取る事が知られています。

$$\sigma^2(n) = n$$

本来、光ショットノイズは光の量のみに依存するノイズのはずですが、実在する画像センサーの特性が理想的なものとは異なるために（非線形性やクロストークなど）、実際の観測では複雑な特性を見せたりします。

結局、ノイズ量も実際のセンサーやカメラ系で測定してカリブレーションする事になります。

最近のカメラ画像処理では、こういったカリブレーションによるノイズ量の推定はノイズモデルと呼ばれ重要視されています。

実際の画像のノイズ

それでは前章で処理した画像のノイズを観察してみましょう。

まずは RAW 現像します。

```
blc_raw = black_level_correction(raw_array,
    raw.black_level_per_channel, raw.raw_pattern)
lsc = [np.array([6.07106808e-07, 9.60556906e-01]),
    np.array([6.32044369e-07, 9.70694361e-01]),
    np.array([6.28455183e-07, 9.72493898e-01]),
    np.array([9.58743579e-07, 9.29427169e-01])]
lsc_raw = lens_shading_correction(blc_raw, lsc)
dpc_raw = defect_correction(lsc_raw, 16)
wb_raw = white_balance(dpc_raw, raw.camera_whitebalance,
    raw.raw_colors)
dms_img = demosaic(wb_raw, raw.raw_colors)
color_matrix =
    np.array([6022,-2314,394,-936,4728,310,300,-4324,8126])/4096
ccm_img = color_correction_matrix(dms_img, color_matrix)
white_level = 1024
gmm_img = gamma_correction(ccm_img / white_level, 2.2)

# 画像表示
plt.figure(figsize=(8, 8))
plt.imshow(gmm_img)
plt.axis('off')
plt.title(u"第 2 章までのRAW現像結果")
plt.show()
```

この画像を拡大すると、このようになっています。

第2章までのRAW現像結果

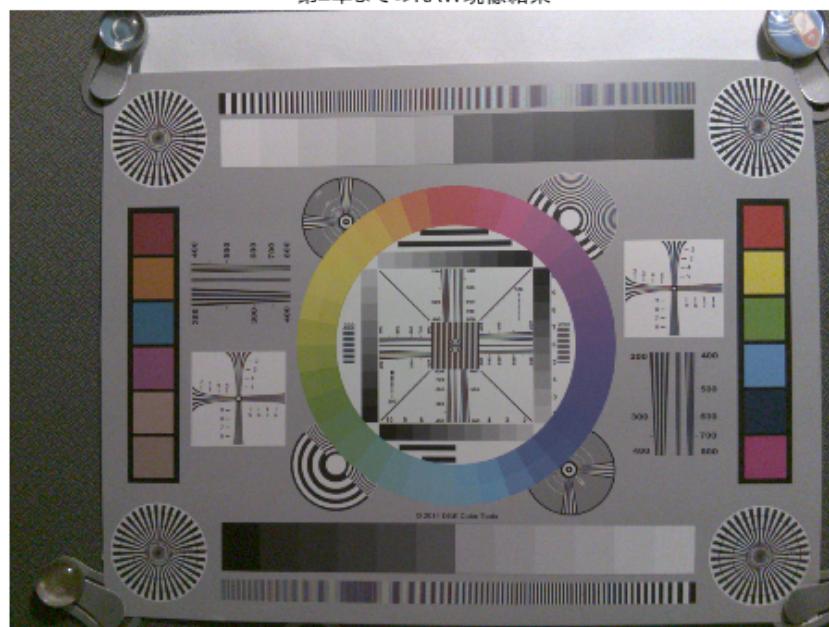


Figure 49: png

```
# 画像表示
plt.figure(figsize=(8, 8))
plt.imshow(gmm_img[1950:2150, 800:1000, :])
plt.axis('off')
plt.title(u"ノイズフィルター無し、拡大")
plt.show()
```

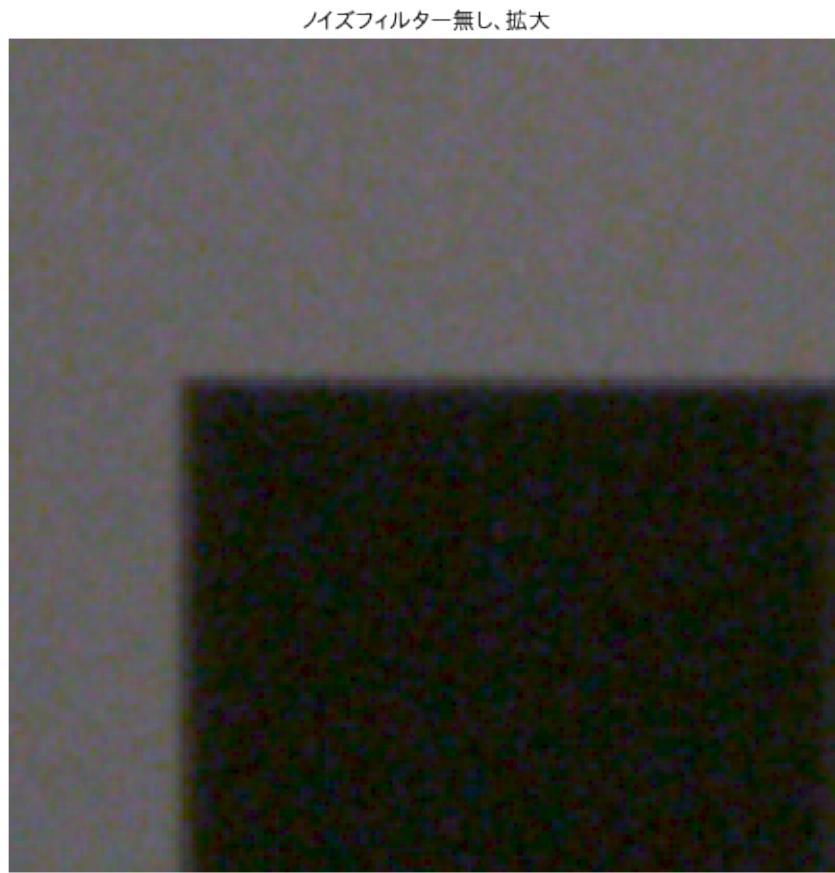


Figure 50: png

なんだかざらざらしていますし、本来グレーや黒の部分に色が浮かんでいます。なるべく明るくして撮影したのですが、まだだいぶノイズがのっているようです。

では実際どの程度の量のノイズがあるのか測定してみましょう。

図のグレイパッチ部分（赤い長方形で囲った部分）のノイズ量を実際に測定してみます。

各グレイパッチの座標を画像ビューワーなどで調べ、測定領域を決めておきます。

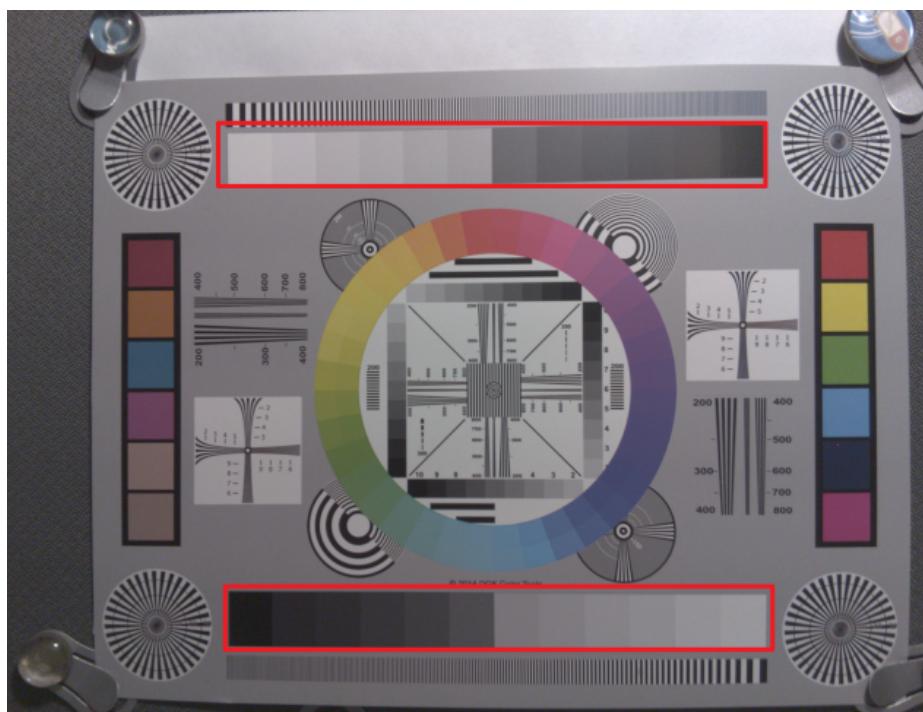


Figure 51: ノイズ測定領域

今回は次のような座標をとり、その点から右下の 100x100 画素の正方形の領域を測定します。

```
patches = [(2586, 2086), (2430, 2092), (2272, 2090), (2112, 2090),
(1958, 2086), (1792, 2094),
(1642, 2096), (1486, 2090), (1328, 2090), (1172, 2086),
(1016, 2084), (860, 2084),
(866, 482), (1022, 480), (1172, 476), (1328, 474), (1480,
470), (1634, 466),
(1788, 462), (1944, 460), (2110, 452), (2266, 452), (2424,
448), (2586, 442)]
```

各パッチ内の画素の分散と平均値を測定してみます。画像のフォーマットが Bayer なので、各色チャンネル毎に統計をとります。

```
variances = []
averages = []
for index, (dx, dy) in enumerate(((0, 0), (1, 0), (0, 1), (1, 1))):
    for patch in patches:
        x, y = patch
        p = blc_raw[y+dy:y+100:2, x+dx:x+100:2]
        s2 = (p * p).mean()
        av = p.mean()
        v = s2 - av * av
        variances.append(v)
        averages.append(av)
```

測定結果を見てみましょう。

```
plt.plot(averages, variances, linestyle='None', marker='o',
         color='blue')
plt.show()
```

どうやら、分散は画素の値に対してほぼ線形になっているようです。光ポワソンノイズが支配的だと予想されます。

一次多項式で近似してみましょう。

```
par = np.polyfit(averages, variances, 1)
print(par)
```

```
[ 0.56326801 -1.1732412 ]
```

では、近似した 1 次式とのグラフを並べてみましょう。

```
xs = np.arange(0, 300, 1)
ys = par[1] + par[0] * xs
plt.plot(averages, variances, linestyle='None', marker='o',
         color='blue')
```

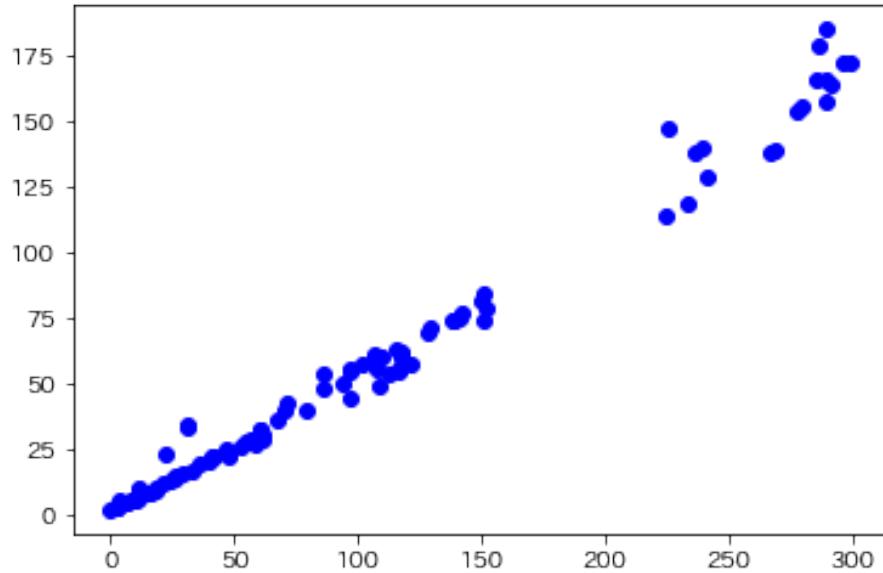


Figure 52: png

```
plt.plot(xs, ys, linestyle='None', marker='.', color='red')
plt.show()
```

ほぼ近似できているようです。

ノイズフィルターの定番バイラテラルフィルター

ノイズフィルターに求められる特性として、ノイズは取り除いてほしいが、元の画像に含まれる情報は残しておきたい、というものがあります。

このような矛盾する要求に答えるものとしてバイラテラルフィルター⁴があります。

バイラテラルフィルターのアイデアは基本的に、

1. ある画素の周辺の画素のうち、値が近いものは同じものだからフィルターをかける
2. 値が遠いものは違うものだからフィルターをかけない。

というものです。

たとえば、各ターゲット画素のまわりに次のような 5x5 の領域を設定します。

赤で囲った画素がターゲットです。

⁴Bilateral Filtering for Gray and Color Images, C. Tomasi and R. Manduchi, Proceedings of the 1998 IEEE International Conference on Computer Vision, pp. 839–846, 1998

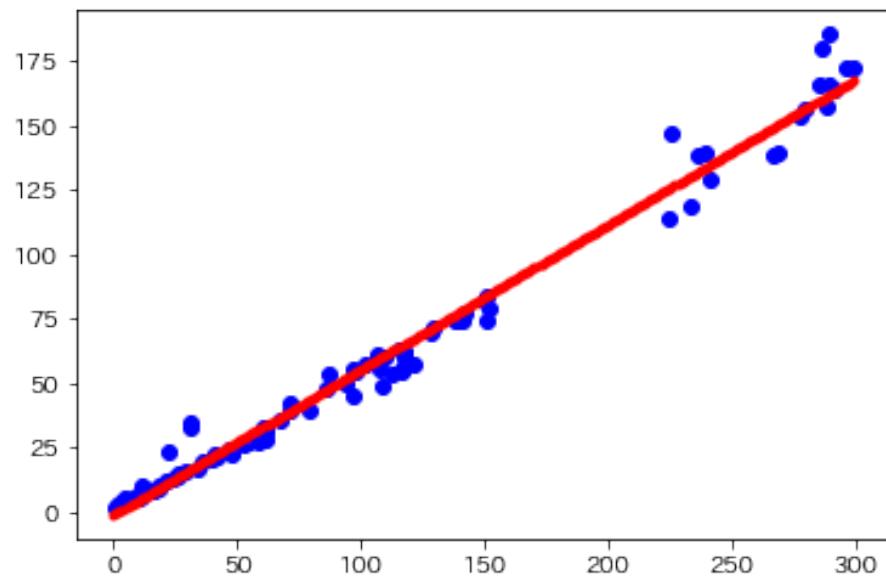


Figure 53: png

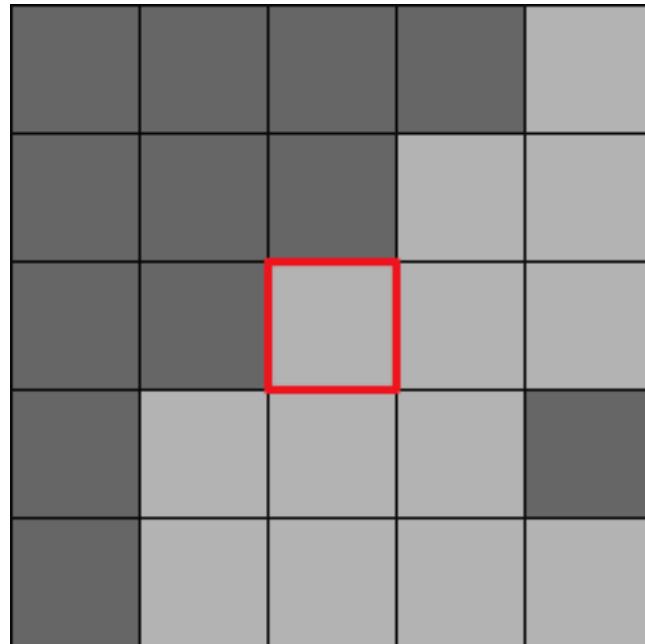


Figure 54: 注目した画素の周辺

周りの画素のうち、灰色の画素はターゲットに近い値を持ち、黒い画素は大きく異なる値を持つとします。

この場合、灰色の部分は画像のある部分（たとえば新聞の地の部分）、黒い部分は他の部分（たとえば新聞の印刷部分）に含まれると考えられます。

そうなると、灰色の画素にのみフィルターをかけ、黒い画素にはフィルターをかけないことで、画像の特徴に影響を与えるノイズフィルターをかける事ができます。

一例としてはこのようなフィルターになるでしょう。（これは説明用の図で、実際のフィルターとは異なります。）

0	0	0	0	1
0	0	0	2	1
0	0	8	4	1
0	2	4	2	0
0	1	1	1	1

Figure 55: フィルターの値

バイラテラルフィルターでは、ターゲット画素と周辺画素の一つ一つの値の差を取り、それをノイズの量とくらべて加重平均のウェイトを計算することで、ターゲット画素に近いものにのみフィルターを適用します。

式としては、例えばこうなります。

$$p_{\text{out}}(x, y) = \frac{\sum_{dy=-N}^{+N} \sum_{dx=-N}^{+N} w(dx, dy) p_{\text{in}}(x + dx, y + dy)}{\sum_{dy=-N}^{+N} \sum_{dx=-N}^{+N} w(dx, dy)}$$

$$w(dx, dy) = \exp \left(-\frac{a|p_{\text{in}}(x + dx, y + dy) - p_{\text{in}}(x, y)|^2 + b|dx^2 + dy^2|}{\sigma^2} \right)$$

ただし、 p_{in} が入力画像で、 p_{out} が出力画像、 (x, y) が座標です。 N はウインドウのサイズ、 a はチューニングパラメータで 0 以上（通常 1 以下）です。

この式によって、ターゲット画素と周辺画素は w の計算の中で比べられます。差が σ に比べて大きければ w は小さくなり、その画素のウェイトは小さくなります。逆にターゲットの差が小さい場合は w の値は 1 に近くなり、大きなウェイトを持つことになります。

もし周辺の画像が全てターゲット画像に近い場合、フィルターの形はガウシアンフィルターに近づいていきます。

一つ注意として、ノイズ分散の扱いがあります。通常の画像処理の教科書ではノイズ分散は画像内で一定という仮定を行います。もしリード・ノイズが支配的ならこの仮定は正しいのですが、先程見たとおり実際の画像ではショットノイズのためにノイズの分散は画素の値に依存します。

もしノイズフィルターをかけるのがカメラ画像処理後の RGB や YUV の画像だとすると、すでにガンマ補正やローカルトーンマップ補正などの処理が行われていて、元のノイズの分布を推定するのは困難です。RAW 画像から処理する場合はこのような処理が行われていないことはわかっているので、ショットノイズなどのノイズ量を推定するのは簡単です。

次は測定したノイズの特性を利用してノイズ処理を行っていきます。

バイラテラルノイズフィルターの適用

ではこのノイズフィルターを実際の画像に適用してみましょう。

Bayer フォーマットのままではフィルターがかけにくいので、デモザイク後の値を使用します。一応この段階の画像を確認しておきましょう。

```
outimg = dms_img / 1024
outimg[outimg < 0] = 0
outimg[outimg > 1] = 1
plt.figure(figsize=(8, 8))
plt.imshow(outimg)
plt.axis('off')
plt.title("デモザイク後の画像、ノイズフィルター無し")
plt.show()
```

さて、ホワイトバランスやデモザイク処理を行った後でも、ノイズの分散は画素の値に比例するのでしょうか？RGB の平均に対して確かめてみましょう。

```
luma_img = dms_img.mean(2)
variances = []
averages = []
for patch in patches:
    x, y = patch
    p = luma_img[y:y+100, x:x+100]
```

デモザイク後の画像、ノイズフィルター無し

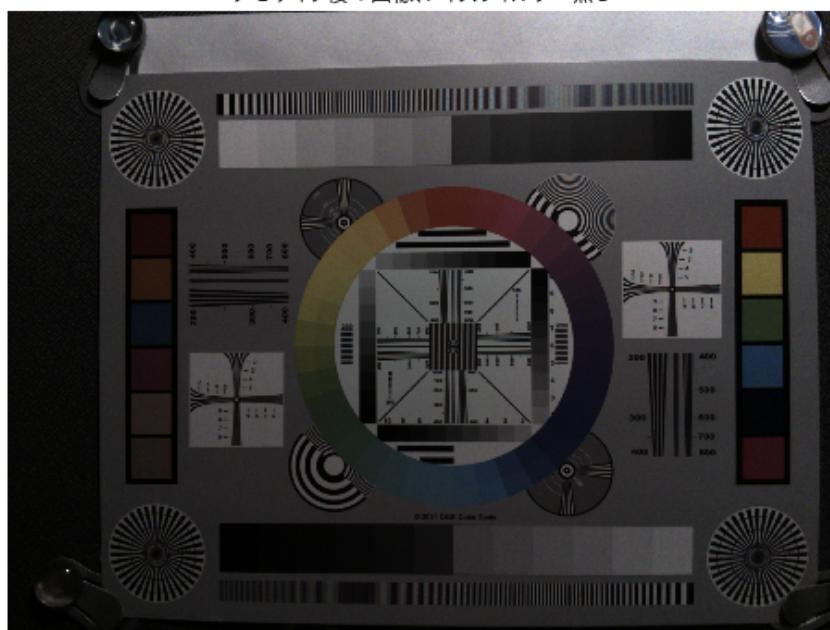


Figure 56: png

```

s2 = (p * p).mean()
av = p.mean()
v = s2 - av * av
variances.append(v)
averages.append(av)

plt.plot(averages, variances, linestyle='None', marker='o',
         color='blue')
plt.title(u"デモザイク後のノイズ分散")
plt.show()

```

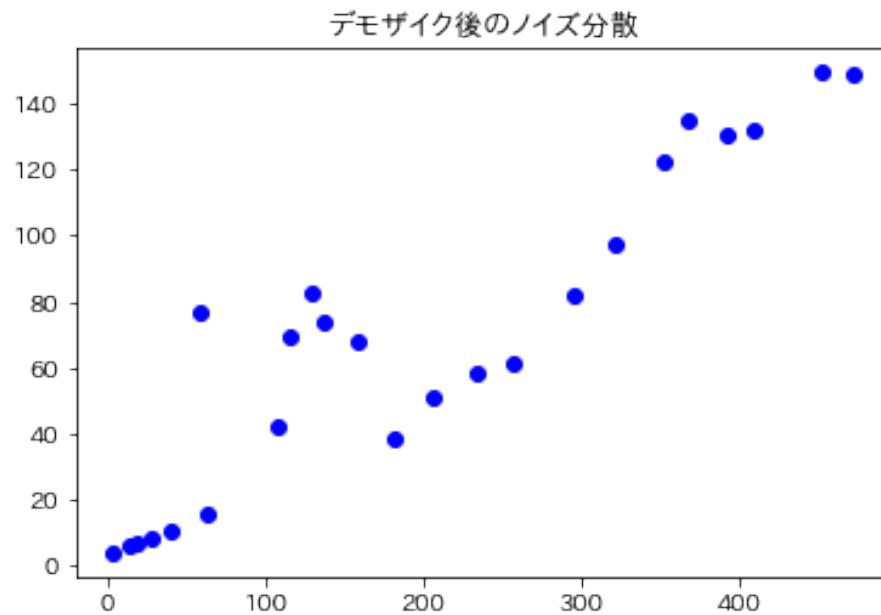


Figure 57: png

先ほどとは多少様子が違いますが、線形近似で大丈夫そうです。

傾きを求めて確かめてみましょう。

```

par = np.polyfit(averages, variances, 1)
print(u'傾き', par[0])

xs = np.arange(0, 500)
ys = par[0] * xs
plt.plot(xs, ys, color='red')
plt.plot(averages, variances, linestyle='None', marker='o',
         color='blue')

```

```
plt.title('ノイズ分散と近似式')
plt.show()
```

```
傾き 0.29300604909937544
```

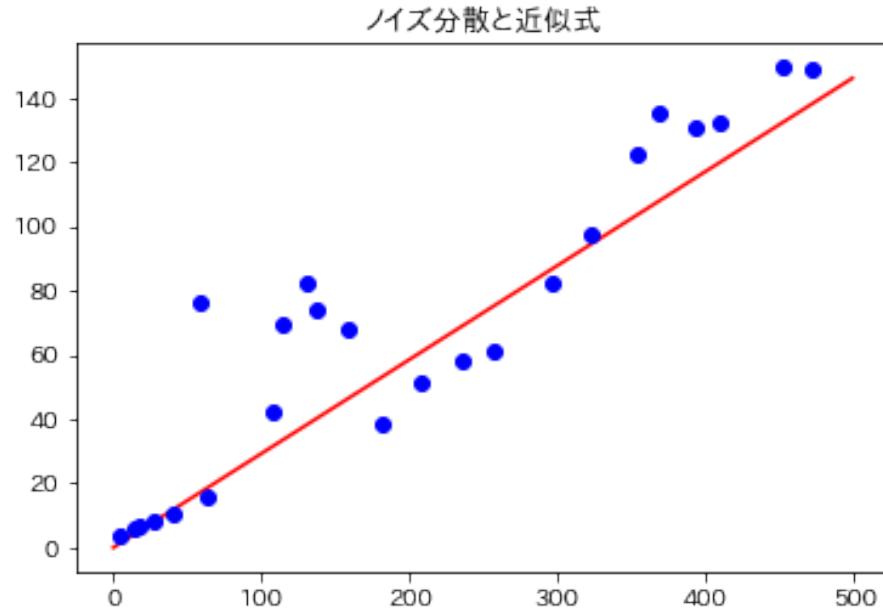


Figure 58: png

Bayer での測定に比べると誤差が大きいようですが、この程度のずれなら使えそうです。

さあ、これでノイズフィルターを適用する準備ができました。実際にかけてみましょう。今回はフルカラーなので、ウェイトの計算は RGB の平均に対して行い、フィルターの適用は各カラーごとに行うという方法を使っています。コード中、coef がバイラテラルフィルターの式の a に相当します。なお今回は $b = 0$ としました。

```
# 注：これは処理をわかりやすく書いたもので非常に実行速度が遅い。
# 実際にはこの後の高速版を使用することをおすすめする
import math

coef = 0.1
img_flt = dms_img.copy()
for y in range(2, h-2):
    for x in range(2, w - 2):
        # 5x5の平均値からノイズの分散(sigma) を推定する
        average = luma_img[y-2:y+3, x-2:x+3].mean()
        sigma = par[0] * average
```

```

sigma = sigma if sigma > 0 else 1

weight = np.zeros((5, 5))
out_pixel = np.zeros(3)
norm = 0
# 5x5内の各画素毎に重みを計算する
for dy in range(-2, 3):
    for dx in range(-2, 3):
        # 中心画素との差
        diff = luma_img[y + dy, x + dx] - luma_img[y, x]
        diff_norm = diff * diff / sigma
        # 差と分散からウェイトを計算し、加重平均値を求める
        weight = math.exp(-coef * diff_norm)
        out_pixel += weight * dms_img[y + dy, x + dx, :]
        norm += weight
# 各色毎にウェイトの和で正規化する
img_flt[y, x, 0] = out_pixel[0] / norm
img_flt[y, x, 1] = out_pixel[1] / norm
img_flt[y, x, 2] = out_pixel[2] / norm

img_flt = noise_filter(dms_img, 0.1, 0, 0.3)
outimg = img_flt.copy()
outimg = outimg / 1024
outimg[outimg < 0] = 0
outimg[outimg > 1] = 1
plt.figure(figsize=(8, 8))
plt.imshow(outimg)
plt.axis('off')
plt.title(u"デモザイク後の画像、ノイズフィルターあり")
plt.show()

```

```

NameError                                Traceback (most recent
call last)

```

```

<ipython-input-13-98646eb76b2c> in <module>()
----> 1 img_flt = noise_filter(dms_img, 0.1, 0, 0.3)
      2 outimg = img_flt.copy()
      3 outimg = outimg / 1024
      4 outimg[outimg < 0] = 0
      5 outimg[outimg > 1] = 1

```

```

NameError: name 'noise_filter' is not defined

```

このままではわかりにくいので、残りの処理（カラーマトリクスとガンマ補正）を行います。

```
ccm_flt_img = color_correction_matrix(img_flt, color_matrix)
gmm_flt_img = gamma_correction(ccm_flt_img / white_level, 2.2)

# 画像表示
plt.figure(figsize=(16, 8))
plt.imshow(gmm_flt_img)
plt.axis('off')
plt.title(u"ノイズフィルターあり")
plt.show()
```

拡大してノイズフィルターなしの画像と比較してみましょう。

```
plt.figure(figsize=(16, 8))
plt.subplot(1, 2, 1)
plt.imshow(gmm_img[1950:2150, 800:1000, :])
plt.axis('off')
plt.title(u"ノイズフィルター無し、拡大")
plt.subplot(1, 2, 2)
plt.imshow(gmm_flt_img[1950:2150, 800:1000, :])
plt.axis('off')
plt.title(u"ノイズフィルターあり、拡大")
plt.show()
```

正常に処理できているようです。他の部分も見てみましょう。

```
plt.figure(figsize=(16, 8))
plt.subplot(1, 2, 1)
plt.imshow(gmm_img[1500:1700, 1650:1850, :])
plt.axis('off')
plt.title(u"ノイズフィルター無し、拡大")
plt.subplot(1, 2, 2)
plt.imshow(gmm_flt_img[1500:1700, 1650:1850, :])
plt.axis('off')
plt.title(u"ノイズフィルターあり、拡大")
plt.show()
```

平坦部のノイズは減っていますが、ディテールの大部分は残っていることがわかります。

ノイズフィルター処理の高速化

最後にもう一点だけ、処理の高速化について触れます。この項は Python での最適化に興味のない方は読み飛ばしてください。

上記のノイズフィルターのコードはプログラムとしては動作しますが、非常に遅いコードです。

これは Python の性能上しかたのない部分もあるのですが、numpy などの機能を利用することでかなり改善できます。⁵

まず、一般的な傾向として numpy ではループの処理は遅いので、なるべく for ループを減らしたほうが高速になる場合が多いです。numpy にはこのような用途のために stride_tricks というライブラリがあります。今回はこのライブラリのas_stridedという機能を利用してループを減らしていきましょう。

まずは、RGB の平均画像 (luma_img) から分散を計算します。

```
# as_stridedのインポート
from numpy.lib.stride_tricks import as_strided
import scipy

#
#      scipyの機能を使って、5x5の平均値フィルター (uniform_filter)をかける。
# mode='mirror'は縁の部分で折り返し処理をする事をしめす。
average = scipy.ndimage.filters.uniform_filter(luma_img, 5,
                                                mode='mirror')

#
#      事前に求めたノイズモデルの切片 (par[0])と傾き (par[1])からノイズ量を求める。
sigma_map = average * par[1] + par[0]
sigma_map[sigma_map < 1] = 1
#
#      stridesはnumpyの配列で引数が1変わった時のオフセットの差（ストライド）を示す。
# この場合sigma_map[0, 0]とsigma_map[1, 0]との間のストライドはsy。
# sigma_map[0, 0]とsigma_map[0, 1]との間のストライドはsx。
sy, sx = sigma_map.strides
#
# as_stridedを使って5x5の各画素のコピーを作り出す。結果はh x w x 5 x
# 5の4次元配列。
# 例えばsigma_tile[y, x, :, :]は画素average[y,
# x]のノイズ量を5x5の配列にしたもの。
# strides=(sy, sx, 0, 0)はsigma_tile[y, x, z,
# w]のメモリーオフセットは
# offset(y,x,z,w) = sy*y+sx*x+0*z+0*wであることをしめしている。
sigma_tile = as_strided(sigma_map, strides=(sy, sx, 0, 0), shape=(h,
                                                                w, 5, 5))
#
# 縁の近くの2画素にはフィルターをかけないので、その分を除いておく。
sigma_tile = sigma_tile[2:h-2, 2:w-2, :, :]
```

ここではまず、3278 x 2444 通りの 5x5 のパッチについて、FIR フィルターを使って平均値を計算し、そこからノイズ分散をもとめています。

⁵ただし、処理の内容が一見わかりにくくなるというトレードオフがあります。これが理由で先程はベタ書きの処理を紹介しました。

次にそのノイズ分散値をコピーして同じ要素の 5x5 の行列を作り、その 5x5 の行列を 3278 x 2444 個並べています。

同様に、各パッチの平均値をコピーして同じ要素の 5x5 の行列を作り、その 5x5 の行列を 3278 x 2444 個並べます。

```
# luma_imgのストライドを求める。
sy, sx = luma_img.strides
# lumaの各画素を5x5ならべたパッチを、さらにh x
# w並べた4次元配列を作成。
luma_tile = as_strided(luma_img, strides=(sy, sx, 0, 0), shape=(h,
w, 5, 5))
# 縁の近くの2画素にはフィルターをかけないので、その分を除いておく。
luma_tile = luma_tile[2:h-2, 2:w-2, :, :]
```

ここで `as_strided` により `luma_tile[y, x, a, b]` には `luma_img[y, x]` の画素がコピーされます。a と b は 0 ~ 4 の範囲です。結果的に `luma_img` の各画素を 25 個コピーして 5x5 の配列にしたものをさらに h x w 個並べた物が `luma_tile` です。

次に、RGB の平均画像 (`luma_img`) から 5x5 のパッチを 3278 x 2444 通り作ります。

```
# luma_imgから5x5の領域を取り出したものをさらに画像サイズ分ならべている。(ただし縁の2画素はのぞむ)
# 取り出す際のメモリーオフセットはoffset(y, x, z,
# v)=sy*x+sx*x+sy*z+sx*v。
luma_box = as_strided(luma_img, strides=(sy, sx, sy, sx),
shape=(h-4, w-4, 5, 5))
```

これで `luma_box[y, x, a, b]` には `luma_img[y + a, x + b]` の画素がコピーされます。つまり `luma_box[y, x, :, :]` をとりだすと `luma_img[y, x]` の周辺の 5x5 の画素になっています。

次にこの `luma_box` から各画素にかける重みを計算します。

```
# 5x5の平均値からの差をとる。
diff = luma_box - luma_tile
# 5x5の各パッチについて、要素毎に重みを計算。
weight = np.exp(-coef * diff * diff / sigma_tile)
# 5x5のパッチ毎に重みの合計を求める。h x w サイズの配列。
weight_sum = weight.sum(axis=(2, 3))
```

これで `weight[y, x, a, b]` には `luma_box[y, x, a, b]` にかけるべき重みが、`weight_sum[y, x]` には重みの合計が入ります。

これで係数の計算はできました。各色毎に処理を行います。

```
# 赤画像処理。まずフルカラー画像から赤のプレーンを取り出す。
red = dms_img[:, :, 0]
# この画像のストライドを取り出す。
```

```

sy, sx = red.strides
#
    赤画像プレーンから、 $5 \times 5$ のパッチを全画素分とりだす（ただし縁の2画素は除く）。
red_boxes = as_strided(red, strides=(sy, sx, sy, sx), shape=(h-4,
    w-4, 5, 5))
#
    パッチと重み(weight)をかけ合わせて、パッチ毎に合計し、重みの和(weight_sum)で正規化。
red_out = (weight * red_boxes).sum(axis=(2, 3)) / weight_sum

```

`red_boxes[y, x, :, :]`には`dms_img[y, x, 0]`の周り 5×5 の赤画素が入っています。これに`weight[y, x, :, :]`をかけて和をとり、`weight_sum[y, x]`で正規化することにより、`red_out`にはフィルターされた画像がコピーされます。

同じように緑画像と青画像も処理します。

```

# 緑画像処理。処理自体は赤画像と同じ。
green = dms_img[:, :, 1]
sy, sx = green.strides
green_boxes = as_strided(green, strides=(sy, sx, sy, sx),
    shape=(h-4, w-4, 5, 5))
green_out = (weight * green_boxes).sum(axis=(2, 3)) / weight_sum

# 青画像処理。処理自体は赤画像と同じ。
blue = dms_img[:, :, 2]
sy, sx = blue.strides
blue_boxes = as_strided(blue, strides=(sy, sx, sy, sx), shape=(h-4,
    w-4, 5, 5))
blue_out = (weight * blue_boxes).sum(axis=(2, 3)) / weight_sum

```

3色の画像を組み合わせてフルカラーの画像にします。

```

# フィルターされた赤、緑、青のプレーンを元の画像にはめ込む。
img_flt2 = dms_img.copy()
img_flt2[2:h-2, 2:w-2, 0] = red_out
img_flt2[2:h-2, 2:w-2, 1] = green_out
img_flt2[2:h-2, 2:w-2, 2] = blue_out

```

画像を表示して確認しましょう。

```

outimg = img_flt2.copy()
outimg = outimg.reshape((h, w, 3))
outimg = outimg / 1024
outimg[outimg < 0] = 0
outimg[outimg > 1] = 1
plt.figure(figsize=(8, 8))
plt.imshow(outimg)
plt.axis('off')
plt.title(u"ノイズフィルター高速版")

```

```
plt.show()
```

うまくいったようです。続けてカラーマトリクス補正とガンマ補正をかけてみましょう。

```
# カラーマトリクス処理。  
ccm_flt2_img = color_correction_matrix(img_flt2, color_matrix)  
# ガンマ補正処理。  
gmm_flt2_img = gamma_correction(ccm_flt2_img / white_level, 2.2)  
  
# 最終画像表示  
plt.figure(figsize=(8, 8))  
plt.imshow(gmm_flt2_img)  
plt.axis('off')  
plt.title(u"ノイズフィルター高速版")  
plt.show()
```

先ほどと同じ部分を拡大して処理ができているか見てみましょう。

```
plt.figure(figsize=(16, 8))  
plt.imshow(gmm_flt2_img[1950:2150, 800:1000, :])  
plt.axis('off')  
plt.title(u"ノイズフィルター高速版、拡大")  
plt.show()
```

うまくいっているようです。

これでノイズ処理からループが一掃されました。処理の速度も数倍になり、実用的になりました。

まとめ

今回はノイズフィルターをとりあげ、エッジを残すノイズフィルターとしてよく使われるバイラテラル・フィルターを解説し、実装しました。

バイラテラルフィルターは移り変わりの激しい画像処理の分野では、もはや古典的ともいえるアルゴリズムです。アルゴリズムの発表から時間が立つとはいえたが基本的な考え方方は現在のノイズフィルターにも受け継がれています。

とはいって、次々に、より高性能なノイズフィルターアルゴリズムが提案されているのは間違いない、性能もどんどん進化しています。今回の内容で物足りない方は、BM3D⁶やディープラーニングを用いたノイズフィルター⁷など、さらに現代的なノイズフィルターアルゴリズムを実装されてはいかがでしょうか。

もう一つ重要な点として、画像内のノイズの解析と分散の推定を行い非常に簡単なノイズモデルを作成しました。これは画像の素性のわかっているカメラ画像処理だからできることであり、このような処理ができる事がRAW現像の利点であるともいえます。今

⁶[<http://www.cs.tut.fi/~foi/GCF-BM3D/:title>]

⁷[<https://web.stanford.edu/class/cs331b/2016/projects/zhao.pdf>]

回は光ショットノイズが支配的な画像でしたが、他のノイズが大きい画像の場合はまた違った処理が必要です。たとえば高 ISO 画像（低照度画像）ではリードノイズが相対的に大きくなっています。

また、最後に numpy による画像処理の高速化テクニックについても触れました。画像処理としては本質的ではありませんが、実用上は重要な点です。